Addressing the Challenges of I/O Variability in Post-Petascale HPC Simulations

Matthieu Dorier

THÈSE / ENS RENNES
sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE RENNES
Mention : Informatique
École doctorale MATISSE

présentée par
Matthieu Dorier
Préparée à l'unité mixte de recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

Addressing the Challenges of I/O Variability in Post-Petascale HPC Simulations

Thèse soutenue le 9 décembre 2014
devant le jury composé de :

Jean Roman */ rapporteur*
Professeur, Institut Polytechnique de Bordeaux, France

Toni Cortes */ rapporteur*
Associate Professor, Universitat Politècnica de Catalunya, Spain

Franck Cappello */ examinateur*
Senior Computer Scientist, Argonne National Laboratory, IL, USA

Jean-François Méhaut */ examinateur*
Professeur, Université de Grenoble 1, France

Gabriel Antoniu */ directeur de thèse*
Directeur de recherche, Inria Rennes - Bretagne Atlantique, France

Luc Bougé */ directeur de thèse*
Professeur, ENS Rennes, France

And word by word they handed down the light that shines today.
Alan Parsons Project (Ammonia Avenue)

Acknowledgements

I would like to start by thanking my reviewers: Toni Cortes and Jean Roman, as well as the other members of my jury: Jean-François Mehaut and Franck Cappello, for taking the time to evaluate my PhD thesis. I felt honored to see my work validated by such a committee.

This work was made possible thanks to the constant support and advice from my two PhD advisors: Gabriel Antoniu, who shared his energy and his connections with the top researchers in our field, and pushed me to publish in top conferences while providing me all the help necessary to reach these targets. I want to thank Luc Bougé as well. Before supervising this thesis, he welcomed me at ENS in 2008 as a bachelor student, hosted me for an internship in 2009 in the KerData team, and later relied on me as a teaching assistant.

I owe the work presented here to the great collaboration that is the JLPC/JLESC, and to its leaders Franck Cappello and Marc Snir. My internship under their joint supervision in 2010 decided of my future carrier. I can truly say that none of the contributions presented hereafter would have been possible without them getting me access to top supercomputers and introducing me to their professional network. I'm also very grateful to Rob Ross, with whom I worked on many occasions since 2012. He hosted me twice at ANL, where I found an excellent place to continue my work.

Many thanks go to all the other contributors: Leigh Orf for helping me understand a real climate code and getting me to evaluate Damaris on a top supercomputer (Kraken), Tom Peterka and Rob Sisneros for our collaboration around in situ visualization, Dries Kimpe for our work on CALCioM, Shadi Ibrahim and Orçun Yildiz for the most recent work on Omnisc'IO and the energy consumption in Damaris.

I would like to thank the present and former members of the KerData team (those I haven't mentioned already): Houssem, Radu, Viet-Trung, Alexandru, Tien Dat, Pierre, Lokman, Alvaro, Luis, Diana, Alexandra, and all the interns, engineers and visitors who made this team such a convivial and lively group.

I was very lucky to have the support and encouragements of my family all these years: my parents, my grand-parents, my uncle and my brothers. Most of them travelled across the country to attend my defense, this meant a lot to me.

A special thank to Aurore, for our musical breaks at INRIA in 2013.

Finally my thoughts go to all the friends I have made while working in this field, and that I'm always very happy to meet during conferences or internships: Amina, Ana, Tatiana, Leo, Thomas, Francieli, Laércio, Kassick.

I still feel like many other would deserve an acknowledgment, from those who helped me understand the code of some software, or solved technical problems, to the team assistants who helped with all the paperwork and the missions... to all these people, thank you for your support!

Résumé en Français

En 2008, la communauté du calcul hautes performances (HPC) atteignait le *Petascale* avec Roadrunner d'IBM, un supercalculateur de 122400 cœurs ayant une performance de 1.375 Petaflops (1.375×10^{15} *floating point operations per second*). La barre du million de cœurs a été atteinte en 2012 avec le supercalculateur Sequoia à LLNL et, en suivant la loi de Moore – laquelle indique que le nombre de transistors dans les systèmes de calculs de pointe double tous les 18 mois –, des supercalculateurs dits *Exascale* (atteignant 10^{18} flops) sont attendus pour 2018. Une telle puissance de calcul est mise à profit dans de nombreux domaines de recherche tels que les sciences de la Terre, la biologie, le climat ou l'astrophysique, domaines dans lesquels les simulations à large échelle sont employées pour mieux comprendre les phénomènes physiques qui nous entourent. Ces simulations ont vocation de remplacer des expériences réelles qui peuvent être trop coûteuses, trop dangereuses ou simplement irréalisables, comme les études portant sur la jeunesse de l'univers. *"We have this problem in astrophysics that we can't go and do experiments in the lab to test our theories"*,[1] explique Mark Vogelsberger, du MIT, dans une interview pour The Guardian[2] au sujet des résultats d'une récente simulation de l'univers.

Les entreprises de production utilisent également les supercalculateurs pour diminuer leurs coûts de conception. Par exemple, l'utilisation de souffleries virtuelles (c'est-à-dire simulées par ordinateur) a permis à Boeing de réduire à 11 le nombre de prototypes d'ailes d'avions construits pour leur modèle 787 "Dreamliner" en 2005, en comparaison des 77 prototypes utilisés pour la conception du modèle 767 dans les années 80.[3] Les simulations hautes performances ont en effet l'avantage d'être plus rapides et moins chères que la conception et les tests de prototypes réels. De plus, ces simulations peuvent être reproduites et les modèles virtuels peuvent être évalués dans des conditions variées avec une très grande précision.

Mais comme Ken Batcher le dit, *"a supercomputer is a device for turning compute-bound problems into I/O-bound problems"*.[4] En effet, de plus grosses machines mènent à une production accrue de données. Ces données doivent être stockées et traitées efficacement en vue d'en tirer un résultat scientifique. L'approche traditionnelle de gestion de données consiste à sto-

[1] Traduction : "En astrophysique, nous avons ce problème de ne pas pouvoir effectuer d'expériences en laboratoire pour tester nos théories."

[2] www.theguardian.com/science/2014/may/07/universe-recreated-computer-simulation-model-big-bang

[3] www.scientificamerican.com/article/big-computers-for-little/

[4] Traduction : "un supercalculateur est un appareil transformant un problème limité par les performances de calculs en un problème limité par les performances des entrées/sorties".

cker les données produites par la simulation dans des fichiers pendant que celle-ci s'exécute, et à analyser ces fichiers plus tard, lorsque la simulation est terminée. On observe cependant un fossé de plus en plus profond entre les performances des systèmes de stockage et les performances des systèmes de calculs dans les supercalculateurs récents. Par exemple, alors que le supercalculateur Jaguar à ORNL (premier du Top 500 en novembre 2009 et Juin 2010) fournit un débit de 240 Go/s vers son système de stockage, pour une performance de pointe de 1,75 Petaflops, son successeur Titan (premier au Top 500 en novembre 2012) fournit un débit de stockage seulement six fois supérieur (1,4 To/s) pour une puissance de calculs dix fois supérieure (17,59 Petaflops). Ce fossé rend obsolètes les approches traditionnelles pour les entrées-sorties (E/S), qui prennent en effet une part grandissante du temps d'exécution des applications et sont sujettes à une *variabilité* croissante de leurs performances.

D'une part, il devient donc nécessaire d'optimiser la pile d'E/S à tous les niveaux, de la simulation jusqu'au système de fichiers, dans le but d'en améliorer les performances ainsi que la predictibilité de ces performances. Cela implique également d'améliorer la manière de gérer une concurrence croissante au niveau du système de fichiers, non seulement entre des centaines de milliers de processus constituant une seule application, mais également entre un nombre croissant d'applications qui s'exécutent sur la même machine et en partagent le système de stockage.

D'autre part, il devient inévitable de rapprocher les tâches d'analyse et de visualisation de la simulation elle-même afin d'éviter de stocker de larges quantités de données. Cette tendance soulève de nouveaux défis liés aux moyens dont disposent les simulations pour communiquer efficacement leurs données et partager ces dernières avec les outils d'analyse sans dégrader leurs performances.

Enfin, la consommation énergétique des futurs supercalculateurs est un problème de plus en plus important dans la communauté HPC. Alors que les machines actuelles consomment une puissance d'environs 10 MW, la Defense Advanced Research Projects Agency (DARPA) a imposé une limite de 20 MW pour les futures machines Exascale. Cela représente une multiplication par deux de la consommation d'énergie pour des plateformes qui devront être mille fois plus performantes en termes de calculs. Cette contrainte ne pourra être satisfaite seulement par des améliorations matérielles, mais nécessitera par la conception d'approches logicielles plus économes en énergie. Les mouvements et le stockage de grandes masses de données constituent notamment des tâches coûteuses en énergie et doivent être optimisées en conséquence.

Contributions

De nombreux problèmes de performance dans le contexte des simulations HPC proviennent en réalité d'un problème de *variabilité*. Des différences dans le temps de complétion des tâches d'E/S d'un processus à un autre au sein d'une application massivement parallèle forcent tous les processus à attendre le plus lent d'entre eux. Ces processus gâchent ainsi un temps précieux et de l'énergie. Dans les approches actuelles de gestion des données, cette variabilité a de multiples causes. Premièrement, dans la majorité des applications les tâches d'E/S sont exécutées de manière périodique par tous les processus en même temps, ce qui produit des pics d'activité au sein du système de fichiers. Ce comportement engendre des conflits d'utilisation des ressources et une variabilité des performances des E/S de chaque

processus, produisant un impact négatif sur les performances générales de l'application. Deuxièmement, coupler les simulations avec des outils de visualisation et d'analyse accroît cette variabilité, notamment lorsque les tâches de visualisation sont exécutées de manière interactive. Enfin une troisième source de variabilité provient des accès concurrents au système de stockage par plusieurs applications indépendantes s'exécutant sur la même machine.

Dans cette thèse, nous nous proposons de relever les défis posés par la variabilité des performances des approches actuelles de gestion de données. Notre travail a été principalement conduit dans le contexte du JLPC (Joint Laboratory for Petascale Computing), un laboratoire commun à l'Inria, le National Center for Supercomputing Applications (NCSA) à l'Université d'Illinois à Urbana-Champaign (UIUC) et Argonne National Laboratory (ANL). Ce travail a mené à différents projets communs entre l'équipe KerData, ANL et UIUC : FACCTS (France and Chicago Collaborating in the Sciences), PUF (Partner University Fund), l'équipe associée Data@Exascale. Les contributions de cette thèse peuvent être résumées comme suit.

Utilisation de cœurs dédiés pour cacher la variabilité des E/S

Alors que le nombre de cœurs dans les nœuds multicœurs augmente, des conflits apparaissent lorsque plusieurs processus d'un même nœud tentent d'accéder à la même interface réseau. Ces conflits entraînent une variabilité substantielle des performances des E/S. On appellera *jitter* ce type de variabilité, qui trouve sa source dans les conflits d'accès au sein des processus d'une même application. Dans ce travail, nous proposons une nouvelle méthode de gestion des E/S, appelée Damaris, qui se sert de cœurs dédiés aux E/S sur chaque nœud multicœur, ainsi que de mémoire partagée. Damaris permet réaliser les tâches de traitement de données et d'E/S de manière asynchrone, et de cacher la variabilité de ces dernières en conséquence. Nous évaluons Damaris sur trois plateformes différentes, notamment le supercalculateur Kraken (11^{e} du Top500 au moment des expériences) avec la simulation atmosphérique CM1. En permettant le recouvrement des E/S et des calculs, et en regroupant les données dans des fichiers plus volumineux tout en évitant les synchronisations entre cœurs, notre solution apporte un certain nombre d'avantages.

1. Elle permet de cacher complètement le jitter ainsi que tous les coûts liés aux E/S, rendant les performances de la simulation prévisibles ;

2. Elle multiplie le débit en écriture par un facteur allant jusqu'à 15 en comparaison des approches standard ;

3. Elle permet un passage à l'échelle de la simulation (testée jusqu'à 9000 cœurs) à l'inverse des approches standards qui ne passent pas à l'échelle ;

4. Elle permet d'atteindre un taux de compression des données de 600 % sans surcoût pour l'application, menant à une réduction majeure de l'espace de stockage nécessaire.

En plus d'initier le développement d'une implémentation de Damaris, ce travail a été récompensé du 2^{e} prix à l'ACM Student Research Competition qui s'est tenue en marge de la conférence ICS '11. Ce travail a également mené à une publication à la conférence CLUSTER '12.

Extension de l'usage des cœurs dédiés à la visualisation in situ

Réduire la quantité de données stockées par les simulations va devenir d'une importance critique pour les prochaines générations de supercalculateurs. En conséquence, de nombreuses recherches tentent de promouvoir des approches dans lesquelles les tâches d'analyse et de visualisation sont exécutées in situ, c'est-à-dire à proximité de la simulation et en partageant les ressources de cette dernière. Ces approches possèdent l'avantage d'éviter de stocker de grandes quantités de données pour des post-traitements. Elles peuvent cependant avoir un impact important sur le temps d'exécution de la simulation si elles ne sont pas implémentées correctement. Notre travail se concentre précisément sur le cas d'une visualisation in situ où le code de visualisation est co-localisé avec le code de la simulation et s'exécute sur les mêmes ressources. Il est important pour une telle technique de visualisation in situ de nécessiter le moins de modifications possibles dans les codes existants, d'être adaptable et d'avoir un faible impact à la fois sur le temps d'exécution et sur l'utilisation des ressources. Nous accomplissons cela grâce à Damaris/Viz, une extension de Damaris fournissant un support pour la visualisation in situ. L'utilisation de Damaris comme passerelle vers des codes de visualisation existants permet

1. de réduire les modifications de code au maximum dans les simulations existantes,
2. de réunir les fonctionnalités de divers outils de visualisation pour offrir une interface unifiée de gestion de données,
3. d'utiliser efficacement des cœurs dédiés pour cacher l'impact de la visualisation in situ sur le temps d'exécution de la simulation, et
4. d'utiliser efficacement la mémoire au travers d'une couche de communication basée sur de la mémoire partagée.

Damaris/Viz est évalué sur Blue Waters et Grid'5000 pour visualiser les données produites par la simulation atmosphérique CM1 et la simulation de dynamique des fluides Nek5000. Ce travail a mené à une publication à la conférence LDAV '13.

Analyse des compromis energie/performance dans diverses approches d'E/S

Un défi majeur pour les futures machines Exascale consiste à atteindre de hautes performances tout en maintenant une faible consommation d'énergie. Beaucoup de travaux récents, et en particulier la première contribution de cette thèse, ont exploré de nouvelles approches pour les E/S visant à réduire le goulot d'étranglement présent au niveau des E/S dans les applications large échelle (permettant ainsi d'améliorer leurs performances). Les travaux évaluant l'impact de ces approches sur la consommation d'énergie restent pourtant rares. Néanmoins, les approches qui permettent un recouvrement des E/S et des calculs ont un effet bénéfique en termes de variabilité des performance et *a fortiori* en termes de consommation d'énergie. Dans ce travail, nous avons complété notre implémentation de Damaris en lui donnant la possibilité d'utiliser des nœuds dédiés à la place de cœurs dédiés, ainsi que la possibilité d'exécuter les tâches de traitement des E/S en mode synchrone, c'est-à-dire sans ressources dédiées. Nous examinons ces différentes approches d'E/S à l'aide d'expériences avec la simulation CM1. Nos résultats, obtenus sur Grid'5000, montrent l'impact sur

les performances et sur la consommation d'énergie de ces différentes approches. Ils mettent également en évidence les relations entre la consommation d'énergie et certains paramètres de l'application et du matériel. Nous proposons ensuite un modèle mathématique permettant d'estimer la consommation d'énergie d'une simulation en fonction de l'approche utilisée pour ses E/S. Ce travail a été en partie publié au workshop DIDC '14, tenu en marge de la conférence HPDC '14.

Atténuation des conflits d'E/S par coordination inter-application

De plus grosses machines étant inévitablement exploitées par un plus grand nombre d'applications de manière concurrente, les interférences produites par plusieurs applications accédant à un système de fichiers parallèle partagé deviennent un problème majeur. Les interférences perturbent souvent les optimisations des E/S utilisées par les applications individuellement, tels que les accès optimisés au préalable pour améliorer la localité des accès disques. Ceci a pour effet de dégrader les performances des E/S de ces applications, d'accroître leur temps d'exécution et la variabilité de ce temps d'exécution, réduisant d'autant plus l'efficacité globale de la machine. Pour résoudre ce problème, nous proposons CALCioM, une approche ayant pour but d'atténuer les interférences d'E/S au travers de la sélection dynamique d'une stratégie d'ordonnancement dépendant d'informations fournies par les applications elles-mêmes. CALCioM permet à plusieurs applications s'exécutant sur un supercalculateur de coordonner leurs stratégies d'E/S en vue d'éviter d'interférer les unes avec les autres. Dans ce travail, nous examinons quatre stratégies qui peuvent être implémentées par CALCioM : sérialiser, interrompre, interférer ou coordonner. Nos expériences sur le supercalculateur BG/P Surveyor d'Argonne ainsi que sur plusieurs sites de Grid'5000 montrent comment CALCioM peut être utilisé pour implémenter des stratégies d'ordonnancement entre des applications qui autrement interfèreraient, avec pour objectif l'optimisation de l'efficacité globale de la machine. Ce travail, partiellement effectué durant un stage de 3 mois à ANL, a mené à une publication à la conférence IPDPS '14.

Prédiction des motifs spatiaux et temporels d'E/S des applications HPC

De nombreuses optimisations des E/S, tels que le préchargement, la mise en cache ou l'ordonnancement, ont été proposées pour améliorer les performances de la pile d'E/S. Afin d'optimiser ces techniques, modéliser et prédire les caractéristiques spatiales et temporelles des E/S des applications HPC alors qu'elles s'exécutent s'avèrent crucial. Dans cette direction, nous proposons Omnisc'IO, une approche ayant pour but de faire un pas en avant vers une gestion intelligente des E/S des applications HPC sur les futures plateformes. Omnisc'IO construit un modèle basé sur des grammaires formelles des E/S de n'importe quelle application HPC. Il utilise ensuite ce modèle pour prédire quand les futures opérations d'E/S vont se produire ainsi que la quantité et la localisation des données en jeu. Omnisc'IO est intégré de manière transparente dans les couches POSIX et MPI-I/O et ne nécessite aucune modification dans les sources des applications ou des bibliothèques d'E/S de haut niveau. Il ne nécessite pas d'information *a priori* sur les applications, et converge vers un modèle permettant des prédictions précises en seulement quelques itérations. L'implémentation d'Omnisc'IO est efficace à la fois en temps et en mémoire. Omnisc'IO a été évalué avec quatre applications HPC réelles – CM1, Nek5000, GTC, and LAMMPS – utilisant différentes

bibliothèques d'E/S allant de POSIX à Parallel HDF5. Nos expériences montrent qu'Omnisc'IO peut atteindre une précision allant de 79.5 à 100% pour la prédiction des paramètres spatiaux des futurs accès, et une précision moyenne de la date de ces futurs accès allant de 0.2 secondes à moins d'une milliseconde. Ce travail a mené à une publication à la conférence SC '14 et a donné lieu au développement de la bibliothèque Omnisc'IO.

Publications

Conférences Internationales

- Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé, **Matthieu Dorier**. *BlobSeer : Bringing High Throughput under Heavy Concurrency to Hadoop Map/Reduce Applications*, Proceeding of the 2010 IEEE International Parallel & Distributed Processing Symposium **(IPDPS '10)**, *Atlanta, septembre 2010*. CORE Rank A (taux d'acceptation de 24%).
- **Matthieu Dorier**, Gabriel Antoniu, Franck Cappello, Marc Snir, Leigh Orf. *Damaris : How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O*, Proceedings of the 2012 IEEE International Conference on Cluster Computing **(CLUSTER '12)**, *Pékin, septembre 2012.* CORE Rank A (taux d'acceptation de 28%).
- **Matthieu Dorier**, Roberto Sisneros, Tom Peterka, Gabriel Antoniu, Dave Semeraro. *Damaris/Viz, a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework*, Proceedings of the 2013 IEEE Symposium on Large Data Analysis and Visualization **(LDAV '13)**, *Atlanta, octobre 2013.* (taux d'acceptation de 37%).
- **Matthieu Dorier**, Gabriel Antoniu, Rob Ross, Dries Kimpe, Shadi Ibrahim. *CALCioM : Mitigating I/O Interference in HPC Systems through Cross-Application Coordination*, Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium **(IPDPS '14)**, *Phoenix, mai 2014.* CORE Rank A (taux d'acceptation de 21%).
- **Matthieu Dorier**, Shadi Ibrahim, Gabriel Antoniu, Rob Ross. *Omnisc'IO : A Grammar-Based Approach to Spatial and Temporal I/O Patterns Prediction*, Proceedings of ACM/IEEE 2014 Supercomputing Conference **(SC '14)**, *La Nouvelle Orléans, novembre 2014.* CORE Rank A (taux d'acceptation de 21%).

Workshops dans des Conférences Internationalles

- Orçun Yildiz, **Matthieu Dorier**, Shadi Ibrahim, Gabriel Antoniu. *A Performance and Energy Analysis of I/O Management Approaches for Exascale Systems*, in Proceedings of the 2014 Data-Intensive Distributed Computing **(DIDC '14)** workshop, tenu conjointement avec le 23^{rd} International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC '14), *Vancouver, juin 2014.*

Posters at International Conferences

- **Matthieu Dorier**. *Damaris - Using Dedicated I/O Cores for Scalable Post-petascale HPC Simulations*, 2011 ACM/SIGARCH International Conference on Supercomputing **(ICS '11)**, *Tucson, avril 2011.* 2^{eme} **prix de l'ACM Student Research Competition.**

- **Matthieu Dorier**. *Efficient I/O using Dedicated Cores in Large-Scale HPC Simulations*, 2013 IEEE International Parallel & Distributed Processing Symposium **(IPDPS '13)** : PhD Forum, *Boston, mai 2013.*

Logiciels

- **Damaris** est un intergiciel pour les noeuds multicœurs, leur permettant de gérer efficacement les transfères de données pour le stockage ou la visualisation en dédiant un sous-ensemble des cœurs aux opérations d'entrées/sorties (E/S). Il permet des E/S asynchrones efficaces et cache tout surcoût lié aux opérations d'E/S, tels que la compression de données, le post-traitement ou la visualisation in situ (via son extension Damaris/Viz). Damaris a été évalué sur Blue Waters (Cray XE6, NCSA), Kraken (Cray XT5, NICS), Titan (Cray XK7, ORNL), Intrepid (IBM BlueGene/P, ANL), Grid'5000 (grille de calculs française), Blue Print (cluster Power5, NCSA), avec la simulation atmosphérique CM1 et la simulation de dynamique des fluides Nek5000. Damaris a été formellement validé pour une utilisation sur le supercalculateur Blue Waters an NCSA. A notre connaissance, il a été utilisé par plusieurs chercheurs du NCSA, de Central Michigan University et de l'Université Fédérale de de Rio Grande do Sul (UFRGS).
  Lien : http://damaris.gforge.inria.fr
  Taille and langage(s) : 19500 lignes, C++, Fortran, XML
  License : LGPL

- **Omnisc'IO** est un intergiciel intégré dans les couches POSIX et MPI-I/O et permet de capturer les E/S des applications HPC de manière transparente, d'en produire un modèle et d'utiliser ce modèle pour prédire les futurs accès. Omnisc'IO est basé sur des grammaires formelles et utilise une version modifiée de l'algorithme Sequitur. Elle a été utilisé sur Grid'5000 avec la simulation atmosphérique CM1, la simulation de dynamique moléculaire LAMMPS, la simulation de fusion GTC et la simulation de dynamique des fluides Nek5000. **Lien :** http://omniscio.gforge.inria.fr
  Taille et langage(s) : 4400 lines, C++
  License : LGPL

- **Darshan-Ruby/Darshan-Web** Darshan-Ruby est une bibliothèque Ruby permettant de lire les fichiers produits par Darhan (outil de trace d'E/S produit par Argonne National Lab), en utilisant un paradygme orienté objets. Darshan-Ruby a été développé dans le but de simplifier l'analyse du comportement d'E/S des applications à grande échelle. Il accède directement au contenu des fichiers Darshan sans nécessiter de conversion en format text. Darshan-Ruby est accessible comme Ruby Gem sur le dépot officiel rubygems.org. Darshan-Ruby a mené au développement du projet Darshan-Web, qui propose une plateforme web pour aider à l'analyse des traces d'E/S. **Lien :** http://darshan-ruby.gforge.inria.fr
  Taille and langage(s) : 400 lines, C, Ruby
  License : LGPL

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Contents

1.1 Context

In 2008, the high-performance computing (HPC) community reached *Petascale* capabilities with IBM's Roadrunner, a 122,400 core supercomputer with a peak performance of 1.375 Petaflops (1.375×10^{15} *floating point operations per second*) [130]. Million-core machines have become a reality in 2012 with LLNL's Sequoia supercomputer and, following Moore's law, which states that the number of transistors in cutting-edge computing systems doubles every 18 months, *Exascale* supercomputers (capable of 10^{18} flops) are expected by 2018 [127]. Such an immense computational power is used in many research areas, including earth sciences, biology, climate, or cosmology, where large scale simulations are conducted to better understand the physical phenomena that surround us. These simulations aim to replace real experiments that are either too expensive, too dangerous or simply unfeasible, such as studies of the early universe: *"We have this problem in astrophysics that we can't go and do experiments in the lab to test our theories"* says Mark Vogelsberger, from MIT, in an interview for The Guardian [112] on the results of a recent simulation of the universe [40, 136].

The manufacturing sector also uses supercomputers to decrease design costs. For example, the use of virtual (i.e., numerically simulated) wind tunnels allowed Boeing to reduce

to 11 the number of wing prototypes effectively constructed for their 787 "Dreamliner" aircraft in 2005, in contrast with the 77 prototypes used in the design of the 767 model back in the 1980s [65]. High performance simulations have indeed the benefits of being faster and cheaper than designing actual prototypes. Besides, HPC simulations can be reproduced, and virtual models can be evaluated in various conditions with very high accuracy.

But as Ken Batcher stated, *"a supercomputer is a device for turning compute-bound problems into I/O-bound problems"*. Indeed, larger machines lead to the production of larger amounts of data that have to be efficiently stored and processed in order to retrieve scientific insights. The traditional approach to data management consists of storing the output of the simulation in files during its run, move these files and analyze them later offline. Yet we observe an increasing gap between the performance of storage systems and the computation capabilities of recent supercomputers. For instance, while ORNL's Jaguar machine (ranked 1^{st} in the Top 500 list of supercomputers [130] in November 2009 and June 2010) provided 240 GB/s of storage throughput for a peak performance of 1.75 Petaflops, its successor Titan (ranked 1^{st} in November 2012) was subject to a tenfold improvement of performance (achieving 17.59 Petaflops) for only a sixfold increase of storage throughput (achieving 1.4 TB/s). This gap makes traditional approaches to I/O (input/output) unsustainable, as they take an increasingly larger portion of the application's run time and lead to a *variability* of this run time.

On one hand, it becomes necessary to optimize the I/O stack at every level, from the simulation down to the file system, in order to improve I/O performance together with the predictability of these performances. This also involves improving the way storage systems deal with a higher degree of concurrency, not only from the hundreds of thousands of processes that constitute a single application, but also from many applications concurrently running on the machine.

On the other hand, bringing data analysis and visualization tasks closer to the simulation will become inevitable to avoid storing massive amounts of data in the near future. This rises challenges in the way simulations can efficiently communicate and share data without impacting their performance.

Finally, the energy consumption of next-generation supercomputers is a rising concern in the HPC community. While current Petascale machines run at around 10 MW, the US's Defense Advanced Research Projects Agency (DARPA) has set a 20 MW power budget for Exascale machines [52]; a twofold increase of energy consumption for a thousandfold increase of computation performance. This target will be achieved not only through hardware improvements, but also with novel, energy-efficient software approaches. In particular, data movements and storage constitute some of the most energy-demanding tasks in high performance computing, and must now evolve with energy-efficiency in mind.

1.2 Contributions

Many performance issues in the context of data management for HPC simulations boil down to a problem of *performance variability*. Differences in the time to complete an I/O task from a process to another in a massively parallel application lead to all processes waiting for the slowest one. These processes thus waste valuable computation time and energy. With today's approaches to data management, this variability has multiple causes. First, in most

HPC applications I/O is concurrently performed by all processes, which leads to I/O bursts. This causes resource contention and substantial variability of the I/O performance of individual processes, which significantly impacts the overall application performance. Second, coupling simulations with visualization and analysis packages further increases this variability, especially when visualization tasks are performed interactively. A third source of variability comes from concurrent accesses to the storage system by many independent applications running on the same machine.

In this thesis, we aim to address the challenges posed by the increasing variability in the performance of current data management approaches. Our work was mainly carried out in the context of the JLPC (Joint Laboratory for Petascale Computing), a joint laboratory between Inria, the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (UIUC), and Argonne National Laboratory (ANL). It led to several projects between the KerData team, ANL and UIUC: FACCTS (France and Chicago Collaborating in the Sciences), PUF (Partner University Fund), and the Data@Exascale associated team. The main contributions of this Ph.D thesis can be summarized as follows.

Using Dedicated Cores in Multicore Nodes to Hide the I/O Jitter

With an increasing use of massively multicore nodes, a first level of contention occurs when many processes in the same node try to concurrently access the same network interface. This causes substantial performance variability. We call *I/O jitter* this type of variability, originating from I/O contention within a single application. In this work, we propose a new approach to I/O, called Damaris, which leverages dedicated I/O cores on each multicore SMP (Symmetric multiprocessing) node, along with the use of shared memory, to efficiently perform asynchronous data processing and I/O in order to hide this jitter. We evaluate Damaris on three different platforms including the Kraken Cray XT5 supercomputer [63], with the CM1 atmospheric model [7]. By overlapping I/O with computation and by gathering data into large files while avoiding synchronization between cores, our solution brings several benefits: (1) it fully hides the jitter as well as all I/O-related costs, which makes the simulation's performance predictable; (2) it increases the sustained write throughput by a factor of 15 compared to standard approaches; (3) it allows almost perfect scalability of the simulation up to over 9,000 cores, as opposed to state-of-the-art approaches which fail to scale; (4) it enables a 600% compression ratio without any additional overhead, leading to a major reduction of storage requirements. In addition to initiating the development of an implementation of Damaris, this work was awarded the 2^{nd} prize at the ACM Student Research Competition held in conjunction with the ICS '11 conference (see [23]). It also led to a publication at the CLUSTER '12 conference (see [24]).

Bringing In Situ Visualization Capabilities to Dedicated Cores

Reducing the amount of data stored by simulations will be of utmost importance for the next generation of large-scale computing. Accordingly, there is active research to shift analysis and visualization tasks to run in situ, that is, closer to the simulation by sharing its resources. This approach is beneficial as it can avoid the necessity to store large amounts of data for post-processing. However, it can lead to an important impact on the simulation's run time if not carefully implemented. This work focuses on the specific case of in

situ visualization where analysis codes are collocated with the simulation's code and run on the same resources. It is important for an in situ technique to require minimum modifications to existing codes, be adaptable, and have a low impact on both run times and resource usage. We accomplish this through the Damaris/Viz framework, which provides in situ visualization support to our implementation of the Damaris approach. The use of Damaris as a bridge to existing visualization packages allows us to (1) reduce code modification in existing simulations, (2) gather capabilities of several visualization tools to offer a unified data management interface, (3) use dedicated cores to hide the run-time impact of in situ visualization and (4) efficiently use memory through a shared-memory-based communication model. Experiments were conducted on Blue Waters and Grid'5000 [53] to visualize the CM1 atmospheric simulation and the Nek5000 CFD solver [98]. This work led to a publication at the LDAV '13 conference (see [26]).

Analyzing the Energy vs. Performance Tradeoff in Diverse I/O Approaches

A major challenge of future Exascale machines consists of sustaining a high performance per watt ratio. Many recent works, including the first contribution of this Ph.D. thesis, have explored new approaches to I/O management aiming to reduce the I/O performance bottleneck exhibited by HPC applications (and hence to improve application performance). There is comparatively little work investigating the impact of I/O management approaches on energy consumption. In particular, approaches that attempt to overlap computation with I/O have a beneficial effect on performance variability and thus, on energy consumption. In this work, we completed our implementation of the Damaris I/O middleware with various approaches to data management, including the possibility to use dedicated nodes instead of dedicated cores, and the possibility to run I/O tasks synchronously, i.e., with no dedicated resources at all. We closely examine these radically different I/O schemes and perform extensive experiments with the CM1 atmospheric model. Our experimental results obtained on the Grid'5000 platform highlights the differences between these approaches and illustrates in which way various configurations of the application and of the system impact performance and energy consumption. We then propose and validate a mathematical model to estimate the energy consumption of a simulation under different I/O approaches. Part of this work was published at the DIDC '14 workshop, held in conjunction with the HPDC '14 conference (see [140]).

Mitigating I/O Contention through Cross-Application Coordination

As larger machines are used by an increasing number of applications in a concurrent manner, the interference produced by multiple applications accessing a shared parallel file system in contention becomes a major problem. Interference often breaks single-application I/O optimizations (such as access patterns preliminarily optimized to improve data locality on disks), dramatically degrading application I/O performance, increasing run time variability and, as a result, lowering machine-wide efficiency. We addressed this challenge by proposing CALCioM, a framework that aims to mitigate I/O interference through the dynamic selection of appropriate scheduling policies. CALCioM allows several applications running on a supercomputer to communicate and coordinate their I/O strategy in order to avoid interfering with one another. In this work, we examine four I/O strategies that can

be accommodated in this framework: serializing, interrupting, interfering and coordinating. Experiments on Argonne's BG/P Surveyor machine and on several clusters of Grid'5000 show how CALCioM can be used to efficiently and transparently improve the scheduling strategy between two otherwise interfering applications, given specified metrics of machine wide efficiency. This work, partially carried out during a 3-month internship at ANL, led to a publication at the IPDPS '14 conference (see [27]).

Predicting the Spatial and Temporal I/O Patterns of HPC Applications

Many I/O optimizations including prefetching, caching, and scheduling, have been proposed to improve the performance of the I/O stack. In order to optimize these techniques, modeling and predicting spatial and temporal I/O patterns of HPC applications as they run have become crucial. In this direction we introduce Omnisc'IO, an original approach that aims to make a step forward toward an intelligent I/O management of HPC applications in next-generation post-Petascale supercomputers. It builds a grammar-based model of the I/O behavior of any HPC application and uses this model to predict when future I/O operations will occur, as well as where and how much data will be accessed. Omnisc'IO is transparently integrated into the POSIX and MPI-I/O stacks and does not require any modification to application sources or to high-level I/O libraries. It works without prior knowledge of the application, and converges to accurate predictions within a couple of iterations only. Its implementation is efficient both in computation time and in memory footprint. Omnisc'IO was evaluated with four real HPC applications – CM1, Nek5000, GTC [43], and LAMMPS [105] – using a variety of I/O backends ranging from simple POSIX to Parallel HDF5 on top of MPI-I/O. Our experiments show that Omnisc'IO achieves from 79.5% to 100% accuracy in spatial prediction and an average precision of temporal predictions ranging from 0.2 seconds to less than a millisecond. This work was published at the SC '14 conference and initiated the development of the Omnisc'IO software (see [28]).

1.3 Publications

International Conferences

- Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé, **Matthieu Dorier**. *BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map/Reduce Applications*, in Proceeding of the 2010 IEEE International Parallel & Distributed Processing Symposium **(IPDPS '10)**, *Atlanta, September 2010*. CORE Rank A (acceptance rate 24%).

- **Matthieu Dorier**, Gabriel Antoniu, Franck Cappello, Marc Snir, Leigh Orf. *Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O*, in Proceedings of the 2012 IEEE International Conference on Cluster Computing **(CLUSTER '12)**, *Beijing, September 2012.* CORE Rank A (acceptance rate 28%).

- **Matthieu Dorier**, Roberto Sisneros, Tom Peterka, Gabriel Antoniu, Dave Semeraro. *Damaris/Viz, a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework*, in Proceedings of the 2013 IEEE Symposium on Large Data Analysis and Visualization **(LDAV '13)**, *Atlanta, October 2013.* (acceptance rate 37%).

- **Matthieu Dorier**, Gabriel Antoniu, Rob Ross, Dries Kimpe, Shadi Ibrahim. *CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination*, in Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium **(IPDPS '14)**, *Phoenix, May 2014.* CORE Rank A (acceptance rate 21%).

- **Matthieu Dorier**, Shadi Ibrahim, Gabriel Antoniu, Rob Ross. *Omnisc'IO: A Grammar-Based Approach to Spatial and Temporal I/O Patterns Prediction*, in Proceedings of ACM/IEEE 2014 Supercomputing Conference **(SC '14)**, *New Orleans, November 2014.* CORE Rank A (acceptance rate 21%).

Workshops at International Conferences

- Orçun Yildiz, **Matthieu Dorier**, Shadi Ibrahim, Gabriel Antoniu. *A Performance and Energy Analysis of I/O Management Approaches for Exascale Systems*, in Proceedings of the 2014 Data-Intensive Distributed Computing **(DIDC '14)** workshop, held in conjunction with the 23^{rd} International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC '14), *Vancouver, June 2014.*

Posters at International Conferences

- **Matthieu Dorier**. *Damaris - Using Dedicated I/O Cores for Scalable Post-petascale HPC Simulations*, 2011 ACM/SIGARCH International Conference on Supercomputing **(ICS '11)**, *Tucson, April 2011.* 2^{nd} **prize at the ACM Student Research Competition.**

- **Matthieu Dorier**. *Efficient I/O using Dedicated Cores in Large-Scale HPC Simulations*, 2013 IEEE International Parallel & Distributed Processing Symposium **(IPDPS '13)**: PhD Forum, *Boston, May 2013.*

1.4 Software

- **Damaris** is a middleware for multicore SMP nodes allowing them to efficiently handle data transfers for storage and visualization by dedicating one or a few cores to the application I/O. It allows efficient asynchronous I/O, hiding all I/O related overheads such as data compression, post-processing and in situ visualization (through its Damaris/Viz extension). Damaris was evaluated on Blue Waters (Cray XE6, NCSA), Kraken (Cray XT5, NICS), Titan (Cray XK7, ORNL), Intrepid (IBM BlueGene/P, ANL), Grid'5000 (French grid testbed), Blue Print (Power5 cluster, NCSA), with the CM1 atmospheric simulation and the Nek5000 CFD simulation. Damaris is at the core of Chapters 3 to 5 of this thesis. Damaris was formally validated for use on NCSA's Blue Waters supercomputer. To our knowledge, it has been used successfully by several researchers from NCSA, Central Michigan University and the Federal University of Rio Grande do Sul (UFRGS).
  Link: http://damaris.gforge.inria.fr
  Size and language(s): 19500 lines, C++, Fortran, XML
  License: LGPL

- **Omnisc'IO** is a middleware integrated in the POSIX and MPI-I/O stacks to transparently observe, model and predict the I/O behavior of any HPC application. It is based on formal grammars and implements a modified version of the Sequitur algorithm. Omnisc'IO has been used on Grid'5000 with the CM1 atmospheric simulation, the LAMMPS molecular dynamics simulation, the GTC fusion simulation and the Nek5000 CFD simulation. Omnisc'IO is at the core of Chapter 7 of this thesis.
  Link: http://omniscio.gforge.inria.fr
  Size and language(s): 4400 lines, C++
  License: LGPL

- **Darshan-Ruby** is an object-oriented extension to simplify the analysis of ANL's Darshan [10] log files (a tool that traces the I/O of simulations running on supercomputers) using the Ruby language. It was developed to help us get a faster insight into the I/O behavior of large-scale applications. Darshan-Ruby efficiently accesses Darshan data without intermediate conversion into text format. It is available as a Ruby Gem package on the official rubygems.org repository and, as of September 2014, was downloaded 1600 times. Darshan-Ruby led to the Darshan-Web project, which proposes a web platform that analyzes traces and provides users with hints on how to improve the I/O performance of their applications.
  Link: http://darshan-ruby.gforge.inria.fr
  Size and language(s): 400 lines, C, Ruby
  License: LGPL

1.5 Organization of the Manuscript

The rest of this manuscript is organized in seven chapters.

The first chapter presents the context of our research. We introduce the applications and platforms, as well as the traditional approaches to data management in post-Petascale systems. We then dive into the challenges posed by these approaches in terms of I/O variability at large scale and highlight the opportunities that drove our contributions.

Chapters 3 to 5 focus on contributions related to the Damaris approach. Three major objectives are being tackled: (1) hiding the I/O costs and variability from HPC applications, (2) offering a non-impacting and adaptable way of conducting in situ analysis and visualization of large-scale simulations and (3) understanding the effect of I/O on energy consumption under different I/O approaches. Chapter 3 addresses the first objective. It presents the core of our Damaris approach and its results in addressing the challenge of hiding the I/O variability while improving I/O performance. Based on the observation that dedicated cores in Damaris remain idle an important fraction of the time, we added the support for in situ visualization. Our results on leveraging Damaris to provide in situ visualization capabilities to simulations are presented in Chapter 4. Finally, in order to compare different approaches to I/O in terms of energy consumption, we added the support for dedicated I/O nodes in Damaris and conducted an extensive evaluation of the energy/performance tradeoff. This contribution is described in Chapter 5. While each chapter focuses on a particular challenge related to data management, the use of Damaris as a common framework for all three chapters allows us to zoom on particular implementation details relevant for each of them.

Chapter 6 takes a step back from optimizing a single application. It studies the effect of I/O contention between different applications at the level of the file system, on the efficiency of the supercomputer. It proposes CALCioM, an approach based on cross-application coordination that aims to mitigate I/O interference. Based on the observation that CALCioM, as well as many optimizations to I/O such as scheduling, caching or prefetching, require a model of the I/O behavior of applications, we introduce the Omnisc'IO approach in Chapter 7. Its goal is to transparently capture, model, then predict the spatial and temporal I/O access patterns of HPC applications.

Chapter 8 concludes our thesis by summarizing our contributions and presenting perspectives.

Chapter 2

Background: I/O and Data Analysis in Supercomputers

Contents

THIS chapter aims to draw a picture of scientific simulations as well as the architecture of current "post-Petascale" infrastructures on which they run. We then dive into the details of data management approaches on these machines, including data storage, analysis and visualization. Finally, we show the challenges they pose.

2.1 The Era of Supercomputing

2.1.1 Large-Scale Scientific Simulations

Numerical simulations are parallel programs that solve a system of mathematical equations describing a physical phenomenon. These simulations usually distribute a large dataset across a number of processes that solve the equations on their parts of the data. Each chunk of data is called a *subdomain* (e.g., a subset of particles in nuclear simulations, or a chunk of atmosphere in climate simulations). While the speed of these simulations depends on the number of processes on which they run, this number of processes also impact the accuracy at which the physical phenomena is simulated.

For example, the CM1 simulation [7, 8], used later in this work, models relatively small-scale processes in the Earth's atmosphere, such as thunderstorms and tornadoes. It discretizes a cube of atmosphere that is distributed across processes following a two-dimensional grid.

Many simulations work in an iterative manner, alternating between computation-intensive phases that solve the equations, and I/O phases that output data representing the current state of the simulated model. These outputs are used for later analysis and visualization purpose, or as a mean to restart the simulation in case of failure.

To achieve high accuracy in a reasonable amount of time, scientific simulations require important computation resources as well as large amounts of memory. Therefore they are parallelized to run on high-performance machines such as supercomputers.

2.1.2 Post-Petascale Supercomputers

Supercomputers are highly parallel machines developed to push the frontiers of computation performance. As opposed to other distributed environments such as grids, which consist of heterogeneous commodity hardware, or clouds, which employ virtualization, supercomputers are clusters of homogeneous cores using cutting-edge hardware, along with libraries and software that are specifically optimized for this hardware.

Post-Petascale supercomputers nowadays have a more and more hierarchical architecture: the traditional cluster of single-CPU nodes interconnected through a network has given way to cabinets of blades of massively multicore nodes. Within a node, these cores share common caches and main memory, while communication across nodes is made available through specialized high-performance interconnects such as InfiniBand or Cray HSN.

The hardware also diversifies, with a popularization of GPGPUs (General Purpose Graphics Processing Unit, as in the Cray XK7 architectures used by NCSA's Blue Waters [5] and ORNL's Titan [129] supercomputers), local storage such as SSDs attached to computation nodes and replacing or complementing DRAM (as in the Cray CS300 Catalyst super-

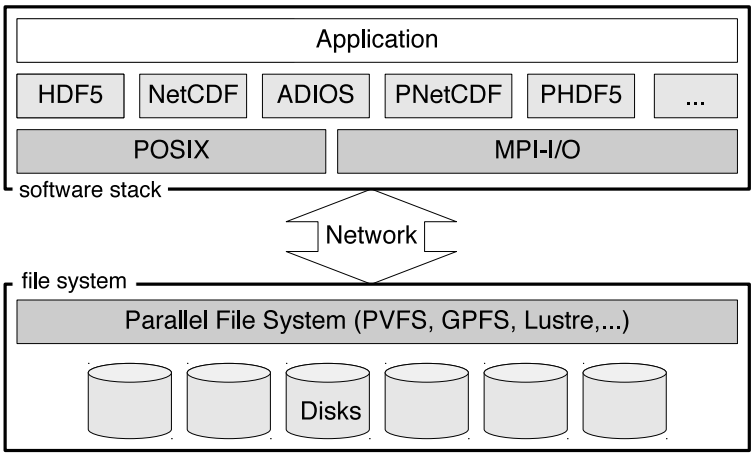


Figure 2.1: The typical I/O Stack of HPC Simulations.

computer, which will be delivered at LLNL [58]), or cores dedicated to the operating system (as in the BlueGene/Q architecture used by the ANL's Mira supercomputer [83]).

On such a machine, users can reserve a number of cores for a given duration. Each process of a large-scale simulation runs on a core of the supercomputer. The MPI (Message Passing Interface) standard [86] was introduced to offer a portable communication interface between cores, providing point-to-point communication primitives as well as complex collective communication algorithms. On recent machines with multicore nodes, this interface is sometimes used for inter-node communications only, while parallelism within a node is handle by a multithreading library such as OpenMP [95].

One important part of these high-performance computing machines in our context is the storage system, which typically consists of a clustered NAS (Network-Attached Storage) on which a parallel file system is deployed. The next section describes the I/O stack from this storage area up to the application level.

2.2 I/O and Storage in HPC Systems

The I/O stack of HPC applications typically consists of three layers, plus the application itself, as shown in Figure 2.1. Data output by large scale simulations are commonly stored in a *parallel file system* (PFS) and formatted using a high-level *I/O library* for portability across platforms and software. These libraries interact with the file system either through the POSIX or MPI-I/O interfaces. The following sections examine these layers in more detail.

2.2.1 Parallel File Systems

The PFS runs on a set of I/O servers in a storage area separated from the computation nodes by a network. These servers provide an interface between computation nodes and a large number of disks. For example the Blue Waters supercomputer features 25 petabytes of storage capacity from a set of 1440 disks managed by a Lustre [22] file system. PFSs are responsible for distributing data across disks and for exposing a consistent namespace with a

hierarchy of directories and files to the user in a way similar to a simple workstation, offering the illusion of a single-disk file system with very large capacity.

Data accesses in PFSs can be parallelized by distributing requests across I/O servers. To handle metadata operations (i.e., opening or creating a file, listing a directory retrieving file permissions, etc.), PFSs such as Lustre [22] features a metadata server. This single metadata server can become a bottleneck when an application accesses hundreds of thousands of files at the same time. In other PFSs such as GPFS [114] and PVFS2 [11], I/O servers also act as metadata servers and, thus, metadata operations can be distributed as well, avoiding this bottleneck.

The POSIX Consistency Semantics

Most parallel file systems expose a POSIX consistency semantics for concurrent accesses: a process reading a file will see either all or none of the effects of a `write` operation performed by another concurrent process (atomicity of `write` operations). This consistency semantics is borrowed from traditional file systems. Yet it is the source of performance issues in the context of parallel file systems due to the distribution of data across servers. In such a configuration, a single write can indeed result in a series of requests to multiple servers that have to be serviced in a consistent manner even under concurrency.

The obvious approach to ensure the POSIX consistency semantics consists of locking an entire file through the metadata server when the file is accessed. However, more elaborate methods can be employed. GPFS enforces this consistency through a distributed byte-range locking protocol [114], which issues access authorizations to individual processes only for the region of the file that is accessed. Even though this algorithm allows better performance than a centralized lock on the whole file, it adds an overhead in small accesses. Other file systems such as PVFS2 openly don't provide this consistency semantics to avoid this overhead.[1]

2.2.2 The MPI-I/O and POSIX Interfaces

Accesses to the file system can be done simply through the standard POSIX system calls,[2] i.e., `open`, `close`, `read`, `write`, etc., as well as their LibC versions `fopen`, `fclose`, `fread`, `fwrite`, etc. and equivalent functions in C++ and Fortran. These functions, provided essentially for compatibility with commodity machines, allow one process to access a file but do not handle concurrent accesses by multiple processes to the same file. They are thus restricted to a file-per-process approach.

Since MPI 2, MPI-I/O is provided as part of the MPI standard [39] to provide a portable I/O interface on top of the diversity of parallel file systems. It provides a generic interface for opening, reading and writing files stored in parallel file systems, such as `MPI_File_open`, `MPI_File_write`, etc. as well as functions enabling collective accesses to a single file by multiple processes.

[1] See http://www.pvfs.org/cvs/pvfs-2-7-branch.build/doc/pvfs2-guide/pvfs2-guide.php.

[2] Not to be confused with the notion of POSIX semantics defined earlier.

Optimizations in MPI-I/O

MPI-I/O provides a number of optimizations [125]. Most of them aim to reduce the number of requests issued to the file system, or the number of processes from which these requests are issued.

Data sieving [18] is a common technique to overcome the inefficiency of reading many non-contiguous regions of a file. Instead of reading one by one each requested segment of the file, the process issues a large, contiguous request covering all requested segments. The unnecessary data is then discarded.

Collective buffering [19] takes place when a set of processes collectively accesses a shared file in a strided pattern, that is, each process issues a series of requests to interleaving segments of the file. In this scenario, the processes involved in the collective operation merge their requests into a subset of processes that issues bigger and more contiguous requests. This algorithm is sometimes termed "two-phase I/O".

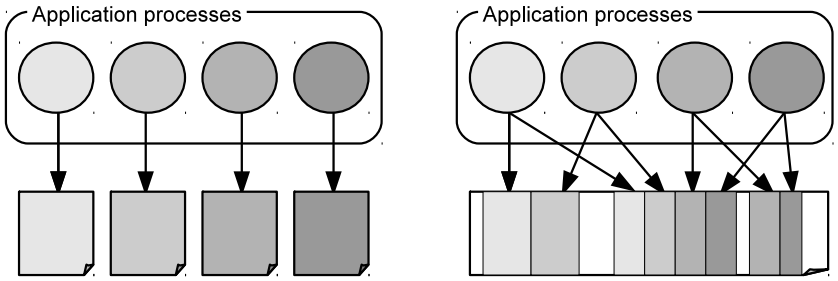
2.2.3 High-Level I/O Libraries

The need for scientists to be able to read output data on different platforms and with various general-purpose analysis programs has led to the development of high-level I/O libraries such as HDF5 [50, 35], NetCDF [90] or ADIOS [2, 1]. They use metadata-rich formats that let users provide a descriptions of their variables, together with the type of the data (e.g., float, integer, bytes, arrays, etc.) or its layout (e.g., number of dimensions, endianness for numerical values, etc.). These formats present datasets in a hierarchical manner, grouping variables in a way similar to the namespace of a file system. A file written in such a format is machine-independent, language-independent and easy to read by other applications, in particular by post-processing tools. In some ways, these formats represent to scientific datasets the equivalent of the PNG or JPG formats for images.

These high-level libraries have shown to provide highly-featured formatting capabilities without much overhead in I/O operations [16] compared to raw MPI-I/O or POSIX outputs. They also provide compression capabilities such as scale-offset (reduction of floating point values from 32 to 16 bits representation), Szip, Gzip or LZF.

While HDF5 and NetCDF use the POSIX interface for independent I/O accesses, parallel versions of these libraries, namely PHDF5 [16] and Parallel-NetCDF [71], support collective I/O through MPI-I/O. ADIOS provides its own data format but can also be configured to output in HDF5 or NetCDF formats. As a result, ADIOS can be seen as an extra software layer between the simulation and I/O libraries allowing more flexible I/O [76].

Contrary to other I/O libraries, ADIOS leverages an XML configuration file that describes the data being read and written by the simulation. This configuration allows a more flexible interface, as the user does not need to recompile his application to change its I/O methods and output format. Besides, the a priori knowledge of the data being accessed helps ADIOS efficiently manage its buffer.



(a) File-per-process Approach (b) Collective I/O Approach

Figure 2.2: Traditional approaches to I/O in HPC simulations.

2.2.4 Application-Level I/O Approaches

At the application level finally, two I/O approaches are commonly employed. These approaches are summarized in Figure 2.2.

The File-per-process approach consists of having each process access its own file. This is usually done through the POSIX interface, which does not handle concurrency and is thus appropriate when the simulation ensures that different processes will not access the same file.

Collective I/O leverages communication phases between processes to aggregate access requests and re-organize them. These operations are typically used when several processes need to access different parts of a shared file, and benefit from tight interactions between the file system and the MPI-I/O layer in order to optimize the application's access pattern [106], as explained in Section 2.2.2.

While the *file-per-process* approach avoids synchronization between processes, parallel file systems are not well suited for this type of load when scaling to hundreds of thousands of processes. In particular, file systems that feature a single metadata server suffer from an important bottleneck when hundreds of thousands of files are created simultaneously.

Collective I/O helps alleviating the problem of metadata overhead by gathering data into very large files, but this comes at the price of an important synchronization overhead between processes.

2.3 Variability in Traditional I/O Approaches

The periodic nature of scientific simulations, which alternate between computation and I/O phases, leads to I/O bursts. With larger machines, the higher degree of I/O concurrency between processes of a single application or between concurrent applications pushes the I/O subsystem to its limits. This leads to a substantial variability in the I/O performance. Reducing this variability is critical, as it is an effective way to make a more efficient use of these new computing platforms through improved predictability of the behavior and of the execution time of applications.

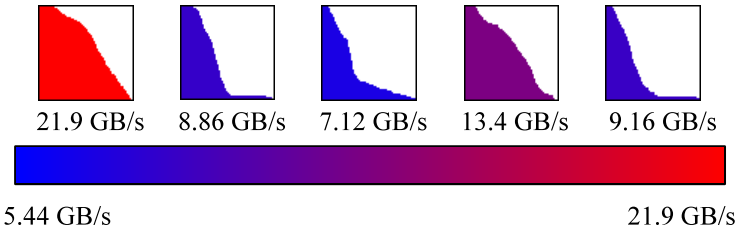


Figure 2.3: Variability between processes and across I/O phases in the IOR benchmark using a file-per-process approach, on Grid'5000 with a PVFS file system. Each graph represents a write phase. Processes are sorted by write time on the y axis and a line is draw with a length that is proportional to this write time. These graphs are normalized so that the longest write time spawns the entire graph. Each graph is colored according to a color scale that gives the aggregate throughput of the phase, that is, the total amount of data written divided by the write time of the slowest process.[4]

Figure 2.3 illustrates this variability with the IOR applications [116], a typical benchmark used to evaluate the performance of parallel file systems with ideal I/O patterns. It shows that even with very well optimized I/O (each process here writes the same amount of data contiguously in a separate file) there is a large difference in the time taken by each process to complete its I/O operations within a single I/O phase and also across I/O phases. Since during these I/O phases all processes have to wait for the slowest one, this I/O variability is the source of a wast of performance, a wast of energy, and unpredictable overall run times. I/O variability is therefore a key issue that we aim to address in this Ph.D. thesis.

2.3.1 Causes and Effects of the I/O Variability

Skinner at al. [117] point out four causes of performance variability in supercomputers (here presented in a different order):

1. Communication, causing synchronization between processes that run within the same node or on separate nodes. In particular, network access contention causes collective algorithms to suffer from variability in point-to-point communications.
2. Kernel process scheduling, together with the jitter introduced by the operating system.
3. Resource contention within multicore nodes, caused by several cores accessing shared caches, main memory and network devices.
4. Cross-application contention, which constitutes a random variability coming from simultaneous accesses to shared components of the computing platform, such as the network or the storage system, by distinct applications.

Issues 1 and 2 respectively cause communication and computation jitter. Issue 1 can be addressed through more efficient network hardware and collective communication algorithms. The use of lightweight kernels with less support for process scheduling can alleviate issue 2. Issues 3 and 4, on the other hand, cause I/O performance variability.

[4]Due to the use of colors, this figure may not be properly interpretable if this document was printed in black and white. Please refer to an electronic version.

At the level of a node, the increasing number of cores per node in recent machines makes it difficult for all cores to access the network all at once with an optimal throughput. Requests are serialized in network devices, leading to a different service time for each core. This problem is further amplified by the fact that an I/O phase consists of many requests that are thus serialized in an unpredictable manner.

Parallel file systems also represent a well-known bottleneck and a source of high variability [134]. The time taken by a process to write some data can vary by several orders of magnitude from one process to another and from one I/O phase to another depending on many factors, including (1) network contention when several nodes send requests to the same I/O server [27], (2) access contention at the level of the file system's metadata server(s) when many files are created simultaneously [25], (3) unpredictable parallelization of I/O requests across I/O servers due to different I/O patterns [75], (4) additional disk-head movements due to the interleaving of requests coming from different processes or applications [36].

Lofstead et al. [75] present I/O variability in terms of *interference*, with the distinction between *internal interference* caused by access contention between processes of the same application, and *external interference* that are due to sharing the access to the file system with other applications, possibly running on different clusters. While the sources of I/O performance variability are numerous and difficult to track, we can indeed observe that some of them originate from contentions within a single application, while other come from the contention between multiple applications concurrently running on the same platform. The following section describes how these two sources of contention can be tackled.

2.3.2 Approaches to Mitigate the I/O Variability

While most efforts today address performance and scalability issues for specific types of workloads and software or hardware components, few efforts target the causes of performance variability. We highlight two practical ways of hiding or mitigating the I/O variability, and we show their limitations.

Approach 1: Asynchronous I/O

The main solution to prevent an application from being impacted by its I/O consists of using asynchronous I/O operations, i.e., non-blocking operations that proceed in the background of the computation.

The MPI 2 standard proposes rudimentary asynchronous I/O functions that aim to overlap computation with I/O. Yet these functions are available only for independent I/O operations. Besides, popular implementations of the MPI-I/O standard such as ROMIO [126] actually implement most of these functions as synchronous. Only the small set of functions that handle contiguous accesses have been made asynchronous, provided that the backend file system supports it.

Released in 2012, the MPI 3 standard completes this interface with asynchronous collective I/O primitives. Again, their actual implementation is mostly synchronous. As of today, there is no way to leverage completely asynchronous I/O using only MPI-I/O. Higher-level libraries such as HDF5 or NetCDF have also no support for asynchronous I/O.

The Damaris approach that we introduce in Chapter 3 precisely aims to alleviate the lack of support for asynchronous I/O by using dedicated cores on multicore nodes. The use of dedicated cores mitigates the contention for the access to the network and the file system, as only one of the cores in each multicore node actually accesses the network. As a result, Damaris is shown to be an effective way to completely hide the I/O variability.

Approach 2: I/O Scheduling

I/O variability can be mitigated by better addressing multi-application contention. In parallel file systems, this contention is resolved through the use of schedulers. At a network level, individual schedulers from different I/O servers will try to service requests from the same application at the same time, in order for all processes of the application to experience the same throughput [147]. At the disk level the requests can be reordered and cached in order to optimize disk accesses.

While most parallel file systems use a scheduler that treats requests in a first-come-first-served manner, more elaborate systems have been proposed that use deadlines [109, 120], or quality of service (QoS) requirements [148]. Other [147, 69] attempt to improve data locality (i.e. minimize disk-head movements). These schedulers usually target either better performance, better QoS, or better fairness. They all work at the level of the parallel file system, and thus impose an overhead on the communications between clients and I/O servers.

We propose a radically different approach in Chapter 6 with CALCioM. Instead of a PFS-level scheduler, CALCioM provides a communication layer across concurrent applications. This communication layer can be used by applications to coordinate there I/O behavior and avoid interfering with one another. CALCioM thus appears as an effective way to deal with performance variability induced by inter-application I/O contention.

2.3.3 I/O Variability: Energy Concerns

The Cost of Energy in HPC

Power has become an essential issue in the design of modern computing systems. Power bills become a substantial part of the total cost of ownership (TCO) of supercomputers: a typical supercomputer of thousands of cores consumes several MegaWatts of power [102], which in turn represents almost 40% of the total cost [44]. Performance has long been the major focus of the HPC community, today's supercomputers are therefore equipped with millions of processing cores that consume a large amount of energy when running parallel programs. For example, Tianhe-2, ranked first in the Top 500 supercomputers list in November 2013 [130], is a 3,120,000 processor supercomputer with a Linpack performance of 33.8 Petaflops, but with 17 MegaWatt of power consumption [42]. This amount of energy will even increase as we reach the era of Exascale systems.

Solutions to consuming less energy include building supercomputers with slower cores. The power consumption of a core is indeed proportional to its frequency. The frequency and voltage of a core can also be adapted dynamically using DVFS (Dynamic Voltage and Frequency Scaling) [68] depending on the computation load.

New hardware tend to put components (memory, arithmetical units, etc.) closer to one another in order to reduce energy dissipation in wires. The use of optical interconnects [104]

also reduces the energy consumption.

Energy and I/O Variability

While data movements themselves consume a lot of energy, the cause of energy inefficiency in I/O approaches can be much more diverse.

In particular, the variability in the duration of I/O tasks performed in parallel by many cores forces some cores to remain idle while waiting for other cores to complete their I/O. During this time, idle cores remain powered on and consume unnecessary energy. Using a data management approach that reduces I/O variability can thus be beneficial in terms of energy consumption as well.

Several approaches have been proposed to overlap I/O and computation in order to hide the I/O variability and make a better use of the platform. Considering the rising concerns on energy consumption in the HPC community, we have tackled in Chapter 5 the important challenge of determining the impact of some of these I/O approaches on both performance and energy consumption. Additionally, we provide a theoretical model to help users chose the best I/O approach with respect to energy consumption.

Parallel file systems also consume a lot of energy. Part of the energy consumed by supercomputers is spent keeping hundreds of thousands of disks spinning [38]. Many factors impact the energy consumption of the parallel file system [61], such as disk-head movements induced by unoptimized scheduling of I/O requests. While Solid State Disks (SSDs) are more and more common on compute nodes of recent supercomputers and have a lower energy footprint than hard disks (HDDs) as they do not feature mechanical components, they are still not economically viable as a replacement of HDDs in parallel file system due to their higher cost.

2.3.4 Variable I/O, Yet Predictable of I/O Patterns

While the performance of I/O in HPC systems is subject to a high variability, the periodic and repetitive behavior of large-scale simulations makes most of their I/O patterns (i.e., which file is accessed, when and how) predictable. This section explores how this predictability can be used to solve I/O performance issues, and what are the challenges of predicting I/O patterns.

A Silver Lining for a "Smarter" I/O Stack

As non-interactive programs periodically solving a system of equations, most HPC applications actually exhibit a spatially and temporally predictable I/O behavior. That is, *a process is likely to repeat the same access patterns across the various files it produces, and to do so at predictable moments*.

This predictability is an opportunity to address many of the challenges posed by post-Petascale I/O; it can lead to the development of a "smarter" I/O stack, that is, an I/O stack capable of *modeling*, *understanding* and *adapting* to the behavior of the applications. We exemplify here three common techniques that can benefit from a prediction of I/O behavior.

Leveraging Predictions for Buffering and Caching: When writing large amounts of data in a series of small accesses, buffering is a useful way to aggregate requests and improve performance. Buffering systems can benefit from a prediction of the I/O patterns by estimating *when* the buffer should be flushed to the file system (i.e., preferably when enough requests have been aggregated), as well as *how much* memory should be allocated for the buffer to work efficiently. When the parallel file system leverages caching mechanisms, understanding the behavior of running applications helps appropriately provisioning separate caches for each application instead of letting them compete for the access to the same cache.

Leveraging Predictions for Prefetching: Prefetching is a common technique for improving read performance. It consists of reading some data that are likely to be requested later by the application ahead of time. This proactive approach self-advocates for I/O predictions, as knowing in advance *what* will be read (and sometimes *when*) can help improving the efficiency of the prefetching system. While the most common strategy consists of prefetching segments of data that immediately follow the segment previously read in the file [139], this strategy is limited when the simulation uses a high-level library, since these libraries move the file pointer in a noncontiguous manner to update headers and metadata sections in the file. Better strategies to predict the future access patterns are thus required to improve prefetching.

Leveraging Predictions for I/O Scheduling: I/O scheduling is implemented in parallel file systems to properly service multiple applications at the same time. The knowledge of the number of requests that an application will send in a near future as well as the location of corresponding accesses in files can help achieving better data locality in disks, better fairness across applications, better quality of service, and better performance overall. Currently, some I/O schedulers [147] assume that an application currently accessing the file system is likely to re-access it immediately, and use time windows (or "reuse distances") allocated to applications. Yet this assumption that replaces an actual prediction of request inter-arrival times is limited. The scheduler is indeed unable to predict the end nor the duration of an I/O phase, nor what will be accessed and how. The time windows can also be too short to offer a chance to an application with slightly large inter-arrival times to benefit from them.

I/O Predictions: a Few Challenges

While predicting the I/O patterns of HPC applications has long been an important goal in large-scale supercomputers, researchers have focused mainly on statistical methods (e.g., hidden Markov models [94], ARIMA models [131]) to help with I/O predictions. These approaches often focus exclusively on either spatial or temporal I/O behaviors. Furthermore, they require a large number of observations to accomplish good prediction; hence, they either need long execution times (several runs in some cases) [9] or they are doomed to offline trace-based training [80] in order to converge.

We list five main requirements for the design of a good I/O-prediction system hereafter:

- **Run time learning:** The system should learn a model of the application's behavior *at run time* with no or little prior knowledge of the application. It should thus converge fast in order to rapidly make correct predictions.

- **Spatial and temporal predictions:** The system should be capable of both spatial and temporal predictions, that is, it should be capable of predicting *where* in the files the next operations will occur, *how much* data will be accessed, and *when* these operations will be performed.

- **Low run time overhead:** The system should not negatively impact the performance of the application, nor make this performance more variable.

- **Low memory footprint:** The system should be memory-efficient.

- **Transparent integration:** The integration of the prediction system should not require any modification in the application nor any high-level I/O library that the application may use. Besides, it should not depend on these libraries.

To address these challenges, we introduce the Omnisc'IO approach in Chapter 7. Omnisc'IO leverages grammar-based models to predict the I/O behavior of any HPC application at run time. It specifically addresses all the requirements presented here by transparently building a model as the simulation runs, with minimal run-time and memory overhead. It then uses this model to predict the location of future accesses (size and offset in file) and the date of these accesses.

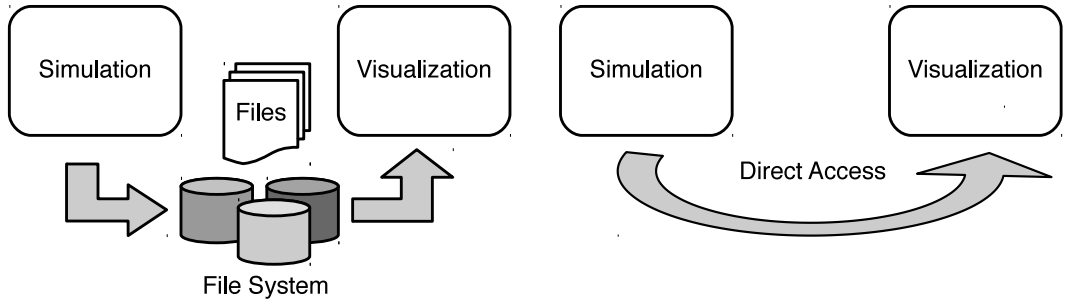
2.4 Analysis and Visualization: an Overlooked Process

Data produced by HPC simulations can serve several purposes. One of them is fault tolerance using a checkpoint/restart method. The other, and most important, is the analysis and visualization of the simulated phenomenon.

Analysis and visualization are important components of the process that leads from running a simulation to actually *discovering knowledge*. Yet scientists tend to leave aside optimizations of the visualization process to focus on the performance of their simulation only. This section describes common techniques and software used for analysis and visualization, as well as the trend toward simulation/visualization coupling.

2.4.1 Visualization Software and Techniques

Analysis and visualization are traditionally carried out in an offline manner. Data produced by a simulation is read back from the file system by visualization software after the simulation has completed. This process is illustrated in Figure 2.4 (a). The visualization and analysis processes are often performed on a dedicated cluster, different from the cluster used by the simulation. For instance, while NICS provides the 1.17 Petaflops Cray XT5 Kraken machine [63] to run simulations, the 8.2 Teraflops SGI Altix UV Nautilus machines [89] is used for analysis and visualization tasks. This machine features GPUs (Graphics Processing Unit) and a set of software specialized for analysis and visualization. Yet due to the increasing amount of data, it becomes more and more common to perform analysis and visualization on the machine that produced the data, rather than having a secondary cluster for this purpose. This further motivates the adoption of GPUs in recent supercomputers.



(a) Traditional Scientific Workflow (b) Coupling Simulation/Visualization

Figure 2.4: Two approaches to retrieve insight from large-scale simulations.

Several general purpose tools such as VisIt [135] or ParaView [101] propose a wide range of visualization algorithms (e.g., iso-surface, streamlines, etc.) to deal with the variety of scientific datasets. Some post-processing tools are designed for specific fields as well. YT [133] is an example of Python library specifically targeting data produced by several cosmology codes.

All these software run in parallel and are able to read files written in common scientific formats such as the ones described in Section 2.2.3. As a rule of thumb, as much as 10% of the number of cores needed to produce the data is used to perform visualization [17].

2.4.2 Toward Simulation/Visualization Coupling

Given the increasing computation power and the trend toward leveraging GPUs on computation nodes, it also becomes more and more common to couple the simulation with the analysis and visualization tools. Simulation/Visualization coupling consists of making the simulation send its data directly to a visualization software instead of storing it and processing it offline. This approach, termed *in situ visualization* (ISV) and illustrated in Figure 2.4 (b), has the advantage of bypassing the storage system and produces results faster. It also allows scientists to control their simulations as they run, efficiently overlapping simulation and knowledge discovery.

2.4.3 A Taxonomy of In Situ Visualization Methods

Several ISV strategies exist that we separate in two main categories – tightly coupled and loosely coupled – depending on the location of the visualization tasks.

Tightly-Coupled In Situ Visualization

In a tightly-coupled scenario, the analysis and visualization codes run on the same node than the simulation and share its resources. The main advantage of this scenario is the proximity to the data, which can be retrieved directly from the memory of the simulation. Its drawback lies in the impact that such analysis and visualization tasks can have on the performance

of the simulation and on the variability of its run time. Within this category, we make a distinction between *time-partitioning* and *space-partitioning*.

Time-partitioning visualization consists of periodically stopping the simulation to perform visualization tasks. This method is the most commonly used. For example, it is implemented in VisIt's *libsim* library [138] and ParaView's *co-processing* library [34].

In a space-partitioning mode, dedicated cores are used to perform visualization in parallel with the simulation. This mode poses challenges in efficiently sharing data between the cores running the simulation and the cores running the visualization tasks, as these tasks progress in parallel. It also imposes to reduce the number of cores used by the simulation.

Loosely-Coupled In Situ Visualization

In a loosely coupled scenario, analysis and visualization codes run on a separate set of resources, that is, a separate set of nodes located either in the same supercomputer that runs the simulation [149, 110], or in a remote cluster [81]. The data is sent from the simulation to the visualization nodes through the network.

Some ISV frameworks such as GLEAN [49] can be viewed as hybrid, placing some tasks close to the simulation in a time-partitioning manner while other tasks run on dedicated nodes.

2.4.4 From Offline to In Situ Visualization: Another Source of Variability

The increasing amounts of data generated by scientific simulations also leads to performance degradations when it comes to reading back data for analysis and visualization [17, 141]. While I/O introduces run time variability, in situ analysis and visualization can also negatively impact the performance of the simulation/visualization compound. For instance, periodically stopping the simulation to perform in situ visualization in a time-partitioning manner leads to a loss of performance and an increase of run-time variability. Contrary to the performance of the simulation itself, the performance of visualization tasks highly depends on the content of the data and is therefore unbalanced across processes and across iterations. This variability is further amplified if the ISV framework is interactive, in which case the user himself impacts the performance of his application.

In a loosely-coupled approach to ISV, sending data through the network potentially impacts the performance of the simulation and forces a reduced number of nodes to sustain the input of a large amount of data. Transferring such large amounts of data through the network also have a potentially larger impact on the simulation than running visualization tasks in a tightly-coupled manner.

2.4.5 Our Vision: Pushing (Harder) Toward In Situ Visualization

Despite the limitations of the traditional offline approach, the fact that scientists are seldom accepting in situ visualization is a recurrent ascertainment [142, 78, 77]. Decades of traditions in offline analysis strengthened the idea that computer scientists will always find new solutions to the storage challenges and let users produce massive amounts of data. We postulate that four main requirements drive the adoption of an ISV framework.

Low impact on run time: As explained earlier, using computational resources collocated with the simulation can affect the performance of the underlying simulation. This is especially true when interactive visualization systems directly connect users to their running simulation.

Optimized resource utilization: Collocated simulations and visualization codes share resources such as local memory and network bandwidth. Efficiently using these resources is critical for an approach to be suitable at a very large scale.

Low impact on the code: Users are less likely to adopt an ISV approach if it requires many code changes in their simulation and the understanding of new tools [128], or if a visualization specialist should be consulted.

High adaptability: The adaptability of a system is its capability to offer a wide range of features without the need for a user to make changes in the connection between a (potentially running) simulation and a visualization backend.

These requirements motivate our work on leveraging the Damaris approach to support ISV in Chapter 4. Indeed, the main design principle of Damaris –namely, using dedicated cores to offload data processing tasks and I/O– offers a unique opportunity to develop a framework that addresses these four points.

2.5 Discussion: Addressing I/O Performance Variability

As supercomputers become larger and more complex, one main challenge consists of dealing with the large amounts of data generated by large-scale simulations. The current approach consists of periodically writing data in a parallel file system, using a high-level I/O library on top of a standardized interface such as MPI-I/O. This data is then read back for analysis and visualization purpose.

As we have shown, one major issue posed by this traditional approach to data management is the high performance variability it introduces. This variability can be observed at different levels. Within a single application, I/O contention across processes leads to large variations in the time each process takes to complete its I/O operations. From one I/O phase to another, this variability is further amplified, in particular due to interference with other applications sharing the same parallel file system. Finally, the trend of coupling simulations with visualization tools also exposes simulations to higher performance variability, as their run time does not depend anymore on their own scalability only, but also on the scalability of visualization algorithms. This particular problem is further amplified in the context of interactive in situ visualization, where the user himself and his interactions with the simulation become the cause of run-time variability.

To make an efficient use of future Exascale machines, it becomes important to provide data management solutions that do not solely focus on pure performance, but address performance variability as well. Addressing this variability is indeed the key to ensure that each and every component of these future platforms is used optimally.

The next chapters describes how, through a number of contributions, we addressed the challenges posed by the different aspects of this variability.

Chapter 3

Damaris: Leveraging Dedicated Cores to Hide the I/O Variability

Contents

In this Chapter, we propose the Damaris approach. Damaris leverages dedicated cores in multicore nodes of recent supercomputers to hide the I/O variability from the application and improve its overall performance. We describe the Damaris approach together

with its implementation. We then evaluate Damaris on three real HPC platforms: NICS's Kraken [63], NCSA's BluePrint and the French Grid'5000 testbed [53].

3.1 Addressing I/O Variability through Dedicated I/O Cores

The typical behavior in large-scale simulations, described in Chapter 2, consists of alternating computation phases and write phases. Often due to explicit barriers or communication phases, all processes perform I/O at the same time, causing network and file system contention. It is commonly observed that some processes exploit a large fraction of the available bandwidth and quickly terminate their I/O, then remain idle (typically from several seconds to several minutes) waiting for slower processes to complete their I/O. This variability, or jitter, can even be observed at relatively small scale, where measured I/O performance can vary by several orders of magnitude between the fastest and slowest processes [134]. With multicore architectures, this variability becomes even more of a problem, as multiple cores in a same node compete for the network access.

While most studies address I/O performance in terms of aggregate throughput and try to improve this metric by optimizing different levels of the I/O stack ranging from the file system to the simulation-side I/O library, few efforts have been made in addressing the I/O jitter. Yet it has been shown that this variability is highly correlated with I/O performance [134]. The origins of this variability can substantially differ due to multiple factors, including the platform, the underlying file system, the network, and the I/O pattern of the application. For instance, using a single metadata server in the Lustre file system causes a bottleneck when following the file-per-process approach (described in Chapter 2), a problem that PVFS or GPFS are less likely to exhibit. In contrast, byte-range locking in GPFS or equivalent mechanisms in Lustre cause lock contentions when writing to shared files. To address this issue, elaborate algorithms at the MPI-I/O level are used in order to maintain a high throughput [106]. Yet these optimization usually rely on all-to-all communications that impact their scalability.

The main contribution of this chapter is precisely to propose an approach that completely hides the I/O jitter exhibited by most widely used approaches to I/O management in HPC simulations: the file-per-process and collective-I/O approaches. Based on the observation that a first level of contention occurs when all cores of a multicore SMP node try to access the network for intensive I/O at the same time, our new approach to I/O, called Damaris (Dedicated Adaptable Middleware for Application Resources Inline Steering), *leverages dedicated I/O cores in each multicore SMP node along with shared memory to perform asynchronous data processing and I/O*. These key design choices build on the observation that it is often not efficient to use all cores for computation, and that reserving one core for tasks such as I/O management may not only help reducing jitter but also increase overall performance. Besides, most data written by HPC applications are only eventually read by analysis tasks but not used by the simulation itself. Thus write operations can be delayed without consistency issues.

Damaris takes into account user-provided information related to the behavior of the application and the intended use of the output in order to perform high-level I/O and data processing within SMP nodes. Some of these ideas have been explored partially in other efforts parallel to ours: a detailed positioning of Damaris with respect to these works is given

in Section 3.4.

3.2 The Damaris Approach

To sustain a high throughput and a low variability, it is preferable to avoid as much as possible access contention at the level of the network interface and at the level of the file system. One solution consists of reducing the number of writers (which reduces the network overhead and allows data servers to optimize disk accesses and caching mechanisms) and the number of generated files (which reduces the overhead in metadata servers). As the first level of contention occurs when several cores in a single SMP node try to access the same network interface, it becomes natural to work at the level of a node.

We propose to gather the I/O operations into one single core that will perform writes of larger data in each SMP node. In addition, this core is dedicated to I/O (i.e., it does not run the simulation code) in order to overlap writes with computation and avoid contention for accesses to the file system. The cores running the simulation and the dedicated cores communicate data through shared memory. We call this approach Damaris. Its design, implementation and API are described below.

3.2.1 Design Principles

The Damaris approach is based on four main design principles described hereafter.

Dedicated Cores

The Damaris approach is based on a set of processes running on dedicated cores in every SMP node. Each dedicated core performs post-processing and I/O in response to user-defined events sent by the simulation. We call "client" a process running the simulation, and "server" a process running on a dedicated core. One important aspect of Damaris is that dedicated cores do not run the simulation. With the current trend in hardware solutions, the number of cores per node increases. Dedicating one or a few cores thus has a diminishing impact on the performance of the simulation. Hence, our approach primarily targets SMP nodes featuring a large number of cores per node: 12 to 24 in our experiments.

Data Transfers through Shared Memory

Damaris handles large data transfers from clients to servers through shared memory. This makes a write as fast as a `memcpy` and also enables direct allocation of variables within the shared memory. This design principle will be especially useful in Chapter 4, where it will be used to efficiently share data between the simulation and visualization tools.

High-Level Data Abstraction

Clients write enriched datasets in a way similar to scientific I/O libraries such as HDF5 or NetCDF. That is, the data output by the simulation is organized into a hierarchy of groups and datasets, with additional metadata such as the description of variables, their type, unit,

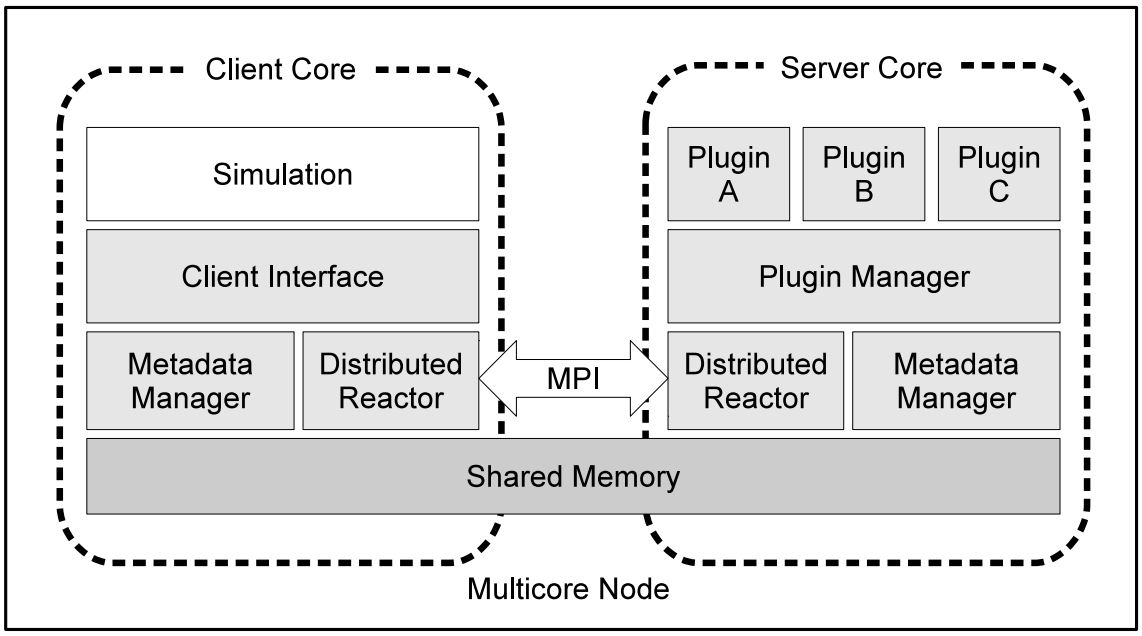


Figure 3.1: Software architecture of the implementation of Damaris.

and layout in memory. The dedicated cores thus have enough knowledge of incoming datasets to write them in an existing high-level formats. This design principle differs from other approaches that capture I/O operations at a lower level [72, 79]. These approaches indeed lose the semantic of the data being written. While our design choice forces us to modify the simulation so that it writes its data using Damaris' API, is allows to implement semantic-aware data processing functionalities in dedicated cores.

Extensibility through Plugins

Servers can perform data transformations prior to writing them. One major design principle in the Damaris approach is the possibility for users to provide these transformations through a plugin system, thus adapting Damaris to the particular requirements of their application. Implementing such a plugin system at a lower level would not be possible, as it would not have access to the high-level structure of the data (e.g., dimensions of arrays, data types, physical meaning of the variable within the simulation, etc.).

3.2.2 Architecture and Implementation

Figure 3.1 presents the software architecture underlying the Damaris approach. While Damaris can dedicate several cores in large multicore nodes, only one client and one server are represented here.

Damaris has been designed in a highly modular way and features a number of decoupled, reusable software components: The *Shared Memory* component handles the shared buffer and ensures the safety of concurrent allocations/deallocations. The *Distributed Re-*

actor handles communications between clients and servers, and across servers. The *Metadata Manager* stores high-level information related to the data being transferred (type, size, layout, etc.). Finally the *Plugin Manager* on the server side loads and runs user-provided plugins.

This modular architecture greatly simplified the adaptation to several HPC platforms and simulations, as well as the development of the extensions that are presented in Chapters 4 and 5. Therefore its implementation deserves an extensive description hereafter, including the technical choices that made our various contributions possible.

Shared Memory

Data communications between the clients and the dedicated cores are performed through the Shared Memory component. A large memory buffer is created on each node by the dedicated cores at start time, with a size set by the user (typically several MB to several GB). Thus the user has a full control over the resources allocated to Damaris. When a client submits new data, it reserves a segment of this shared-memory buffer. It then copies its data using the returned pointer so that the local memory can be reused.

Implementation: Damaris leverages the *Boost.Interprocess* library[1] to implement several versions of the Shared Memory component. The default version uses POSIX interprocess communication (IPC) primitives (e.g., `shm_open`, `shm_create`). These primitives being absent from some platforms such as BlueGene/P machines running CNK (IBM's Compute Node Kernel), other implementations are available that use System-V IPC (e.g., `shmget`, `shmat`).

As HPC simulations aim to run for days to weeks or even months, an important aspect when implementing the Damaris approach was to ensure the absence of memory leaks. In particular, letting users implement their own plugins leads to a high risk that the user loses a reference to an object in shared memory, and that the shared memory becomes full. In most of Damaris' implementation, we minimized this risk by using Boost's *shared pointers*. This ensures that when the last instance of a shared pointer to an object disappears, the object's destructor is invoked. But shared pointers do not work in shared memory. We therefore replaced them by a custom *DataSpace* abstraction that borrows the semantics of shared pointers while supporting the transfer of ownership between processes: a client writing data in shared memory creates a DataSpace and is responsible for it, i.e., the DataSpace will be deleted from memory if the client looses all references to it. Eventually, the client notifies the server of the existence of the DataSpace in shared memory, and transfers the responsibility to deallocate it to the server. That is, the existence of the DataSpace no longer depends on whether the client still holds a reference to it. The DataSpace will be effectively deallocated when all references to it in the server have disappeared.

Distributed Reactor

The Distributed Reactor is the most complex component of Damaris. It is primarily used for clients to send events to servers. These events indicate that some data has been written in shared memory, or that a plugin should be executed.

[1] See http://www.boost.org/

The Distributed Reactor builds on the *Reactor* design pattern [113] to provide the mean by which different cores communicate. Reactor is a behavioral pattern that handles requests concurrently sent to an application by one or more clients. The Reactor asynchronously listens to a set of *channels* connecting it to its clients. The clients send small events that are associated with event handlers (i.e., functions) in the Reactor. A synchronous event demultiplexer is in charge of queuing the events received by the Reactor and calling the appropriate event handlers. Contrary to a normal Reactor design pattern (as used in *Boost.ASIO* for example), our Distributed Reactor also provides elaborate collective operations.

Asynchronous atomic multicast: A process can broadcast an event to a group of processes at once. This operation is asynchronous, that is, the sender does not wait for the event to be processed by all receivers to resume its activity. A receiver only processes the event when all other receivers are ready to process it as well. It is also atomic, that is, if two distinct processes broadcast a different event, the Distributed Reactor ensures that all receivers will handle the two events in the same order.

Asynchronous atomic labeled barrier: We call a "labeled" barrier across a set of processes a synchronization barrier associated with an event (its label). After all processes reach the barrier, they all invoke the event handler associated with the event. This ensures that all processes agree to execute the same code at the same *logical* time. This primitive is asynchronous: it borrows its semantics from MPI 3's `MPI_Ibarrier` non-blocking barrier. It is atomic according to the same definition as the asynchronous atomic multicast.

These two distributed algorithms are very important in the design of post-processing tasks that involve communications between servers.

Implementation: Our implementation of the Distributed Reactor relies on MPI 2 communication primitives and, in particular, non-blocking *send* and *receive* operations. Events are simply implemented as 0-byte messages with the MPI *tag* carrying the type of the event. Since the MPI 3 standard provides new non-blocking collective functions such as `MPI_Ireduce` or `MPI_Ibarrier`, our Distributed Reactor could be easily re-implemented with these MPI 3 functions without any impact on the rest of Damaris' implementation.

Metadata Manager

The Metadata Manager component keeps information related to the data being written, including *variables*, *layouts* (describing the type and shape of blocks of data), *parameters*, etc. It is initialized using an XML configuration file.

This design principle is directly inspired by ADIOS [76] and also present in other tools such as EPSN [32]. In traditional data formats such as HDF5, several functions have to be called by the simulation to provide metadata information prior to actually writing data. The use of an XML file in Damaris presents several advantages. First, the description of data provided by the configuration file can be changed without changing the simulation itself, and the amount of code required to use Damaris in a simulation is reduced compared to existing data formats. Second, it prevents clients from transferring metadata to dedicated

cores through shared memory. Clients communicate only data along with the minimum information required by dedicated cores to retrieve the full description in their own Metadata Manager.

Implementation: The XML file constitutes a *model* of the data being produced by the simulation. We used Model-Driven Engineering (MDE) techniques to implement the Metadata Manager. Most of the source code of the Metadata Manager is indeed automatically generated from an XSD *metamodel*. This metamodel describes the concepts of *variables*, *layouts*, etc. as well as their relations to one another and how they are described in an XML format. The XSD file is used to synthesize C++ classes that correspond to the metamodel.

These engineering techniques greatly simplify the extensions of Damaris. They allow us to implement new features simply by adding a few lines in the XSD metamodel instead of manually programming all C++ classes required by this feature. For instance, we used this facility to extend Damaris to support complex data structures such as meshes (see Chapter 4).

Plugin Manager

Finally the Plugin Manager is the component that loads and stores plugins. Plugins are pieces of C++ or Python codes provided by the user. The Plugin Manager is capable of loading functions from dynamic libraries or scripts as well as from the simulation's code itself. It is initialized from the XML configuration file. Again, the use of a common configuration file between clients and servers allows different processes to refer to the same plugin through an identifier rather than its full name and attributes.

A dedicated core can call a plugin when it receives its corresponding event, or wait for all clients in a node or in the entire simulation to have sent the event. In these later cases, the collective algorithms provided by the Distributed Reactor ensure that all servers call the plugins in the same order.

Implementation: The Plugin Manager uses system calls such as `dlopen` to locate plugin functions in shared libraries or the program's code itself. To allow Python plugins, we use *Boost.Python* and the Python/C API [108] to wrap data in shared memory using NumPy arrays. NumPy is a very common data structures used in many Python libraries such as MatplotLib [82].

3.2.3 Client API

The Damaris approach is intended to be the basis for a generic, platform-independent, application-independent, easy-to-use tool. Our implementation provides client-side interfaces for C, C++ and Fortran applications written with MPI. This API can be summarized by the following functions.

- `damaris_initialize("config.xml")` initializes the resources used by Damaris using the configuration file given as parameter. All cores (clients and servers) call this function at the beginning of the simulation.

- `damaris_start()` is called by all cores to start servers on dedicated cores. The servers block within this function, while the clients return and proceed with the simulation.
- `damaris_get_client_comm()` provides an MPI communicator gathering only clients. Indeed the MPI_COMM_WORLD communicator contains both clients and servers and cannot be used by the simulation anymore for its collective communications.
- `damaris_write("var_name",data)` is called by clients. It copies the data in shared memory along with minimal information and notifies the server on the same node. All additional information such as the size of the data and its layout can be found by the servers in the configuration file.
- `damaris_signal("event_name")` is called by clients to send a custom event to the server in order trigger a plugin predefined in the configuration file.
- `damaris_end_iteration()` notifies the servers that the simulation has reached the end of an iteration and will start the next one. This allows dedicated cores to know that the data written in shared memory is consistent and nothing more should be expected for this iteration.
- `damaris_stop()` stops the servers on dedicated cores, making them leave the `damaris_start` function.
- `damaris_finalize()` frees the resources used by Damaris. It is called by all processes after servers have been stopped on dedicated cores (using `damaris_stop`) before terminating the simulation.

Listing 3.1 is an example of a Fortran program that makes use of Damaris. It writes a 3D array and sends an event to the dedicated core. The associated configuration file, shown in Listing 3.2, describes the data that is expected to be received by the servers, and the action to perform upon reception of the event. More specifically, line 11 of this XML file defines a *layout*, which describes the type and dimensions of a piece of data. Line 13 defines a variable that uses this layout. Finally line 17 associates an event with a function (or *action*) to be called when the event is received. It also locates the function within a dynamically-loaded library.

3.2.4 Writing with Damaris

Damaris is not a data format. It only provides a framework to dedicate cores for custom data processing and I/O tasks, to transfer data through shared memory and to call plugins. Following this approach, we implemented a plugin that gathers data from client cores and write them into HDF5 files. The next section uses this plugin to show how Damaris can hide all I/O costs and I/O variability from the simulation and improve its overall performance.

3.3 Experimental Evaluation

We evaluated our approach based on dedicated I/O cores by comparing it with standard approaches (file-per-process and collective I/O) with the CM1 atmospheric simulation, using three platforms: NICS's Kraken, Grid'5000 and NCSA's BluePrint cluster.

```
1  program example
2    integer ierr, is_client
3    real, dimension(64,16,2) :: my_data
4
5    ! initialization
6    call damaris_initialize_f("my_config.xml", MPI_COMM_WORLD, ierr)
7    call damaris_start_f(is_client, ierr)
8
9    if(is_client.eq.1) then
10     do while(...) ! simulation main loop
11       ...
12       call damaris_write_f("my_group/my_variable", my_data, ierr)
13       call damaris_signal_f("my_event", ierr)
14       call damaris_end_iteration_f(ierr)
15       ...
16     enddo
17     ! stopping the servers
18     call damaris_stop_f(ierr)
19   endif
20
21   ! finalization
22   call damaris_finalize_f(ierr)
23 end program example
```

Listing 3.1: Example of Fortran simulation using Damaris.

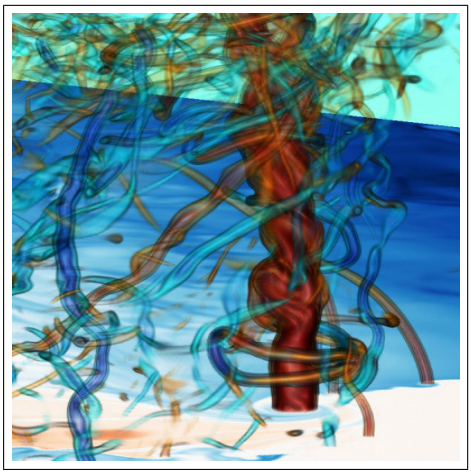


Figure 3.2: Simulation of a supercell producing a long-track EF5 tornado. This simulation was realized on NCSA's Blue Waters supercomputer by Leigh Orf (Central Michigan University) and Bob Wilhelmson (NCSA) using the CM1 code. See [46].

3.3.1 The CM1 Application

CM1 [7] (Cloud Model 1) is used for atmospheric research and is suitable for modeling small-scale atmospheric phenomena such as thunderstorms and tornadoes. It follows a typical behavior of scientific simulations, which alternate computation phases and I/O phases. The simulated domain is a fixed 3D array representing part of the atmosphere. Each point in this domain is characterized by a set of variables such as local *temperature* or *wind speed*. CM1 is written in Fortran 90. Parallelization is done using MPI, by distributing the 3D array along a 2D grid of equally-sized subdomains, each of which is handled by a process. The I/O

```
<?xml version="1.0"?>
<simulation name="example" language="fortran"
  xmlns="http://damaris.gforge.inria.fr/damaris/model">
  <architecture>
    <domains count="1"/>
    <dedicated cores="1"/>
    <buffer name="the_buffer" size="67108864" />
    <queue name="the_queue" size="100" />
  </architecture>
  <data>
    <layout name="my_layout" type="real" dimensions="64,16,2" />
    <group name="my_group">
      <variable name="my_variable" layout="my_layout" />
    </group>
  </data>
  <actions>
    <event name="my_event" action="my_function" using="my_plugin.so" />
  </actions>
</simulation>
```

Listing 3.2: Configuration file associated with the Fortran example.

phase leverages either HDF5 to write one file per process, or pHDF5 to write in a shared file in a collective manner. One of the advantages of using a file-per-process approach is that compression can be enabled, which cannot be done with pHDF5. However, at large process counts, the file-per-process approach generates a large number of files, making all subsequent analysis tasks intractable.

3.3.2 Platforms and Configuration

We conducted our experiments on three machines. Since our initial goal was to optimize CM1 for a future use on the upcoming Blue Waters Petascale supercomputer, we started with NCSA's BluePrint as it was supposed to be representative of Blue Waters' hardware. On this platform, we evaluate the scalability of the application with respect to the size of its output, with the file-per-process and Damaris approaches. We then experimented on the French Grid'5000, in particular, its *parapluie* cluster. This cluster features 24-core nodes, which makes it very suitable to our approach based on dedicated cores. Finally, we experimented on NICS's Kraken supercomputer, which, in addition to allowing runs at much larger scales, has a hardware configuration very close to that of Blue Waters' final design. These three platforms are detailed hereafter, along with the configuration of CM1 we used.

BluePrint

BluePrint is a test platform used at NCSA until 2011 when IBM was still in charge of delivering the Blue Waters supercomputer.[2]

[2] As IBM terminated its contract with NCSA in 2011 and Blue Waters was finally delivered by Cray, BluePrint was later decommissioned and replaced with a test platform, JYC, matching the new Blue Waters' design.

BluePrint features 120 Power5 nodes. Each node consists in 16 cores and includes 64 GB of memory. As file system, GPFS is deployed on 2 I/O servers. CM1 was run on 64 nodes (1024 cores), with a $960 \times 960 \times 300$-point domain. Each core handles a $30 \times 30 \times 300$-point subdomain with the standard approaches, that is, when no dedicated cores are used. When dedicating one core out of 16 on each node, computation cores handle a $24 \times 40 \times 300$-point subdomain. On this platform we vary the size of the output by enabling or disabling some of the variables from the output. We enabled the compression feature of HDF5 for all the experiments done on this platform.

Grid'5000

Grid'5000 [53] is a French grid testbed. We use its *parapluie* cluster (featuring 40 nodes of 2 AMD 1.7 GHz CPUs, 12 cores/CPU, 48 GB RAM) to run CM1 on 28 nodes (672 cores) and 38 nodes (912 cores). We deploy PVFS on 15 nodes of the *parapide* cluster (2 Intel 2.93 GHz CPUs, 4 cores/CPU, 24 GB RAM, 434 GB local disk). Each PVFS node was used both as I/O server and metadata server. All nodes communicate through a 20G InfiniBand 4x QDR link connected to a common Voltaire switch. We use Mpich [87] with ROMIO [125] compiled against the PVFS library, on a Debian operating system.

The total domain size in CM1 is $1104 \times 1120 \times 200$ points, so each core handles a $46 \times 40 \times 200$-point subdomain with a standard approach, and a $48 \times 40 \times 200$-point subdomain when one core out of 24 is used by Damaris.

Kraken

Kraken is a supercomputer deployed at NICS. It was ranked 11^{th} in the Top500 at the time of the experiments, with a peak Linpack performance of 919.1 Teraflops. It features 9408 Cray XT5 compute nodes connected through a Cray SeaStar2+ interconnect and running Cray Linux Environment (CLE). Each node has 12 cores and 16 GB of local memory. Kraken provides a Lustre file system using 336 block storage devices managed by 48 I/O servers and one metadata server.

On this platform, we have studied the weak scalability of the file-per-process, collective I/O and Damaris approaches, that is, we measured how the run time varies with a fixed amount of data per node. When all cores in each node are used by the simulation, each process handles a $44 \times 44 \times 200$-point subdomain. Using Damaris, each client process (11 per node) handles a $48 \times 44 \times 200$-point subdomain, which makes the total problem size equivalent for a given total number of cores.

3.3.3 Experimental Results

In this section, we present the results achieved in terms of I/O variability, I/O performance and resulting scalability of the application. We also evaluate two improvements to Damaris: compression and transfer delays.

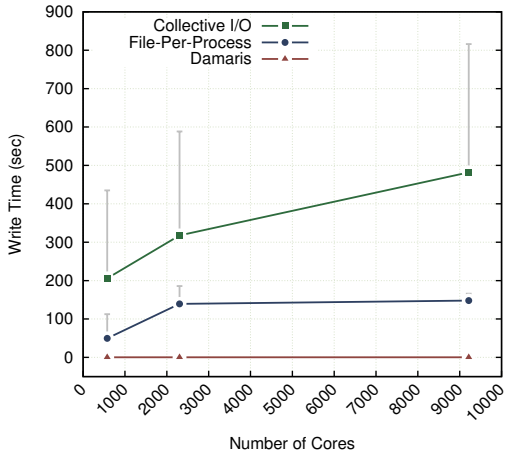


Figure 3.3: Duration of a write phase on Kraken (average and maximum). For readability reasons we don't plot the minimum write time. Damaris shows to completely remove the I/O variability while file-per-process and collective-I/O have a big impact on the run-time predictability.

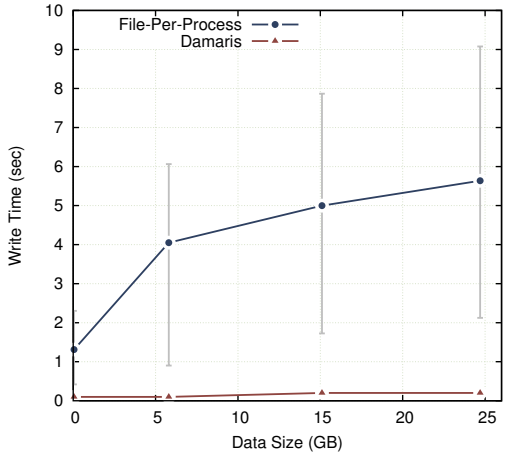


Figure 3.4: Duration of a write phase (average, maximum and minimum) using file-per-process and Damaris on BluePrint (1024 cores). The amount of data is given in total per write phase.

How Damaris Affects the I/O Variability

Impact of the Number of Cores on the I/O Variability: We studied the impact of the three I/O approaches –file-per-process, collective I/O, and Damaris– on the simulation's write time, with different number of cores. To do so, we ran CM1 on Kraken with 576, 2304 and 9216 cores.

Figure 3.3 shows the average and maximum duration of an I/O phase on Kraken from the point of view of the simulation. It corresponds to the time between the two barriers delimiting the I/O phase. This time is extremely high and variable with Collective I/O, achieving more than 800 seconds on 9216 cores. The average of 481 seconds still represents about 70% of the overall simulation's run time, which is unacceptable given the 5% limit usually accepted by users.

By setting the stripe size to 32 MB instead of 1 MB in Lustre, the write time went up to 1600 seconds with a collective I/O approach. This shows that bad choices of file system's configuration can lead to extremely poor I/O performance. Yet it is hard to know in advance what configuration of the file system and I/O libraries will lead to good performance.

Unexpectedly, the file-per-process approach appears to lead to a lower variability, especially at large process count, and better performance than collective I/O. Yet it still represents an unpredictability (difference between the fastest and the slowest phase) of about $\pm$17 seconds. For a one month run, writing every 2 minutes would lead to an uncertainty of several hours to several days of run time.

When using Damaris, we dedicate one core out of 12 on each node, thus potentially reducing the computation performance for the benefit of I/O efficiency (the impact on overall application performance is discussed in the next section). As a means to reduce the I/O variability, this approach is clearly effective: the time to write from the point of view of the simulation is cut down to the time required to perform a series of copies in shared memory. It leads to a write time of 0.2 seconds and does not depend anymore on the number of processes. The variability is in order of $\pm$0.1 seconds (too small to be represented here).

Impact of the Amount of Data on the I/O Variability: On BluePrint, we vary the amount of data. We aim to compare the file-per-process approach with Damaris with respect to different output sizes.

The results are reported in Figure 3.4. As we increase the amount of data, the variability of the I/O time increases with the file-per-process approach. With Damaris however, the write time remains in the order of 0.2 seconds for the largest amount of data and the variability in the order of $\pm$0.1 seconds again.

Impact of the Hardware: Finally, we studied the impact of the hardware on the I/O variability using Grid'5000. With its large number of cores per node (24) and a network that is substantially less performant than that of Kraken and BluePrint, we aim to illustrate the large variation of write time across cores for a single write phase.

We ran CM1 using 672 cores, writing a total of 15.8 GB uncompressed data (about 24 MB per process) every 20 iterations. With the file-per-process approach, CM1 reported spending 4.22% of its time in I/O phases. Yet the fastest processes usually terminate their I/O in less than 1 second, while the slowest take more than 25 seconds. Figure 3.5 shows the CDF of

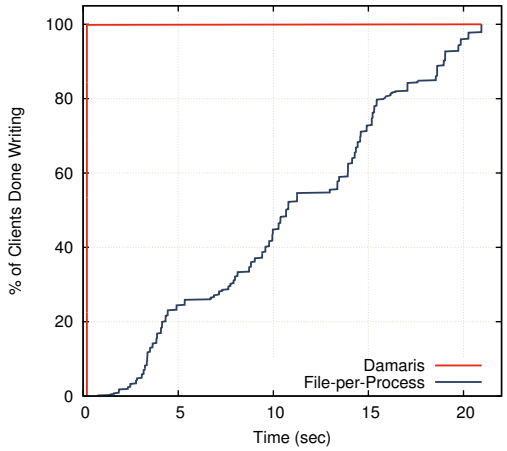


Figure 3.5: Cumulative distribution function of the write time when running CM1 on 672 processes of Grid'5000.

write times for one of these write phases, with a file-per-process approach and with Damaris. It shows that, with a file-per-process approach, there is a large difference in write time between the fastest and the slowest process, due to access contention either at the level of the network or within the file system. With Damaris however, all processes complete there write at the same time. This is due to the absence of contention when writing in shared memory.

Conclusion: Our experiments show that by replacing write phases with simple copies in shared memory and by leaving the task of performing actual I/O to dedicated cores, Damaris is able to completely hide the I/O variability from the point of view of the simulation, making the application run time more predictable.

Application's Scalability and I/O Overlap

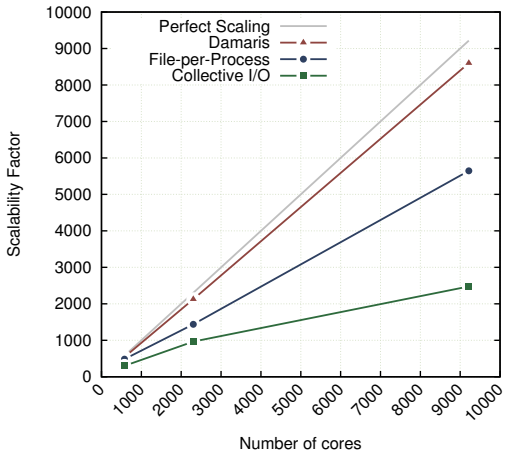
Impact of Damaris on the Scalability of CM1: CM1 exhibits a very good weak scalability and very stable performance when it does not perform any I/O. Thus, as we increase the number of cores, the scalability becomes mainly driven by the scalability of the I/O phases. To measure the scalability of an approach, we define the scalability factor as follows:

$$S = N * \frac{C}{T_N}, \tag{3.1}$$

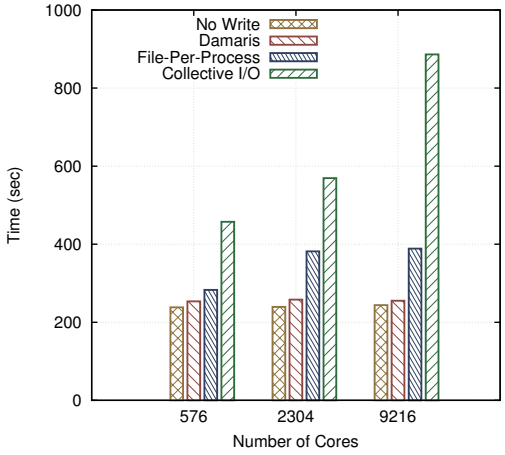
where N is the number of cores considered. We take as a baseline the time C of 50 iterations of CM1 on 576 processes without dedicated core and without any I/O. T_N is the time CM1 takes to perform 50 iterations plus one I/O phase, on N cores. A perfect scalability factor on N cores should equal N.

The scalability factor on Kraken for the three approaches is given in Figure 3.6 (a). Figure 3.6 (b) shows the associated application run time for 50 iterations plus one write phase. Damaris exhibits a nearly perfect scalability where other approaches fail to scale. In particular, going from collective I/O to Damaris leads to a 3.5× speedup on 9216 cores.

Spare Time in Damaris: Since the scalability of our approach comes from the fact that I/O overlaps with computation, we still need to show that the dedicated cores have enough time to perform the actual I/O while computation goes on.

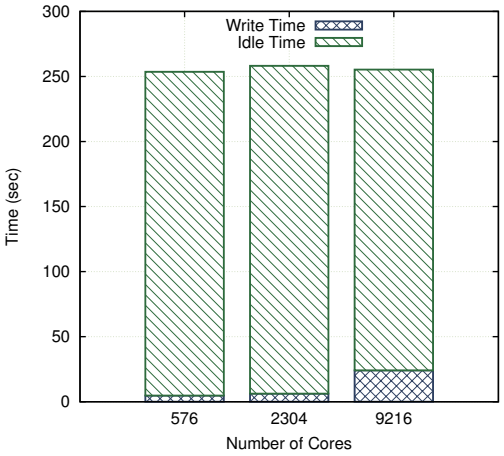


(a) Scalability Factor on Kraken

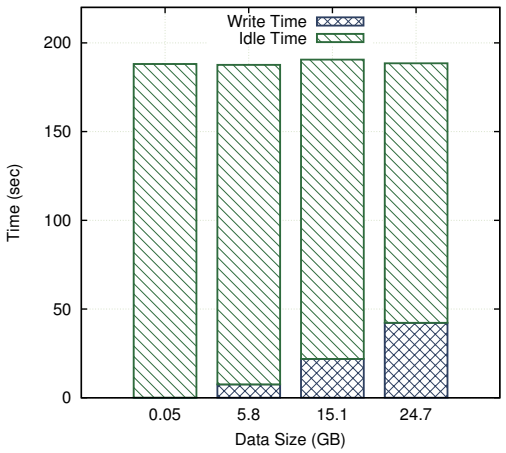


(b) Run Time on Kraken

Figure 3.6: Scalability factor (a) and overall run time (b) of the CM1 simulation for 50 iterations and 1 write phase, on Kraken.



(a) Write / Idle Time on Kraken



(b) Write / Idle Time on BluePrint

Figure 3.7: Time spent by the dedicated cores writing data for each iteration, and time spared (idle). The spare time is the time dedicated cores are not performing any task.

Figure 3.7 shows the time used by the dedicated cores to perform the I/O on Kraken and BluePrint, as well as the time they remain idle, waiting for the next iteration to complete.

As the amount of data on each node is the same, the only explanation for the dedicated cores to take more time at larger process count on Kraken is the access contention for the file system. On BluePrint the number of processes is constant for each experiment, thus the differences in write time come from the different amounts of data. In all configurations, our experiments show that Damaris spares a lot of time, during which it remains idle. Similar results were obtained on Grid'5000.

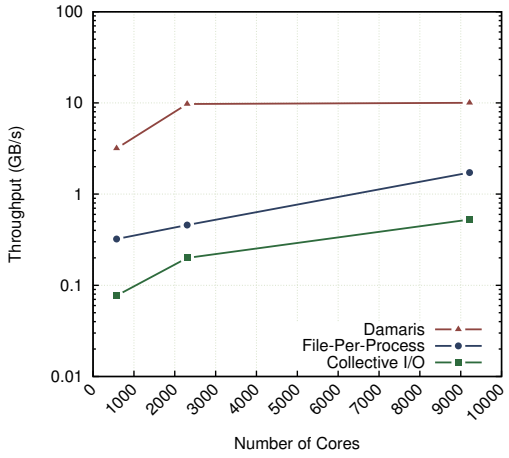


Figure 3.8: Average aggregate throughput achieved on Kraken with the different approaches. Damaris shows a 6 times improvement over the file-per-process approach and 15 times over Collective I/O on 9216 cores.

	Aggregate throughput
File-per-process	695 MB/s
Collective-I/O	636 MB/s
Damaris	4.32 GB/s

Table 3.1: Average aggregate throughput on Grid'5000, with CM1 running on 672 cores.

Conclusion: On all platforms, Damaris shows that it can fully overlap writes with computation and still remain idle 75% to 99% of time (see Figure 3.7). Thus, without impacting the application, it becomes possible to further increase the frequency of output or perform in situ data analysis and visualization. The later case will be the focus of Chapter 4.

Effective I/O Throughput

We then studied the effect of Damaris on the aggregate throughput observed from the simulation to the file system. Note that in the case of Damaris, this throughput is only observed by the dedicated cores.

Figure 3.8 presents the aggregate throughput obtained with the three approaches on Kraken. At the largest scale (9216 cores) Damaris achieves an aggregate throughput about 6 times higher than the file-per-process approach, and 15 times higher than collective I/O. The results obtained on 672 cores of Grid'5000 are presented in Table 3.1. The throughput achieved with Damaris here is more than 6 times higher than the other two approaches. Since compression was enabled on BluePrint, we don't provide the resulting throughputs, as it would depend on the overhead of the compression algorithm and the resulting size of the data.

Conclusion: By avoiding process synchronization and access contention at the level of a node and by gathering data into bigger files, Damaris reduces the pressure on metadata servers and issues bigger operations that can be more efficiently handled by storage servers. As a result, on all platforms, Damaris substantially increases the aggregate throughput, thus making a more efficient use of the file system.

3.3.4 Improvements: Leveraging the Spare Time

Section 3.3.3 showed that, with the CM1 application, dedicated cores remain idle most of the time. In order to leverage the spare time in dedicated cores, we implemented two improvements: compression, and transfer delays.

Compression

We used dedicated cores to compress the output data prior to writing it. Using lossless gzip compression, we observed a compression ratio of 187%. When writing data for offline visualization, the floating point precision can also be reduced to 16 bits, leading to nearly 600% compression ratio when coupling with gzip. On Kraken, the time required by dedicated cores to compress and write data was twice longer than the time required to simply write uncompressed data. Yet contrary to enabling compression in the file-per-process approach, the overhead and jitter induced by the compression phase is completely hidden within the dedicated cores, and do not impact the running simulation. In other words, *compression is offered for free* by Damaris.

Data Transfer Delays

Additionally, we implemented in Damaris the capability to delay data movements. The algorithm is very simple and does not involve any communication between processes: each dedicated core computes an estimation of the duration of an iteration of the simulation by measuring the time between two consecutive calls to `damaris_end_iteration` (about 230 seconds on Kraken). This time is then divided into as many slots as there are dedicated cores. Each dedicated core waits for its slot before writing. This avoids access contention at the level of the file system. We evaluated this strategy on 2304 cores on Kraken, the aggregate throughput reaches 13.1 GB/s on average, instead of 9.7 GB/s when this algorithm is not used.

Summary: These two improvements have also been evaluated on 912 cores of Grid'5000. All results are synthesized in Figure 3.9, which shows the average write time in dedicated cores. The delay strategy reduces the write time in both platforms. Compression however introduces an overhead on Kraken, thus we are facing a tradeoff between reducing the storage space used or reducing the spare time. A potential optimization would be to enable or disable compression at run time depending on the need to reduce write time or storage space.

3.4 Related Work

3.4.1 Positioning Damaris in the "I/O Landscape"

Through its capability of gathering data into larger buffers and files, Damaris can be compared to the data aggregation feature in ROMIO [125]. This feature is an optimization of collective I/O that leverages a subset of processes, called "aggregators" to actually perform

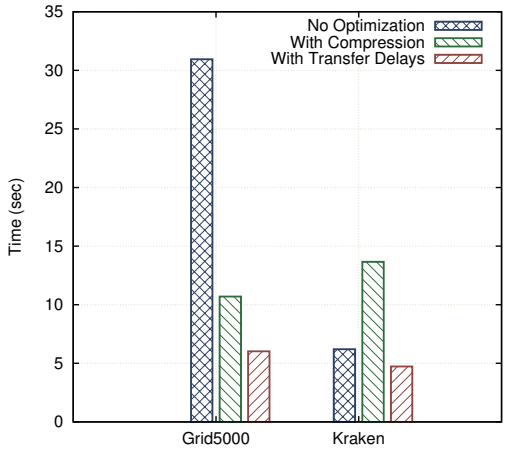


Figure 3.9: Write time in the dedicated cores when enabling compression or transfer delays.

the I/O on behalf of other processes. Yet, data aggregation is performed synchronously in ROMIO: all processes that do not perform actual writes in the file system must wait for the aggregator processes to complete their operations. Besides, aggregators are not dedicated processes, they run the simulation after completing their I/O. Through space partitioning, Damaris can perform data aggregation and potential transformations in an asynchronous manner and still use the idle time remaining in the dedicated cores.

Other efforts focus on overlapping computation with I/O in order to reduce the impact of I/O latency on overall performance. Overlap techniques can be implemented directly within simulations [103], using asynchronous communications. Non-blocking I/O primitives started to appear as part of the current MPI 3 standard, but as explained in Chapter 2, these primitives are still implemented as blocking in practice.

Other approaches leverage data-staging and caching mechanisms [93, 56], or forwarding approaches [3] to achieve better I/O performance. Forwarding architectures run on top of dedicated resources in the platform, which are not configurable by the end-user, that is, the user cannot run custom data processing in forwarding resources. Similarly to the parallel file system, these dedicated resources are shared by all users. This leads to cross-application access contention and thus, to I/O variability. However, the trend toward I/O delegate systems underlines the need for new I/O approaches. Our approach relies on dedicated I/O cores at the application level rather than relying on hardware I/O-dedicated or forwarding nodes, with the advantage of letting users configure their dedicated resources to best fit their needs.

The use of local memory to alleviate the load on the file system is not new. The Scalable Checkpoint/Restart (SRC) by Moody et al. [84] already makes use of node-level storage to avoid the heavy load caused by periodic global checkpoints. Yet their work does not use dedicated resources or threads to handle or process data, and the checkpoints are not asynchronous.

3.4.2 Dedicated-Core-Based Approaches

Space partitioning at the level of multicore SMP nodes, along with shared memory, has also successfully been used to optimize communications between coupled simulations [144]. In

contrast to this work, which does not focus on I/O, our goal is precisely to optimize I/O to remove the performance bottleneck usually created by massively concurrent I/O and the resulting variability.

Closest to our work are the approaches by Li et al. [72], and Ma et al. [79]. While the general goals of these approaches are similar (leveraging service-dedicated cores for non-computational tasks), their design is different, and so is the focus and the (much lower) scale of their evaluation. The first one mainly explores the idea of using dedicated cores in conjunction with SSDs to improve the overall I/O throughput. Architecturally, it relies on a FUSE interface, which introduces unnecessary copies through the kernel and reduces the degree of coupling between cores. Using small benchmarks we noticed that such a FUSE interface is about 10 times slower in transferring data between cores than using shared memory. In the second, active buffers are handled by dedicated processes that can run on any node and interact with cores running the simulation through the network. In contrast to both approaches, Damaris makes a much more efficient design choice using the shared intra-node memory, thereby avoiding costly copies and buffering. The approach defended by Li et al. is demonstrated on a small 32-node cluster (160 cores), where the maximum scale used in the work by Ma et al. is 512 cores on a Power3 machine, for which the overall improvement achieved for the global run time is marginal. Our experimental analysis is much more extensive and more relevant for today's scales of HPC simulations: we demonstrated the excellent scalability of Damaris on a real supercomputer (Kraken, ranked 11^{th} in the Top500 supercomputer list at the time of the experiments) with up to almost 10,000 cores, and with the CM1 tornado simulation, one of the target applications of the Blue Waters post-Petascale supercomputer project. We demonstrated not only a speedup in I/O throughput by a factor of 15 (never achieved by previous approaches), but we also showed that Damaris totally hides the I/O jitter and substantially cuts down the application run time at such high scales. With Damaris, the execution time for CM1 at this scale is even divided by 3.5 compared to approaches based on collective I/O! Moreover, we further explored how to leverage the spare time of the dedicated cores. We demonstrated for example that it can be used to compress data by a factor of 6.

3.5 Conclusions and Discussion

Efficient management of I/O variability on Petascale and post-Petascale HPC infrastructures is a key challenge. I/O variability indeed has a huge impact on the ability to sustain high performance at the scale of such machines. Proposing efficient solutions to reduce its effects is critical for preparing the advent of Exascale supercomputers and their efficient usage by applications at full-machine scale.

The Damaris approach can efficiently overlap computation with I/O by using dedicated cores in multicore nodes. As a result, all I/O costs and their associated variability are hidden from the simulation. This makes the simulation's overall run time less subject to variability.

3.5.1 Theoretical Usefulness

An important concern that potential users of Damaris have is whether or not our approach will effectively bring an improvement to their application. While our experiments demon-

strate such an improvement with one simulation (CM1), it can be useful to understand, from a theoretical perspective which conditions are necessary for Damaris to be useful.

We consider a machine featuring N cores per nodes. Let us call W_{std} the time spent writing and C_{std} the computation time between two I/O phases with a traditional approach (i.e., file-per-process or collective I/O), C_{ded} the computation time when the same workload is divided across $N-1$ cores per node. We here assume that the I/O time is null or negligible when using the dedicated core from the point of view of the simulation, which is experimentally verified. We call W_{ded} the time that the dedicated cores spend writing. A theoretical performance benefit of our approach occurs when

$$W_{std} + C_{std} > max(C_{ded}, W_{ded}) \tag{3.2}$$

Assuming a perfect scalability of the program when parallelized across N cores per node, we have $C_{ded} = \frac{N}{N-1} C_{std}$. Provided that the dedicated cores fully manage to overlap computation with I/O the inequality 3.2 becomes

$$W_{std} + C_{std} > C_{ded}, \tag{3.3}$$

which is true when the program spends at least $p = \frac{100}{N-1}$ percent of its time in I/O phase with a traditional approach. As an example, with 24 cores $p = 4.35\%$, which is already under the 5% usually admitted for the I/O phase of such applications. Thus assuming that the application effectively spends 5% of the time writing data, on a machine featuring more than 24 cores per node, it is already more efficient to dedicate one core per node to hide the I/O phase. With recent supercomputers featuring more and more cores per node, this percentage decreases, making the Damaris approach more and more relevant to use.

3.5.2 Key Results

Results obtained with the CM1 atmospheric simulations, one of the challenging target applications of the Blue Waters post-Petascale supercomputer, clearly demonstrate the benefits of Damaris in experiments with up to 9216 cores performed on the Kraken supercomputer. Damaris completely hides the I/O variability and all I/O-related costs and achieves a throughput 15 times higher than existing approaches. Besides, it reduces the application's execution time by 35% compared to the conventional file-per-process approach. The execution time is divided by 3.5 compared to approaches based on collective I/O. Moreover, it substantially reduces storage requirements, as the dedicated I/O cores enable overhead-free data compression with up to 600% compression ratio.

3.5.3 Let's Use our Spare Time

We have seen in Section 3.3.3 that dedicated cores remain idle a substantial fraction of the time. This brings room for potential improvements. In the next Chapter, we use this spare time to provide efficient in situ visualization.

Chapter 4

Extending Damaris to Support In Situ Visualization

Contents

As we approach Exascale, the limits of offline analysis [51] will be magnified. The previous chapter showed that simulations already endure scalability issues arising from unmatched computation and I/O performance as well as higher I/O variability. Also, with an increase in problem size it becomes more difficult to transfer data from one

supercomputer to another, and data-parallel visualization tasks start to suffer from the same I/O bottleneck [17, 141]. Consequently, in situ visualization (ISV) has been proposed to couple the simulation with visualization tools and perform visualization while the simulation runs.

This chapter explores the use of dedicated cores for in situ visualization. Based on requirements drawn for in situ visualization in Chapter 2, we extend the Damaris approach presented in Chapter 3. We illustrate this extension through a new framework, called Damaris/Viz, which provides the following contributions to the field of ISV.

1. It reduces visualization-related code modifications to a minimum in existing simulations;
2. It adapts to the specific needs of simulations by gathering the capabilities of existing visualization packages under a unified data management interface;
3. It hides the performance impact of a coupled visualization code by using dedicated cores to execute it in parallel with the simulation;
4. It efficiently leverages double-buffering techniques with shared memory to optimize the memory usage.

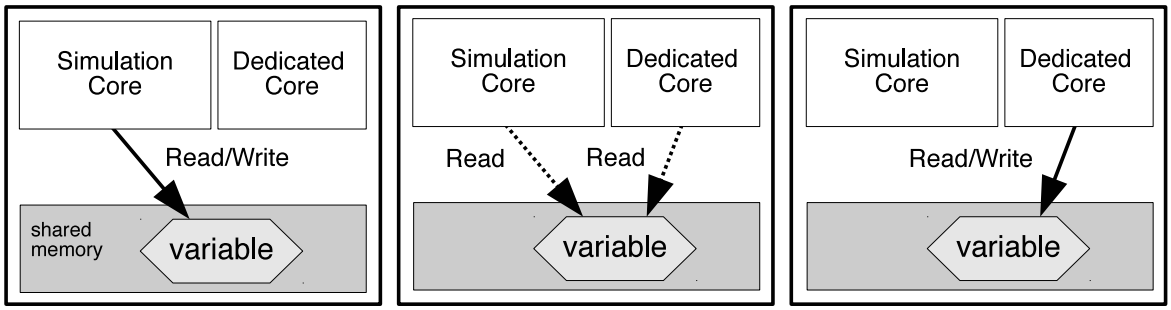
We compare our framework to representative packages: VisIt [135], ParaView [101], and custom analysis modules written using the C/Python interface. We evaluate it in the context of two real simulations: CM1, and the Nek5000 [98] CFD solver. These experiments were carried out on the Blue Waters [5] machine at NCSA and on the Grid'5000 testbed, with representative visualization scenarios.

4.1 In Situ Visualization With Damaris

4.1.1 Towards a New In Situ Visualization Framework

Coupling simulations with visualizations requires understanding the interfaces of both pieces of software. These interfaces can be difficult to master and the coupling may necessitate significant changes to the code of the simulation. Additionally, changing from one visualization software to another requires other modifications in the code to adapt to each software's API. Yet, many software provides the same features. For instance, both ParaView and VisIt can work with rectilinear meshes, but the codes required to support VisIt or Para View in situ are very different. It thus becomes necessary to unify the API provided by different visualization packages under a common interface. This interface should be simple and flexible enough to get widely accepted by HPC users.

A useful feature for ISV is the ability to work on raw in-memory data without performing any copy, thus reducing the memory consumption of in situ analysis tasks. There is a trend to reduce local memory per core on next-generation supercomputers; consequently this "zero-copy" property becomes invaluable. In addition, the ability to overlap simulation with visualization has performance benefits. Periodically stopping the simulation to perform visualization tasks increases the overall run time of a simulation as well as the run time variability. Interactivity with an end-user imposes additional overhead and variability.



(a) After damaris_alloc (b) After damaris_commit (c) After damaris_clear

Figure 4.1: Semantics of the three functions: (a) at iteration 1, an segment is allocated for a given variable through `damaris_alloc`, the simulation holds it. (b) Eventually, a call to `damaris_commit` notifies the dedicated core of the location of the data. From there on, the segment can be read by both processes, but should not be written or deleted by neither of them. Finally, (c) a call to `damaris_clear` indicates that the simulation does not need the segment anymore, dedicated cores can modify it, delete it or move it to a persistent storage.

Therefore, users take the risk of slowing down their computation with every connection they make to their running simulation.

These considerations motivated our choice to build our framework using Damaris. As shown in the previous chapter, the dedicating cores still remain idle a large fraction of the time that could be leveraged for additional data processing. Additionally, the use of intra-node shared memory in Damaris enables efficient resource sharing in the context of simulation/visualization coupling. In this chapter we show that Damaris can also serve as a bridge between a simulation and various visualization software through a unified interface. We call this new framework Damaris/Viz.

4.1.2 Damaris/Viz: an In Situ Visualization Framework Based on Damaris

The initial implementation of Damaris, seen in Chapter 3 provides a `damaris_write` function, with the idea of imitating classical file-based I/O libraries (for example HDF5, NetCDF or ADIOS). When entering an I/O phase, the simulation calls this function to copy its local data into a shared memory segment, and notifies the dedicated cores that data has been written. This API is suitable when the simulation is not memory-constrained. With a visualization process sharing the simulation's resources however, it becomes necessary to limit the memory used by Damaris.

The use of space-partitioning in Damaris presents two problems: the first is to expose the data to visualization components, and the second, more important is to ensure the consistency of simultaneous accesses from different components to the same data.

Leveraging Built-in Double Buffering

By studying several simulations using time-varying data, we noticed a frequent use of double-buffering techniques, where two buffers are used to store different versions of the

same variable: a first buffer holds data as input for the solver and another one is used for storing the results. The two buffers are then swapped before entering the next iteration so that the buffer that contains the most recent output serves as the new input, and the old input buffer, no longer useful, serves as output buffer.

We can thus decompose the life of a buffer in three phases:

1. Equations are solved and the output data is written in one of the buffers, to serve as input for the next iteration. During this step, the buffer is intensively written;
2. The data in this buffer serves as input for the next iteration and is not written over, however it is intensively read;
3. The data in this buffer is no longer needed by the simulation. The buffer can be reused to write the output of the next iteration.

These observations indicate that there is potential for using the input buffer also as input for visualization tasks, as this buffer is only read by the simulation and not modified until the swap phase. Using dedicated cores and shared memory, doing so requires to allocate these buffers directly in shared memory so that dedicated cores can access them. We thus provided new functions to the Damaris API in order to better leverage this double-buffering techniques.

- `damaris_alloc("variable")` is similar to *malloc* (or *allocate* in Fortran, *new* in C++). It is called by clients to allocate a portion of shared memory to hold the variable for a given iteration and returns a pointer. Only the simulation is aware of this allocation, dedicated cores cannot access the data. The returned buffer is expected to be used as output buffer for the next iteration.
- `damaris_commit("variable")` is called by clients when the simulation has finished writing to the current buffer associated with the variable. It sends the location of the data to the dedicated cores. Both the simulation and dedicated cores can read the data. At this point, clients will use the buffer as input for the next iteration while dedicated cores will use it as input for visualization tasks.
- `damaris_clear("variable")` is called by clients to notify the dedicated cores that the committed data for this variable will no longer be used by the simulation. It can safely be processed, stored or removed from shared memory. The clients will issue another `damaris_alloc` to get a new portion of shared memory to use as output buffer.

The only modification needed in the code of the simulation involves changing the allocation methods of visualizable variables, in order to allocate them in a place from which the dedicated cores can immediately access them. The `damaris_clear` function does not free the memory; simulation processes expect the dedicated cores to maintain enough free space in shared memory by removing old data. Dedicated cores must free the memory quickly enough to avoid consuming all of the shared memory. In the event that shared memory is full rather than blocking the simulation, `damaris_alloc` uses the local memory of the process instead of the shared memory; `damaris_commit` has no effect; `damaris_clear` frees the memory, and `damaris_end_iteration` (already presented in Chapter 3) will notify servers

that the data has not been placed in shared memory. A blocking version of this API, in which `damaris_alloc` waits for enough memory to be available, is also provided. Figure 4.1 summarizes the semantics of the three functions.

Similarly to the original API of Damaris, the only parameter needed for most Damaris functions is the name of a variable. Other required information such as the size of the data and number of domains are supplied by the configuration file.

Synchronizing Dedicated Cores

One challenge posed by in situ visualization using Damaris is that all dedicated cores should be synchronized to perform ISV tasks at the same moment. In particular, if a user connects a visualization software, it may happen that not all dedicated cores see the same iteration as completed if some clients have called `damaris_end_iteration` and some haven't yet.

The distributed algorithms provided by the Distributed Reactor (described in Chapter 3) help alleviating this problem. Upon reception of an "end of iteration" event from its clients, a dedicated core enters an atomic asynchronous labeled barrier, with the iteration number as label. The connection of any visualization software triggers a uniform broadcast from the first dedicated core (i.e., the dedicated core who's rank is 0 in MPI_COMM_WORLD) to all dedicated cores. The *uniform* nature of both primitives ensures that either all dedicated cores execute the barrier then the visualization task, or the other way around. Having some dedicated cores execute the barrier while other execute the visualization task would lead to a deadlock. The Distributed Reactor ensures that this does not happen.

Providing a Time-Partitioning Mode

Although we advocate for using dedicated cores as a mean to hide the performance variability induced by in situ visualization tasks, some simulations may not be able to afford dedicating cores, or some researchers could still prefer to perform visualization in a time-partitioning manner. Thus we also implemented in Damaris the possibility to use time partitioning.

The use of dedicated cores in Damaris can be simply turned off using the XML file as follows: `<dedicated cores="0" />`. In this mode, any call to `damaris_signal` triggers the plugin locally, in a synchronous way. In situ visualization is performed at every call to `damaris_end_iteration` and all cores synchronously participate in the ISV task.

This mode also allows us to compare the time-partitioning and space-partitioning approaches on the basis of the same framework, that is, without the need for different modifications in the simulation's code.

4.1.3 Connection to Existing Visualization Packages

Now that Damaris provides an API to enable efficient communication through shared memory, we can connect it to existing visualization and analysis packages, including VisIt or ParaView, in order to build a full ISV framework.

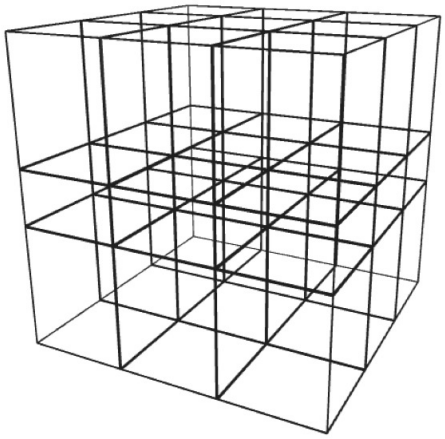


Figure 4.2: Example of a $4 \times 4 \times 4$ rectilinear grid described by three arrays of coordinates. In this example there is a scalar value (such as *temperature* or *wind velocity*) at each node. The mesh itself is described through three coordinate arrays: `mesh_x = {0.0,1.0,2.0,3.0}; mesh_y = {0.0,1.0,2.0,3.0}; mesh_z = {0.0,1.2,1.8,3.0}`.

Support for VisIt and ParaView

Both VisIt and ParaView perform in situ visualization from in-memory data. Given that each of these software has strengths, a major advantage of our approach is the ability to switch between them with no code modification in the simulation.

We leverage the configuration file in Damaris to provide the necessary information to bridge the simulation to existing visualization software. By investigating the in situ interfaces of different visualization packages including ParaView, VisIt, ezViz [33] and VTK [115], we devised with a generic description of visualizable structures such as meshes, points or curves. Listing 4.1 presents how a mesh drawn in Figure 4.2 is described using an XML configuration file. The coordinates of the points in the mesh are variable entries, declared in lines 21 to 28. These variables are not themselves *visualizable*, since they compose a mesh. The mesh is described lines 15 to 19. Each *coord* entry refers to the variable from which to get the coordinates. The type of mesh (*rectilinear* here) is provided, as well as the topological dimension, to help visualization software build the mesh from its coordinates. Finally, a variable such as "temperature", declared in line 30, can be mapped onto a mesh using its *mesh* attribute that has to refer to a declared mesh entity.

This file provides the necessary information for Damaris to execute VisIt or ParaView codes, but hides from the user the details of those interfaces. Therefore, both VisIt and ParaView (or other visualization software) can be used without code modification in the simulation. Listing 4.2 shows the lines of code changed in the simulation itself.

Python Support for Simple Analysis

We enhanced the plugin system of Damaris to load Python scripts. From these scripts, all variables are wrapped into NumPy arrays. Related metadata information (current iteration number, boundaries of a data chunk, process IDs for writers) are also accessible to Python. Wrapping C arrays into NumPy arrays does not produce a copy of data, thus Python plugins work on the original data supplied by the simulation and provide an easy way to write analysis tasks without any modification to simulation code. Listing 4.3 provides an example of a statistical computation performed on all chunks of iteration 1 of the data. The SciPy and

```
<simulation name="mesh" language="c"
    xmlns="http://damaris.gforge.inria.fr/damaris/model">
    ...
    <data>
        <parameter name="w" type="int" value="4" />
        <parameter name="h" type="int" value="4" />
        <parameter name="d" type="int" value="4" />

        <layout name="mesh_x_layout" type="float" dimensions="w" />
        <layout name="mesh_y_layout" type="float" dimensions="h" />
        <layout name="mesh_z_layout" type="float" dimensions="d" />

        <layout name="data_layout" type="double" dimensions="w,h,d"/>

        <mesh name="mesh3d" type="rectilinear" topology="3">
            <coord name="coordinates/x3d" unit="m" label="Width"/>
            <coord name="coordinates/y3d" unit="m" label="Height"/>
            <coord name="coordinates/z3d" unit="m" label="Depth"/>
        </mesh>

        <group name="coordinates">
            <variable name="x3d" layout="mesh_x_layout"
                visualizable="false" time-varying="false" />
            <variable name="y3d" layout="mesh_y_layout"
                visualizable="false" time-varying="false" />
            <variable name="z3d" layout="mesh_z_layout"
                visualizable="false" time-varying="false" />
        </group>

        <variable name="temperature" layout="data_layout" mesh="mesh3d"/>
    </data>
    ...
    <visit>
        <path>/usr/local/visit</path>
    </visit>
</simulation>
```

Listing 4.1: Description of a mesh in the Damaris/Viz configuration.

Matplotlib Python libraries provide a wide range of functionalities to write diagnostic tasks or generate images from simulation data. However, upon initial testing, we noticed that performance degrades when loading Python modules simultaneously from many processes; we thus recommend using Python for small analyses, and we decided to make performance comparisons among only those packages appropriate for large scales.

4.1.4 Automatic Adaptation of Output Frequency

The choice of non-blocking allocation functions, described in Section 4.1.2 has an immediate impact on the behavior of Damaris with respect to visualization. Rather than stalling the simulation, a shortage of memory causes the dedicated cores to skip rendering frames and

```
1  float* mesh_x = damaris_alloc("coordinates/x3d");
2  float* mesh_y = damaris_alloc("coordinates/y3d");
3  float* mesh_z = damaris_alloc("coordinates/z3d");
4  double* temp = damaris_alloc("temperature");
5  ...
6  damaris_commit("coordinates/x3d");
7  damaris_commit("coordinates/y3d");
8  damaris_commit("coordinates/z3d");
9  ...
10 damaris_commit("temperature");
11 ...
12 damaris_clear("temperature");
13 ...
14 damaris_end_iteration();
```

Listing 4.2: Allocation for data accessed by Damaris. The size is given in the Damaris configuration file.

```
1  var = damaris.open("temperature")
2  for chunks in var.select( iteration = 1 )
3         print numpy.average(chunks.data)
```

Listing 4.3: Accessing simulation's data through the Damaris Python interface: computing the average of a value.

free memory instead. Thus, Damaris self-adapts to the complexity of the visualization task and outputs the maximum number of frames that the dedicated cores are able to render without impacting the simulation. In other words, it is possible that visualization is only performed when it doesn't cost anything to the simulation nor impact the variability of its run time, which fits well with certain typical in situ use cases, such as simply verifying that a simulation is producing correct output.

4.2 Impact on Development and Flexibility

We compared our framework to two representative software packages used for tightly-coupled ISV, VisIt and ParaView, in terms of code modification and adaptability. We conducted this study around a particular scenario of a rectilinear mesh with temperature values. This scenario, already used in Section 4.1, will be applied in Section 4.3 to the CM1 atmospheric simulation, and is characteristic of a climate simulation handling a 3D *temperature* array of double precision values. This array represents the temperature at the vertices of a rectilinear mesh. The coordinates of the vertices are given by three arrays *x3d*, *y3d* and *z3d* of respective extents w, h and d.

	VisIt	Damaris	
Simulation	C	C	XML
curve.c	144 lines	8 lines	32 lines
mesh.c	167 lines	10 lines	46 lines
var.c	271 lines	15 lines	63 lines
point.c	161 lines	9 lines	33 lines
blocks.c	188 lines	10 lines	45 lines
life.c	305 lines	10 lines	40 lines

Table 4.1: Code modifications of different VisIt examples. Damaris requires code modifications and an external XML file.

4.2.1 Data Access Code for In Situ Visualization

Damaris vs. VisIt

VisIt was described in Chapter 2 as one of the major software for parallel scientific visualization. It offers in situ visualization capabilities through the *libsim* [138] library. This library allows the simulation to act as a parallel rendering engine when receiving commands from a VisIt client. Visualization tasks can also be scripted to run without user intervention. VisIt works directly on the data provided by the simulation without making a copy. The "Getting data into VisIt" manual [74] provides a complete documentation on how to instrument a simulation. This instrumentation requires to restructure the simulation's main loop in order to periodically check for pending visualization requests from the user, using the `VisItDetectInput(...)` function. When a connection is started with a client, the simulation has to provide callback functions to the *libsim* library. These callbacks access data, metadata and issue commands. In our example, two callback functions are provided in addition to the callback functions required for metadata access and response to commands. Listing 4.4 presents an overview of these data access functions.

In contrast, the code modifications required by Damaris boil down to the few lines presented in Listing 4.2 in previous section.

In addition to our previous example and to quantify more precisely the modification costs in VisIt and in Damaris, we rewrote the examples provided in VisIt's source to work with Damaris. Table 4.1 summarizes the number of lines of code required to instrument these examples with the two frameworks.[1] We removed all comments and blank lines in order to count only the lines of code relevant to the simulation/visualization coupling. Note that all of these examples except the last are serial. The last one, *life.c*, requires further modifications with VisIt to provide callback functions for collective communications. All these codes (including the unmodified ones from VisIt) are available in the Damaris release.[2]

These number of lines clearly show that Damaris greatly simplifies the code modifications required to couple existing simulations with visualization tools, thus helping users adopt in situ visualization as an alternative to offline visualization.

[1] These numbers might differ from the paper in which the results presented in this chapter were published, as a newer version of Damaris with a slightly different API was used here.

[2] See http://damaris.gforge.inria.fr.

```
1  // This function is called to retrieve the mesh
2  visit_handle get_mesh_data(int domain,
3      const char *name, void *cbdata) {
4    visit_handle h = VISIT_INVALID_HANDLE;
5    if(strcmp(name, "my_mesh") == 0) {
6      if(VisIt_RectilinearMesh_alloc(&h) == VISIT_OKAY) {
7        visit_handle hxc, hyc, hzc;
8        VisIt_VariableData_alloc(&hxc);
9        // ... idem for hyc and hzc
10       VisIt_VariableData_setDataF(hxc, VISIT_OWNER_SIM, 1, NX, mesh_x);
11       // ... idem for hyc and hzc
12       VisIt_RectilinearMesh_setCoordsXYZ(h,hxc,hyc,hzc);
13       }
14     }
15     return h;
16   }
17 }
18
19 // This function is called to retrieve the data
20 visit_handle get_variable_data(int domain, const char *name, void *cbdata) {
21   visit_handle h = VISIT_INVALID_HANDLE;
22   if(strcmp(name, "temperature") == 0) {
23     if(VisIt_VariableData_alloc(&h) == VISIT_OKAY) {
24       int size = NX*NY*NZ;
25       VisIt_VariableData_setDataD(h, VISIT_OWNER_SIM, 1, size, temp);
26     }
27   }
28   return h;
29 }
30
31 // When a VisIt client connects, the callback functions has to be provided using
32 VisItSetGetMesh(get_mesh_data,NULL);
33 VisItSetGetVariable(get_variable_data,NULL);
```

Listing 4.4: Data access functions for our sample application using VisIt. The first function retrieves the mesh coordinates, while the second retrieves the temperature field. The two last lines register the two functions as callbacks handling data accesses. This sample code does not show the modifications to the simulation's main loop.

Damaris vs. ParaView

Like VisIt, ParaView is based on VTK. The ParaView in situ interface, called *Catalyst*[3] or *co-processing library* [34] integrates a visualization pipeline (written in C++ or in Python) into the simulation. The simulation periodically feeds this predefined pipeline with data in order to produce visualization outputs, for example images.

While VisIt's *libsim* is based on callback functions and works in C, C++ and Fortran, Catalyst requires the simulation to wrap its data into VTK C++ objects. One solution consists of writing C++ classes that inherit from the right VTK objects such as `vtkDoubleArray`. An-

[3] See http://catalyst.paraview.org/.

other is to write functions that wrap the original data by creating instances of existing VTK classes. This solution is especially more appropriate when it comes to instrumenting C or Fortran simulations. Indeed ParaView does not provide any C or Fortran binding, leaving the developer with the difficult task of bridging the languages. Listing 4.5 summarizes the main steps in creating the right VTK objects for our sample application.

```
1  // Create the variable data
2  vtkDataArray* wrapMyData(...)
3  {
4    vtkDoubleArray* myArray = vtkDoubleArray::New();
5    myArray->SetName("temperature");
6    vtkIdType size = NX*NY*NZ;
7    myArray->SetArray(temp, size, 1);
8    return myArray;
9  }
10
11 // This function is called to retrieve the mesh
12 vtkObject* wrapMeshData(...)
13 {
14   // creates the necessary coordinate arrays
15   vtkFloatArray* xCoords, yCoords, zCoords;
16   xCoords = vtkFloatArray::New();
17   xCoords->setArray(mesh_x,PTX,1);
18   // ... idem for yCoords and zCoords
19   vtkRectilinearGrid *grid = vtkRectilinearGrid::New();
20   grid->setDimensions(NX,NY,NZ);
21   grid->setXCoordinates(xCoords);
22   // ... idem for Y and Z coordinates
23   vtkDataArray* array = wrapMyData(); // see above
24   grid->GetPointData()->AddArray(array);
25   array->Delete();
26   return (vtkObject*)grid;
27 }
```

Listing 4.5: Data access functions for our sample application using ParaView. The first function wraps the temperature field into the VTK object which is used by the second function that adds information related to the mesh coordinates. This code does not show all the additional codes required to initialize the visualization pipeline.

The advantage of an *a priori* definition of the visualization pipeline in ParaView is the possibility to start a simulation and be able to periodically check the generated images. The downside is the lack of interactivity and flexibility at run time of the visualization tasks. Note also that part of the ParaView pipeline can be relocated to a visualization cluster. This case is out of the scope of our work.

Other visualization software such as ezViz have a C or C++ API that can be used to perform in situ visualization in a way similar to ParaView and VisIt.

A first attempt to provide in situ visualization through VTK objects was done with the Nek5000 [98] simulation (later used in our experiments). The VTK code was made of 600 lines of C++ code, that we reduced to 20 lines of Fortran with Damaris, along with 60 lines

of XML, for the same visual result using VisIt as a backend.

4.2.2 The Case of Enzo and YT

Finally, we studied how Damaris would compare to a simulation that already uses in situ visualization. We choose the Enzo [97] code for this purpose.

Enzo is a well known astrophysical simulation based on adaptive mesh refinement (AMR). The particular needs of the Enzo community in terms of visualization led to the development of the YT [133] package, a Python library working on top of Matplotlib and supporting most visualization scenarios of AMR simulations. YT was originally designed to work as an offline visualization package, fed with Enzo's output files. Yet in recent versions of Enzo, interesting developments have been made towards in situ visualization capabilities.

The current version of Enzo[4] periodically wraps its data and metadata hierarchy into Python structures, in particular NumPy arrays. It then calls a user-provided Python script from which these information can be accessed for in situ analysis purpose. Wrapping Enzo structures in Python represents about 800 lines of C++ code spread in five different files, where Damaris would require less than 100 lines. This number is based on manually counting the number of variables that Enzo exposes to Python, knowing that Damaris would require at most two lines of code per variable. Some of these variables are however cosmological or structural constants that can be supplied directly within the configuration file. The actual number of lines required should thus be even lower.

The aforementioned in situ visualization scenario is specific to Enzo and uses its YT package. Yet any simulation developer could use these techniques to offer in situ analysis capabilities to his application. Damaris alleviates the task of building the C/Python interface.

4.3 Experimental Evaluation

In this section, we evaluate our Damaris/Viz framework with respect to performance impact and scalability. We use VisIt version 2.5.2 for visualization along with two real-life simulations: the CM1 atmospheric simulation, and the Nek5000 computational fluid dynamic (CFD) solver. We use both the time-partitioning and space-partitioning mode implemented in Damaris to compare their performance.

4.3.1 Experiments with the CM1 Simulation

CM1 was already described in Chapter 3. Its data layout corresponds to the sample code we have considered in previous sections, that is, a rectilinear 3D mesh on which variables are mapped.

The experiments were done on the Blue Waters supercomputer, NCSA's Cray XE6 Petascale supercomputer [5]. Our goal is to show that ISV approaches depend on the scalability of the rendering algorithm being used. We therefore complete a strong-scaling evaluation of two rendering methods described bellow.

[4] Version 2.1 as we write this thesis.

Using VisIt for 2D and 3D Rendering

Some of CM1's visualization scenarios including 2D and 3D rendering of various fields. Two-dimensional visualization in CM1 consists of slicing 3D fields horizontally, and converting real values into pixels using colormaps, isocontours or quiver maps. Some examples of such fields to be visualized include potential temperature (th) on the ground ($z = 0$), horizontal wind velocity (u and v) and vertical wind velocity (w) at different altitudes, or reflectivity dbz (as exemplified in Figure 4.3 (c)). Examples of 3D rendering in CM1 include volume rendering of the reflectivity dbz (as exemplified in Figure 4.3 (a) and (b)), or wind velocity (u, v and w). These tasks are available in VisIt and can be made interactive with our modification of CM1 with Damaris/Viz.

In our experiments, we focus on two scenarios of 3D rendering.

- Ray casting[5] on the dbz field (image shown in Figure 4.3 (a)).
- 10-level isosurface rendering of this same field (which corresponds to Figure 4.3 (b)).

Methodology

CM1 requires a long run time before an interesting atmospheric phenomenon appears, and such a phenomenon may not appear at small scale. Yet, we need visualizable phenomena to appear in order to evaluate the performance of in situ visualization tasks. Thus we first ran CM1 with the help of atmospheric scientists to produce relevant data. We generated a representative dataset of $3840 \times 3840 \times 400$ points spanning several iterations.

We then extracted the I/O kernel from the CM1 code and built a program that replays its behavior at a given scale and with a given resolution by reloading, redistributing and interpolating the precomputed data. The I/O kernel, identical to the I/O part of the simulation, calls Damaris/Viz functions to transfer the data to Damaris. Damaris/Viz then performs in situ visualization, either in a time-partitioning or in a space-partitioning manner.

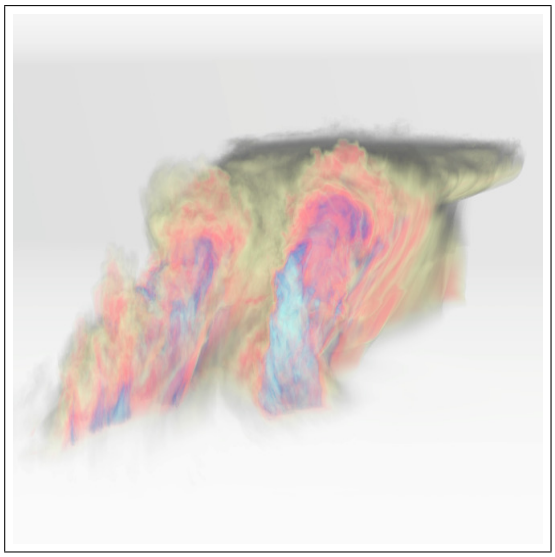
Time Partitioning vs. Space Partitioning: Results

We measured the time to complete a rendering (average of 15 iterations) using time partitioning and space partitioning for each scenario. The comparative results are reported in Figure 4.4.

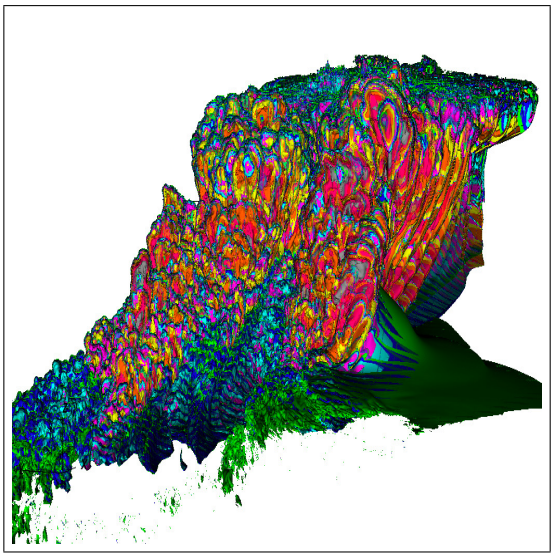
The isosurface algorithm scales well with the number of cores using both in situ approaches. A time-partitioning approach would thus be appropriate if the user does not need to hide the run time impact of in situ visualization. However, on 6400 cores, it takes as much time to complete the rendering as on 400 dedicated cores. In terms of pure computational power, a space-partitioning approach is thus 16 times more efficient.

The ray-casting algorithm on the other hand has a poorer scalability. After decreasing, the rendering time goes up again at a 6400-core scale, and it becomes about twice more efficient to use a reduced number of dedicated cores to complete this same rendering.

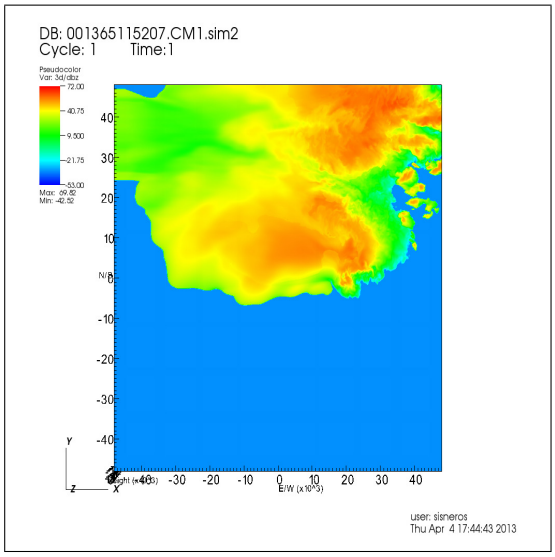
[5]Ray casting compositing (Sobel gradients, rasterization sampling, 2500 samples per ray).



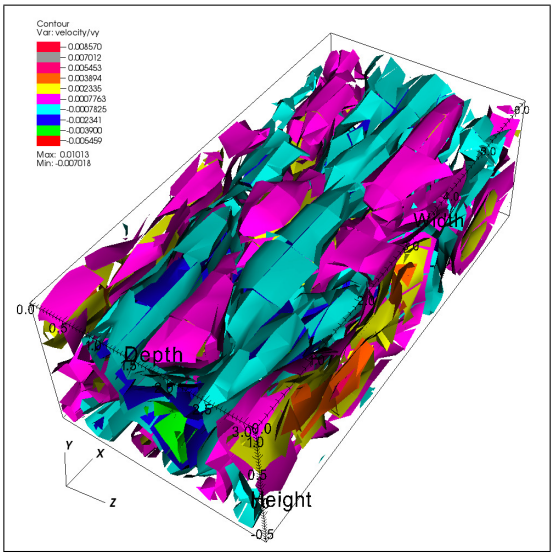
(a) CM1 Ray Casting



(b) CM1 Isosurface



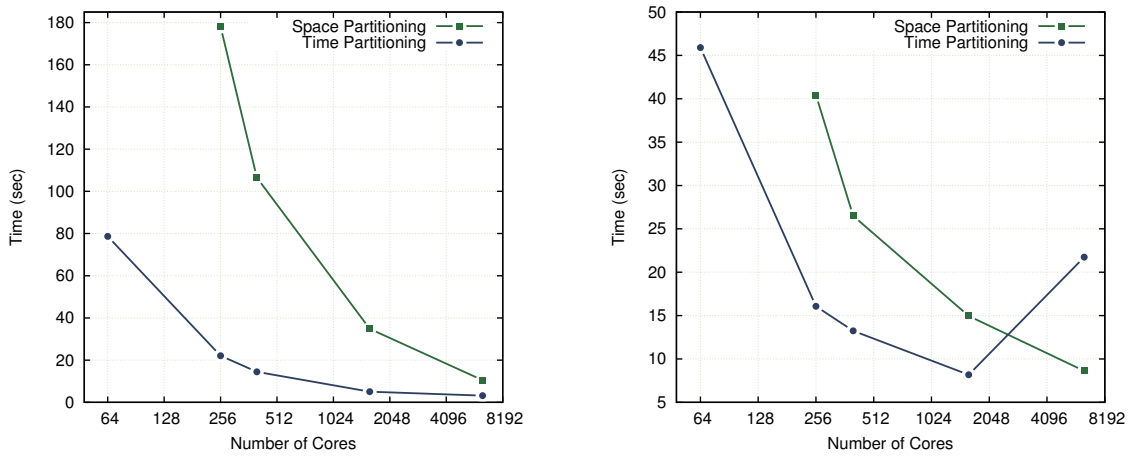
(c) CM1 Color Map



(d) Nek5000 Isosurface

Figure 4.3: Example results obtained in situ with Damaris: (a) Ray-casting of the *dbz* variable on 6400 cores (Blue Waters). (b) 10-level isosurface of the DBZ variable on 6400 cores (Blue Waters). (c) Color map of the DBZ variable on 256 cores (Blue Waters). (d) Ten-level isosurface of the y velocity field in the TurbChannel configuration of Nek5000.

Discussion: The choice of using a space-partitioning versus a time-partitioning ISV approach depends on (1) the intended visualization scenario, (2) the scale of the experiments and (3) the intended frequency of visual output. Our experiments indeed show that at small scale, the performance of rendering algorithms are good enough to be executed in a time-partitioning manner, provided that the user is ready to increase the run time of his simulation. At large scale however, it becomes more efficient to use a space-partitioning approach, especially when using ray-casting, where the observed rendering performance is substan-



(a) In Situ Isosurface (b) In Situ Ray Casting

Figure 4.4: Rendering time using ray-casting and isosurfaces, with time-partitioning and space-partitioning with CM1. Note that the number of cores represents the total number used for the experiments; using a space-partitioning approach, 1/16 of this total number is effectively used for in situ visualization, which explains the overall higher rendering time in space-partitioning.

tially better when using a reduced number of processes.

4.3.2 Experiments with the Nek5000 Simulation

Our goal in this series of experiments is to show the impact of in situ visualization tasks on the run-time variability of the simulation, and to show how space partitioning in Damaris helps alleviating this variability. We show in particular the effect of interactivity on this variability. We use the Nek5000 [98] code for this purpose.

Nek5000 is a computational fluid dynamics solver based on the spectral element method. It is actively developed at ANL's Mathematics and Computer Science Division. It is written in Fortran 77 and solves its governing equations on an unstructured mesh. This mesh consists of multiple elements distributed across processes; each element is a small curvilinear mesh. Each point of the mesh carries the three components of the fluid's local velocity. We chose Nek5000 for this particular meshing structure, different from CM1, and for the fact that it is substantially more memory-hungry than CM1. We modified Nek5000 in order to pass the mesh elements and velocity data to Damaris/Viz and we used VisIt for visualization.

Configurations

Nek5000 takes as input the mesh on which to solve the equations, along with initial conditions. We call this set a *configuration*. We used two configurations:

- the *TurbChannel* experiment, which runs well on 32 to 64 cores;
- the *MATiS* experiment, which was designed for 512 to 2048 cores.

We used the first to assess the impact of interactivity on run time with a time-partitioning and a space-partitioning approach. Figure 4.3 (d) shows the result of a 10-level isosurface rendering of the fluid velocity along the y axis, with the TurbChannel case. We then used the second configuration to show the scalability of our approach based on Damaris against a standard time-partitioning approach.

Experiments with the TurbChannel Configuration

Experiments were carried out on the Reims *stremi* cluster of the French Grid'5000 testbed, which features 40 nodes (HP ProLiant DL165 G7) with 24 cores per node, connected through a 1G Ethernet network.

To assess the impact of in situ visualization on the run time, we run TurbChannel on 48 cores using the two approaches: first we use a time-partitioning mode, in which all 48 cores are used by the simulation and synchronously perform ISV. Then we use a space-partitioning mode with Damaris/Viz, in which the simulation uses 46 cores while 2 cores asynchronously run the ISV tasks.

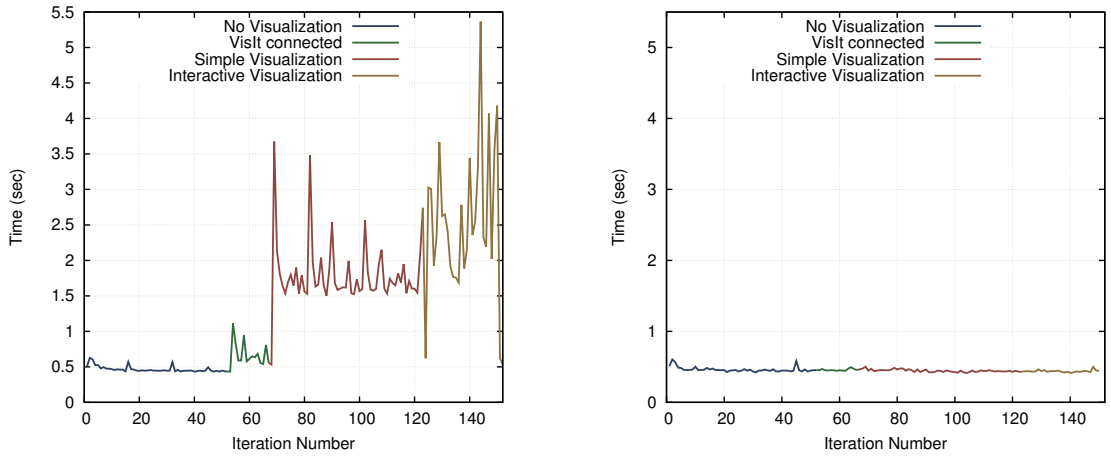
In each case, we consider four scenarios:

1. The simulation runs without visualization;
2. A user connects VisIt to the simulation but does not ask for any output;
3. The user asks for isosurfaces of the velocity fields but does not interact with VisIt any further (letting the Damaris/Viz update the output after each iteration);
4. The user has heavy interactions with the simulations (for example rendering different variables, using different algorithms, zooming on particular domains, changing the resolution).

Results Discussion: Figure 4.5 presents a trace of the duration of each iteration during the four aforementioned scenarios using the two approaches. Figure 4.5 (a) shows that ISV using a time-partitioning approach has a large impact on the simulation run time, even when no interaction is performed. The simple fact of connecting VisIt without rendering anything forces the simulation to at least update metadata at each iteration, which takes time. Figure 4.5 (b) shows that space-partitioning ISV, on the other hand, is completely transparent from the point of view of the simulation.

Experiments with the MATiS Configuration

The MATiS configuration requires a larger scale; we ran it on 816 cores. Each iteration takes approximately one minute and due to the huge number of points that the mesh contains, it is difficult to perform interactive visualization. We therefore connect VisIt and simply query for a 3D pseudo-color plot of the vx variable that is then continuously updated. For the following results, the time-partitioning approach outputs one image every time step, while the space partitioning method adapted the output frequency to one image every 25 time steps.



(a) Time-Partitioning (b) Space-Partitioning

Figure 4.5: Variability in run time induced by different scenarios of in situ interactive visualization.

Iteration Time		Average	Std. dev.
Time Partitioning	w/o vis.	75.07 sec	22,93
	with vis.	205.21 sec	57.15
Space Partitioning	w/o vis.	67.76 sec	20.09
	with vis.	64.79 sec	20.44

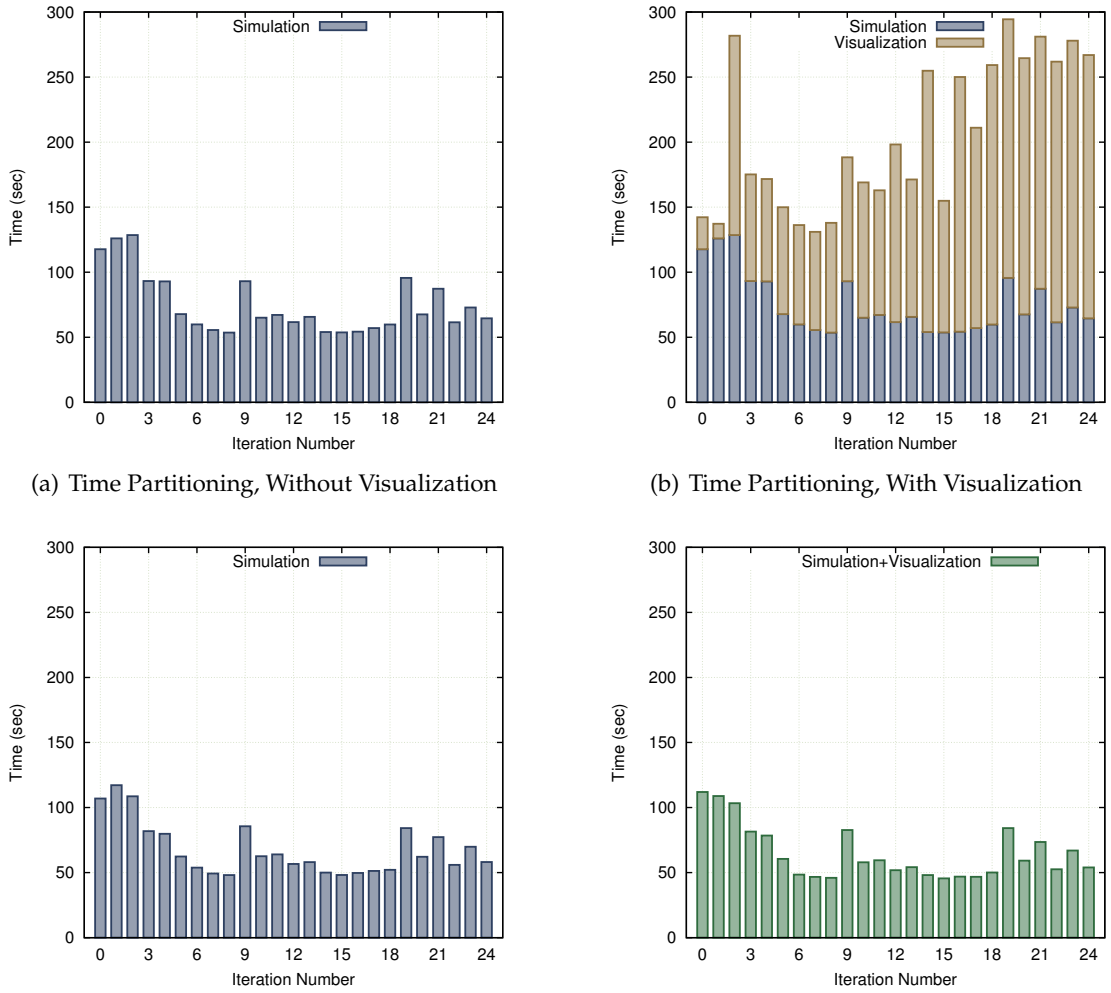
Table 4.2: Average iteration time of the Nek5000 MATiS configuration with time-partitioning and space-partitioning approaches, with and without visualization.

Results Discussion: Figure 4.6 reports the behavior of the application with and without visualization performed, and with and without dedicated cores. Corresponding statistics are presented in Table 4.2.

Time-partitioning visualization not only increases the average run time but also increases the standard deviation of this run time, making it more unpredictable. On the other hand, the space-partitioning approach yields more consistent results. One might expect a space-partitioning approach to interfere with the simulation as it performs intensive communications while the simulation runs. However, in practice we observe very little run time variation.

We also remark that decreasing the number of cores used by the simulation actually decreases its run time. This is due to the fact that Nek5000 reaches its limit of scalability. Yet due to its memory requirements, it is still necessary to run it on this number of nodes. In other words, as reducing the number of cores per node actually used by the simulation increases its performance, it further motivates the use of these spare cores for extra tasks such as visualization.

Finally, while the time-partitioning approach performs visualization at every time step here, the space-partitioning approach has adapted the frequency of its output to 1 image every 25 time steps. If a time-partitioning approach were to only output 1 image every 25 time steps (which corresponds to having only the 25^{th} iteration being impacted in Figure 4.6 (b)), the completion time for 25 time steps would be 2007 seconds on average. With space-partitioning in Damaris/Viz, this takes 1620 seconds, that is, a 20% speedup. Furthermore,



(a) Time Partitioning, Without Visualization

(b) Time Partitioning, With Visualization

(c) Space Partitioning, Without Visualization

(d) Space Partitioning, With Visualization

Figure 4.6: Iteration time of the MATiS configuration without visualization (left) and with visualization enabled (right). Top: with time partitioning, visualization time adds to the simulation time. Bottom: With space partitioning, visualization time entirely overlaps with simulation time.

since space partitioning in Damaris overlaps the visualization and simulation, the total run time is unchanged with the addition of ISV.

4.4 Related Work

In this section, we discuss our contribution with respect to relevant works in the field of simulation/visualization coupling. We separate loosely-coupled from tightly-coupled ISV. We describe how each approach meets the requirements introduced in Chapter 2 and how our approach differs from them.

4.4.1 Loosely-Coupled Visualization Strategies

Ellsworth et al. [31] propose to use distributed shared memory (DSM) to avoid writing files when performing concurrent visualization. Such an approach has the advantage of decoupling the simulation and visualization processes, but reading data from the memory of the simulation's processors can increase run time variability. The scalability of a distributed shared memory design is also a limiting factor.

Rivi et al. [111] introduce the ICARUS plugin for ParaView together with a description of VisIt and ParaView's ISV interfaces. ICARUS employs an HDF5 DSM file driver to ship data to a distributed shared memory buffer that is used as input to a ParaView pipeline. This DSM stores a view of the HDF5 files that can be concurrently accessed by the simulation and visualization tools. The HDF5 API allows to bridge the simulation and ParaView with minimum code changes (provided that the simulation already uses HDF5), but it produces multiple copies of the data and a complete transformation of data into an intermediate HDF5 representation. Also, the visualization library on the remote resource requires the original data to conform to this HDF5 representation. Damaris, on the other hand, is not based on any data format and efficiently leverages shared-memory to avoid as much as possible unnecessary copies of data. Besides, its API is simpler than that of HDF5 for simulations that do not already use HDF5.

Malakar et al. [81] present an adaptive framework for loosely-coupled visualization, in which data is sent over a network to a remote visualization cluster at a frequency that is dynamically adapted depending on resource availability. Our approach also adapts output frequency to resource usage.

The PreDatA [149] middleware proposes to dedicate a set of nodes as a staging area to perform a first step of data processing prior to I/O for the purpose of subsequent visualization. The coupling between the simulation and the staging area is done through the ADIOS [76] I/O layer. The use of the ADIOS backend allows to decouple the simulation and the visualization by simply integrating data analysis as part of an existing I/O stack [150]. While Damaris borrows the use of an XML file from ADIOS in order to simplify its API, it makes the orthogonal choice of using dedicated cores rather than dedicated nodes. Thus it avoids potentially costly data movements across nodes.

GLEAN [110] provides in situ visualization capabilities with dedicated nodes. The authors use the PHASTA simulation on the Intrepid supercomputer and ParaView for analysis and visualization on the Eureka machine. Part of the analysis in GLEAN is done in a time-partitioning manner at the simulation side, which makes it a hybrid approach involving tightly- and loosely-coupled in situ analysis. Our approach shares some of the same goals, namely to couple a simulation with run-time visualization, but we run the visualization tool on one core of the same node instead of dedicated nodes. GLEAN is also used in conjunction with ADIOS [85].

EPSN [32] is an environment providing steering and visualization capabilities to existing parallel simulations. Simulations instrumented with EPSN ship their data to a visualization pipeline running on a remote cluster, thus EPSN is an hybrid approach including both code changes and the use of additional remote resources. In contrast to EPSN, all visualization tasks using Damaris can be performed on dedicated cores, closer to the simulation, thus reducing the network overhead.

Zheng et al. [151] have provided a model to evaluate the tradeoff between in situ syn-

chronous visualization and loosely-coupled visualization through staging areas. This model can be applied to compare in situ space-partitioning using dedicated cores instead of remote resources, with the difference being that approaches utilizing dedicated cores do not have network communication overhead.

4.4.2 Tightly-Coupled In Situ Visualization

When it comes to tightly integrate analysis tasks in simulations codes, the existing solutions often do not meet all of the requirements presented in Chapter 2.

SciRun [57] is a complete computational-steering environment that includes visualization. Its in situ capabilities can be used with any simulation implemented with SciRun solvers and structures. SciRun is an example of the trend towards integrating visualization, data analysis and computational steering in the simulation process. Simulations are written specifically for use in SciRun in order to exchange data with zero data copy, but adapting an existing application to this framework can be a daunting task.

Tu et al. [132] propose an end-to-end approach for an earthquake simulation using the Hercule framework. All the components of the simulation, including visualization, run in parallel on the same machine, and the only output consists of a set of JPEG files. The data processing tasks in Hercule are still performed in a synchronous manner, and any operation initiated by a process to perform these tasks impacts the performance of the simulation.

In the context of ADIOS, CoDS (Co-located DataSpaces) [145] builds a distributed object-based data space abstraction and can use dedicated nodes (and recently dedicated cores with shared memory) with PreDatA, DataStager and DataSpace. ADIOS+CoDS has also been used for code coupling [144] and demonstrated with different simulation models. While the use of dedicated cores to accomplish two different tasks is a common theme in our approach, our objective in this chapter was to compare the performance impact on the simulation of a collocated visualization task with a directly embedded visualization. Besides, placement of data in shared memory in the aforementioned works is done through the ADIOS interface, which creates a copy of data from the simulation to the shared memory using a file-writing interface. We leverage the double-buffering technique usually implemented in simulations as an efficient alternative for sharing data.

Posteriorly to our work, Dreher and Rafin [29] built on the FlowVR framework (initially proposed for real-time interactive parallel visualization in the context of virtual reality) to provide a solution integrating both time partitioning and space partitioning using dedicated cores and dedicated nodes. They address usability by providing a simple *put*/*get* interface and a Python script that describes the various component of the visualization pipeline. They went one step further by providing in situ interactive simulation steering in a cave-like system with haptic devices [30], highlighting a case where the simulation process and research are part of the same workflow.

4.5 Conclusions and Discussion

Tightly-coupled in situ visualization appears to be a viable approach to reduce the pressure on file systems. Yet the synchronous aspect of existing solutions and their impact on the simulation's performance has limited their adoption in the HPC community.

4.5.1 Our Contribution

In this chapter we proposed Damaris/Viz, an in situ visualization framework based on the Damaris approach. By leveraging dedicated cores, external high-level structure descriptions and a simple API, our framework provides adaptable in situ visualization to existing simulations at a low instrumentation cost.

Results obtained with the Nek5000 CFD solver and the CM1 atmospheric simulation show that our framework can completely hide the performance impact of visualization tasks and the resulting run-time variability. In addition, the proposed API allows efficient memory usage through a shared-memory-based, zero-copy communication model.

4.5.2 What Remains to Study

In a 2013 report [14], the US Department of Energy (DOE) summarized the challenges of data-intensive sciences at Exascale. The authors greatly emphasize the need for in situ visualization and provide a number of reasons why ISV is still not the mainstream approach. These reasons include the software development costs and the run-time impact of ISV, two key challenges that we had foreseen when starting our research on ISV and successfully addressed with Damaris/Viz.

According to this report, another challenge of ISV, that we did not address, is *resiliency*. The increasing complexity of coupled simulations and visualization codes and the interactivity offered by some ISV frameworks indeed poses a great risk that a failure in ISV components forces an otherwise healthy running simulation to crash. This challenge needs to be addressed in the context of Damaris in order to push for its adoption by a larger number of users.

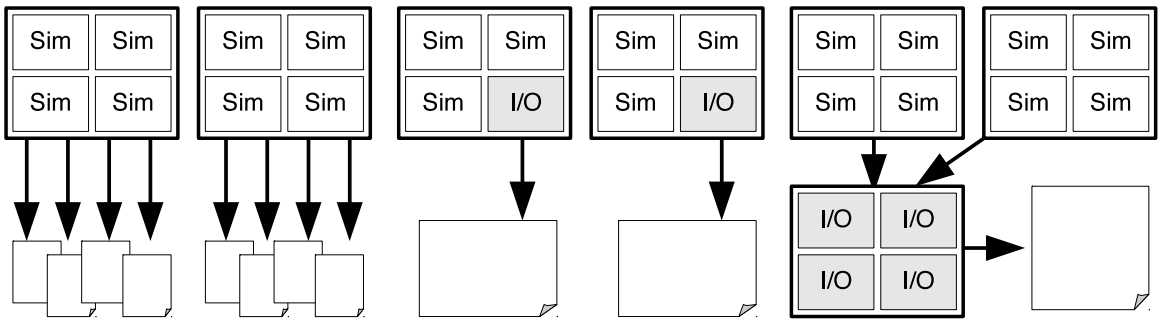
Another challenge highlighted by the DOE report is the *energy consumption* of post-Petascale machines. This challenge is presented in the context of both data storage, I/O and analytics. This motivated us to investigate the tradeoffs between performance and energy consumption in the next chapter.

Chapter 5

Energy and Performance Tradeoffs in Data Management Approaches

Contents

OVER the past few years, energy has become a growing concern in the HPC community. While hardware optimizations allow for a lower energy consumption, a large fraction of the energy consumed by a supercomputer when running an HPC application can be spared through better software design. In particular, we have seen in previous chapters that data management can introduce a substantial performance variability that, in



(a) Time Partitioning (b) Dedicated Cores (c) Dedicated Nodes

Figure 5.1: Three approaches to I/O for HPC applications.

turn, leads to a suboptimal use of computing resources and thus, higher energy consumption.

With the rise of new I/O approaches leveraging dedicated cores or nodes, it becomes important to understand their impact not only on performance, but also on the energy consumption. This is precisely the problem we address in this chapter. We first build on the Damaris middleware to provide the possibility to use dedicated nodes rather than dedicated cores. We use three approaches implemented in Damaris, namely time partitioning, dedicated cores and dedicated nodes, with the CM1 atmospheric simulation [8, 7] on the Grid'5000 [53] testbed and bring out tradeoffs between performance and energy consumption. Considering that choosing the most energy-efficient approach for a particular simulation on a particular machine can be a daunting task, we provide a model to estimate the energy consumption of a simulation under different I/O approaches. Our model is validated experimentally using the CM1 simulation on Grid'5000.

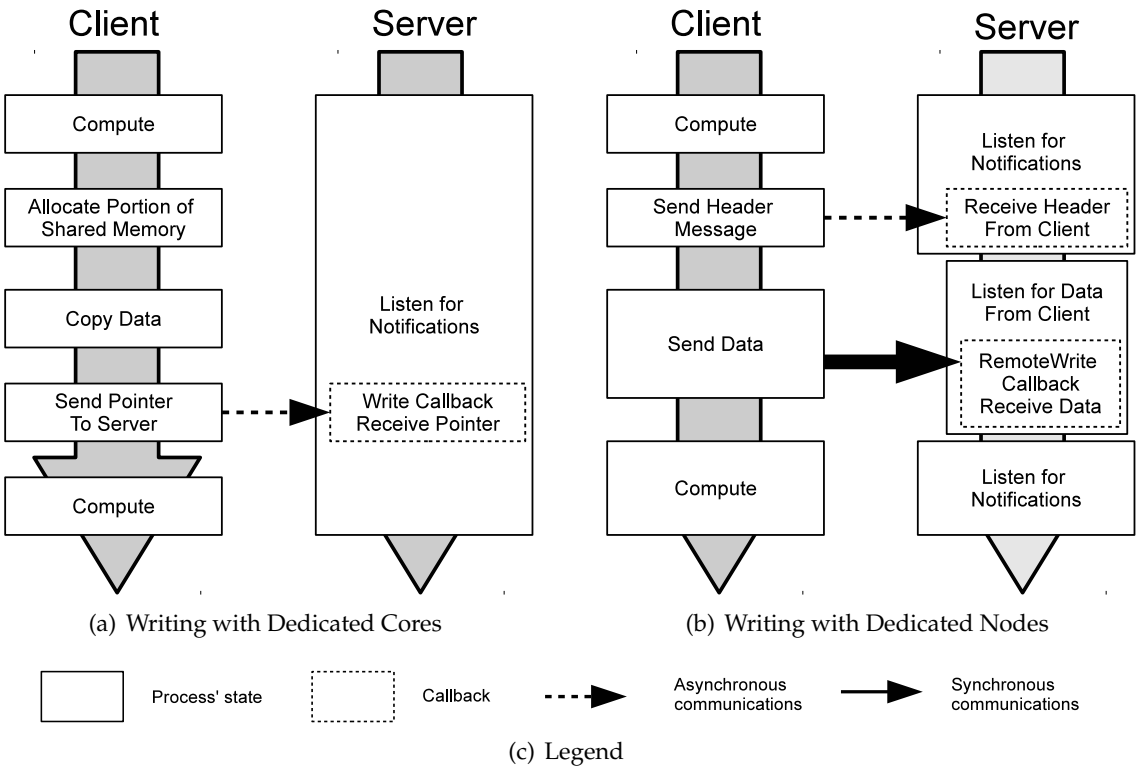
5.1 All-in-One: a Third I/O Approach in Damaris

5.1.1 Three I/O Approaches

The mismatch between computation performance and the performance of storage systems in recent supercomputers has led to the development of various novel approaches to data management. While most simulations still use a time-partitioning approach, where the simulation periodically stops to perform I/O, we have presented an alternative approach in Chapter 3, which consists of dedicating cores in multicore nodes. A third approach consists of using dedicated nodes. This approach is sometimes called "staging area" [149] or "forwarding layer" [3]. Figure 5.1 summarizes the architecture of these three approaches.

5.1.2 From Dedicated Cores to Dedicating Nodes

Damaris was initially proposed to enable dedicated I/O cores in HPC simulations. It was later extended to support time partitioning. The time-partitioning mode was used in Chapter 4 in the context of in situ visualization. It will be used here for I/O tasks. In order to



(a) Writing with Dedicated Cores

(b) Writing with Dedicated Nodes

(c) Legend

Figure 5.2: Data transfer protocols using dedicated cores and dedicated nodes.

evaluate the energy/performance tradeoff of all three approaches presented in Section 5.1.1, we implemented a third approach, based on dedicated nodes, within Damaris as well.

Implementation in Damaris

The implementation of dedicated nodes in Damaris relies on its Distributed Reactor, described in Chapter 3. Each simulation core is associated with a server running in a dedicated node. A dedicated node hosts one server on each of its cores. Different simulation cores may thus interact with the same dedicated I/O node, but with a different core in this node.

The protocol used to send data from the simulation to dedicated nodes is shown in Figure 5.2 (b). As a comparison, the protocol used by dedicated cores is shown in Figure 5.2 (a).

When a client calls `damaris_write`, it first sends an event to the Reactor of its associated server. This event triggers a `RemoteWrite` callback in the server. When the server enters this callback, it starts a blocking `receive` to get the data sent by the client. The client sends its data to the server, along with metadata information such as the *id* of the variable to which the data belongs. A small buffer is maintained in clients to allow these transfers to be non-blocking. When the client calls `damaris_write`, it copies the data into this buffer and issues a non-blocking `send` to the server using the copied data. The status of this operation is checked in later calls to the Damaris API and the buffer is freed when the transfer is completed.

This design is different and certainly less efficient than other solutions based on RDMA (remote direct memory access) such as DART [21]. Yet it is sufficient to evaluate the dif-

ferences with dedicated cores and time partitioning in terms of their energy/performance tradeoffs. Besides, the flexibility of our design, along with the recent addition of dynamic RDMA windows in the MPI 3 standard, would ease such an RDMA-based implementation in Damaris in a near future.

"Switching Gears"

The user can easily switch between each mode thanks to the configuration file, without even recompiling the application.

- `<dedicated cores="n" nodes="0"/>` enables *n* dedicated cores per node. This number of dedicated cores must divide the number of cores per node.
- `<dedicated cores="0" nodes="n"/>` enables *n* dedicated nodes. The number of dedicated nodes must divide the total number of nodes.
- `<dedicated cores="0" nodes="0"/>` disables dedicated cores and nodes. It triggers the time-partitioning mode.

This configuration would allow for a hybrid approach that uses both dedicated cores and dedicated nodes. However this approach is not supported by Damaris yet, as we haven't found any real-life scenario that would benefit from it.

The implementation of all three approaches within the same framework allows us to evaluate their respective energy consumption and performance in the next sections.

5.2 Experimental Insight into the Energy/Performance Tradeoff

In this section, we experimentally highlight the existence of a tradeoff between performance (i.e., run time) and energy consumption when using the different I/O approaches described earlier. First we present the methodology used to carry out our experiments. We then provide a detailed analysis of key results.

5.2.1 Methodology

Platforms

We run our experiments on the Rennes and Nancy sites of the Grid'5000 testbed. Contrary to Petascale platforms such as Kraken or Blue Waters, used in the previous chapters, Grid'5000 includes several clusters equipped with hardware for measuring the energy consumption.

On the Nancy site: we use the *graphene* cluster. Each node of this cluster consists of a 4-core Intel Xeon 2.53 GHz CPU with 16 GB of RAM. Intra-cluster communication is done through a 1G Ethernet network. A 20G InfiniBand network is used between these nodes and the PVFS file system deployed on 6 I/O servers. 40 nodes of the cluster are equipped with power monitoring hardware consisting of 2 Power Distribution Units (EATON PDUs), each hosting 20 outlets mapped to a specific node.

On the Rennes site: we use the *parapluie* cluster. Each node of this cluster has two 12-core AMD 1.7 GHz CPU with 48 GB of RAM. The nodes communicate with one another through a 1G Ethernet network and with a PVFS file system deployed on 3 I/O servers across a 20G InfiniBand network. 40 nodes of this cluster are equipped with power monitoring hardware consisting of 4 EATON PDUs, each hosting 10 outlets mapped to a specific node.

We acquire coarse and fine-grained power monitoring information from PDUs using the Simple Network Management Protocol (SNMP). We measure the energy consumption with a resolution of one second.

Application and Experimental Deployment

We use the CM1 application, already extensively described in Chapters 3 and 4. We deploy CM1 on 32 nodes (128 cores) on the Nancy site. On the Rennes site, we deploy it on 16 nodes (384 cores). In both cases, we configure CM1 to complete 2520 time steps. We vary its output frequency, using 10, 20 or 30 time steps between outputs. Damaris is configured to run with CM1 in five different scenarios that cover the three I/O approaches considered: time partitioning (abbreviated TP), dedicated cores (one or two – DC(ONE) and DC(TWO)), and dedicated nodes using a ratio of 7:1 (DN(7:1), 7 compute nodes for one dedicated node) or 15:1 (DN(15:1), 15 compute nodes for one dedicated node). DN(7:1) thus uses four dedicated nodes on the Nancy site, two on the Rennes site. DN(15:1) dedicates two nodes on the Nancy site, one on the Rennes site.

In our first set of experiments, performed on Nancy, the output frequency is set to 10 time steps. In order to understand the impact of the output frequency, we modify it to 20 and 30 in our second set of experiments. Finally to illustrate the impact of the system's architecture, the third set of experiments consists of running the first set of experiments on the Rennes site.

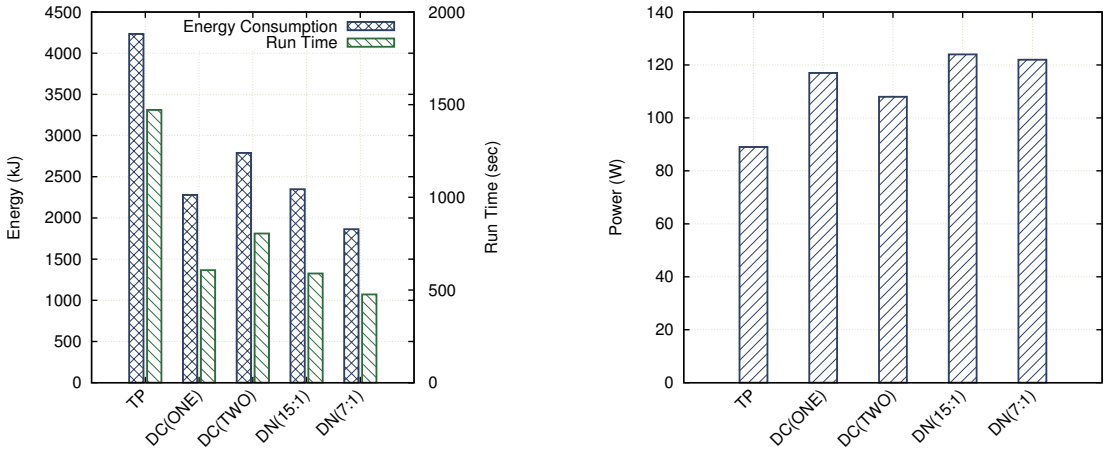
5.2.2 Experimental Results

Impact of the I/O Approach

This first set of experiments is carried out on Nancy. We aim to show the impact of the I/O approach chosen on overall performance and energy consumption.

Results Discussion: In terms of performance, Figure 5.3 (a) shows that the time partitioning approach performs poorly; an observation already made in Chapter 3. This poor performance results from the contention for the access to the storage system, which causes a high variability. As a result, all processes have to wait for the slowest one to complete its I/O before starting the next iteration. This variability also results in significant waste of energy, as idle cores remain powered on while waiting.

On the other hand, approaches that perform I/O asynchronously achieve considerably better performance. Among these approaches, *dedicating nodes with a 7:1 ratio outperforms the other configurations in terms of run time*. The larger impact of approaches based on dedicated cores on the simulation's run time can be explained by the small number of cores per node. Dedicating even one of these cores already removes a substantial fraction of the



(a) Energy vs. Completion Time

(b) Average Power Usage

Figure 5.3: Energy consumption, completion time and power usage of the different I/O approaches on the Nancy site of Grid'5000, with CM1.

computation capacity from the simulation. In approaches based on dedicated nodes, the ratio between computation nodes and I/O nodes is important: when only two nodes are dedicated (i.e., with the 15:1 ratio), a bottleneck appears in the I/O nodes because they are not able to complete their I/O in the time the simulation takes to complete 10 time steps. Therefore, the simulation has to block.

In terms of energy consumption, Figure 5.3 (a) shows a strong correlation between completion time and energy consumption. We also note that the DC(ONE) and DN(15:1) configurations exhibit very similar performance both in terms of run time and energy consumption, although one uses dedicated cores while the other leverages dedicated nodes.

Figure 5.3 (b) illustrates the power behavior of the different configurations by showing the average power usage (energy consumption divided by run time). The power usage of the time-partitioning configuration is lower than that of other approaches. This is a direct consequence of the I/O performance variability across processes: many processes indeed remain idle while waiting for the I/O phase to complete. The power usage of an idle core is lower than the power usage of a core performing computation, which leads to a lower average power usage.

Table 5.1 shows statistics on the cluster-wide energy consumption, including the minimum, maximum and standard deviation of energy consumption across nodes. We observe a higher variability in energy consumption with the dedicated-node approach (standard deviation of 4.5). This variability results from the fact that nodes running the simulation and dedicated nodes don't have the same workload to complete.

Offloading I/O to dedicated resources allows the cores running the simulation to keep performing computation without waiting, increasing their average power usage. The variability in power usage is offloaded as well to dedicated resources, as they spend most of their time idle, waiting for data to be sent by the simulation, or waiting for their I/O to complete. Therefore, we observe a similar power usage among these configurations. Since the

Table 5.1: Statistics on energy consumption with different I/O approaches, on Nancy.

Approach	Total (kJ)	Average (kJ)	Min (kJ)	Max (kJ)	Std. dev.
Time Partitioning	3324	101	94	108	3.4
Dedicated Core (ONE)	1777	54	38	56	2.9
Dedicated Cores (TWO)	2211	67	52	70	3
Dedicated Nodes (15:1)	1736	53	35	56	4.2
Dedicated Nodes (7:1)	1340	41	28	44	4.5

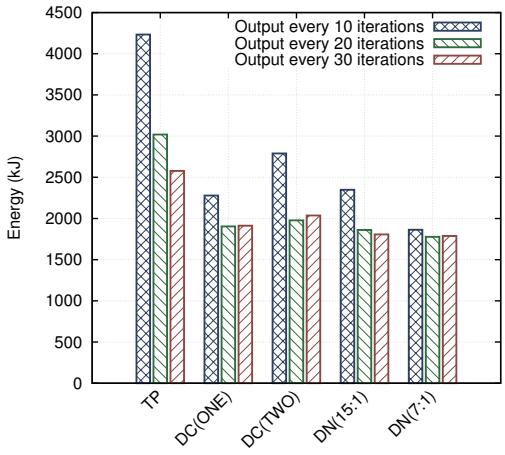


Figure 5.4: Energy consumption on Nancy with different output frequencies.

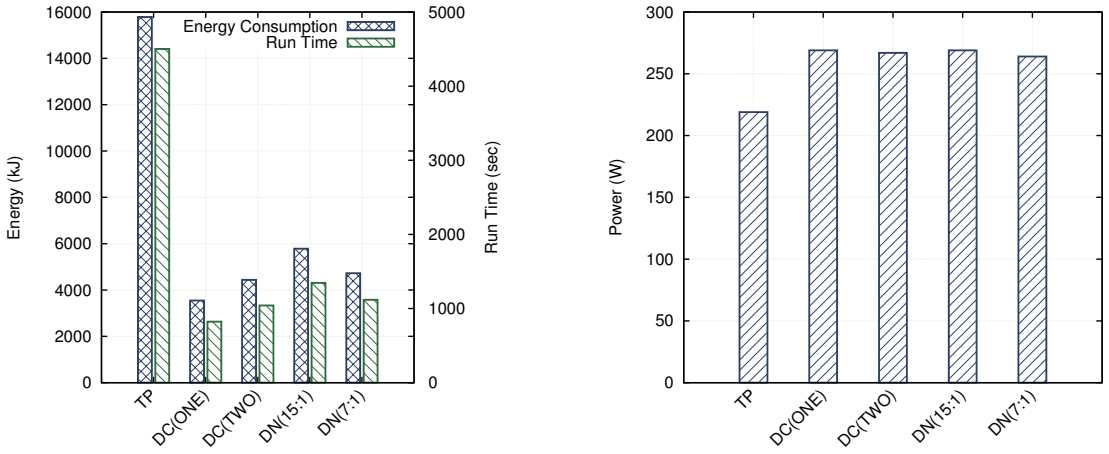
average CPU utilization affects the completion time, we observe longer execution times for lower CPU utilization.

Impact of the Output Frequency

How frequently a simulation outputs data is also a factor to consider when looking for the best I/O configuration. We therefore varied the output frequency of the CM1 application by making it output data every 10, 20 or 30 time steps.

Results Discussion: Figure 5.4 shows the total energy consumption with all five configurations with these different output frequencies. Unsurprisingly, these results show a correlation between energy consumption and output frequency with the time-partitioning approach.

In configurations that use dedicated resources, there is no clear difference between the last two output frequencies (i.e., every 20 and 30 seconds). This is due to the fact that in both cases, I/O fully overlaps with computation, thus the run time is the same and the energy consumption similar. We observe an increase in the energy consumption only when the frequency is high enough to start impacting the simulation. DN(7:1) seem to provide enough resources to sustain such a high output frequency without impact the simulation in terms of energy consumption.



(a) Energy vs. Completion Time

(b) Average Power Usage

Figure 5.5: Energy consumption, completion time and power usage of the different I/O approaches on the Rennes site of Grid'5000, with CM1.

Impact of the System's Architecture

Our last set of experiments aims to show the effect of the system's architecture on the performance and energy consumption. We therefore reproduced on the Rennes site the experiments carried out in Section 5.2.2 on the Nancy site.

Results Discussion: Figure 5.5 (a) presents the energy consumption and performance of the five configurations. This time, dedicating one core per node outperforms the other configurations both in terms of performance and energy consumption. The reason behind this result is a lower impact on the simulation when 1 core out of 24 is dedicated to I/O, compared to dedicating 1 core out of 4 on the Nancy site. Dedicating nodes seems to lower the performance of the simulation and increase its energy consumption compared to dedicating cores. We explain this result by the fact that a larger number of cores in computation nodes send their data to dedicated nodes, which leads to a higher network utilization and, thus, an increase of the simulation's run time.

Figure 5.5 (b) presents the related power usage. Again, we notice a similar behavior among approaches that leverage dedicated resources, while the average power usage of the time-partitioning configuration is much lower.

Figure 5.6 compares the energy consumption on the Nancy and Rennes sites, and Figure 5.7 depicts the performance (in number of steps per second) and average power usage during the simulation using the two system architectures. The results indicate that the comparative behavior of the different approaches with respect to performance and energy efficiency depends on the system on which they run. With a larger number of cores per node, it becomes more efficient both in terms of run time and energy consumption to dedicate some of these cores to perform I/O tasks. Platforms with a smaller number of cores per node, on the other hand, benefit from a configuration based on dedicated nodes.

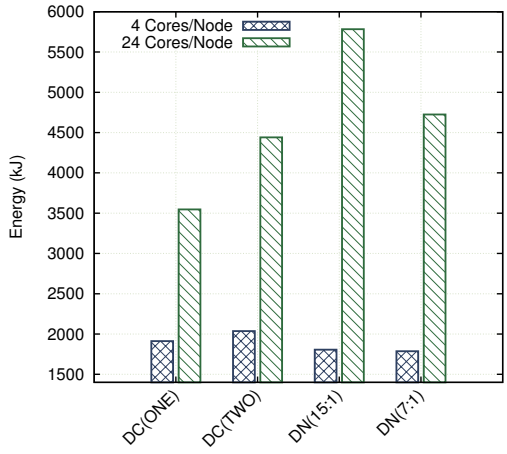
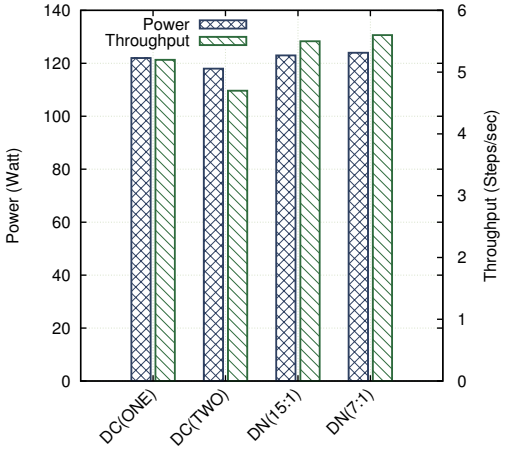
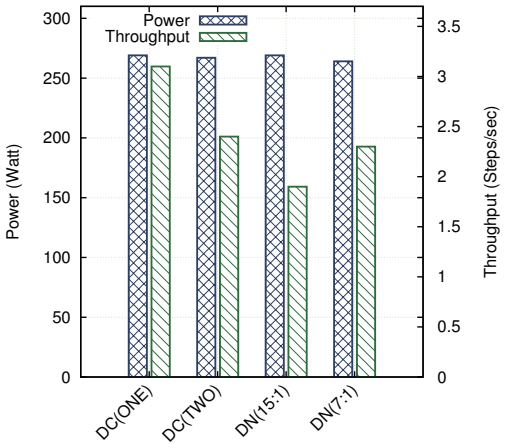


Figure 5.6: Energy consumption on Nancy and Rennes sites with different I/O approaches.



(a) Nancy (4 cores/node)



(b) Rennes (24 cores/node)

Figure 5.7: Average power usage and throughput with different I/O approaches on the Nancy and Rennes sites of Grid'5000.

Besides the number of cores per node, we can also attribute these different behaviors to the different hardware used on each site (i.e., Intel 2.53 GHz CPU on Nancy and AMD 1.7 GHz CPU on Rennes).

Summary of our Findings

Our experiments have shown that there exists a tradeoff between energy consumption and performance, and that a smaller run time does not necessarily implies a lower energy consumption. Additionally, while a time-partitioning configuration always appears to perform poorly and to consume more energy, the choice of the appropriate configuration, whether one targets lower energy consumption or smaller run time, depends on many factors, including the output frequency and the system's architecture. In the following section, our goal is precisely to provide a model of energy consumption that will help users select the

best configuration.

5.3 Model of Energy Consumption

5.3.1 Model Formulation

The model we develop aims to provide an estimation of the energy consumed by an application under different I/O approaches and configurations of these approaches. We start by assuming that any approach that uses dedicated cores or nodes perfectly hides the I/O time from the point of view of the simulation. This implies that (1) the communication cost between the simulation and dedicated nodes, or the cost of memory copies in dedicated cores, is negligible, and (2) the actual I/O time in dedicated resources is smaller than the duration of an iteration in the simulation, so the simulation does not block waiting for dedicated resources to complete their I/O. These assumptions are realistic for computation-intensive, periodic applications, that is, application whose primary task is to solve equation on in-memory data and periodically checkpoint some results, with an I/O time that is intended to be substantially smaller than the computation time. CM1, like most HPC simulations, fall into this category of applications and experimentally verifies our assumptions.

Under these assumptions, the energy consumption can be expressed as follows:

$$E = T_{sim} \times P_{sim}, \tag{5.1}$$

where T_{sim} is the total execution time of the simulation and P_{sim} is the average power consumption during this execution. Therefore, estimating E can be done by estimating the execution time T_{sim} and the average power usage P_{sim}.

Estimating the Execution Time

The execution time depends on the number of nodes (n_{node}) on which the simulation runs, the number of cores per node (n_{core}) used by the simulation, and the scalability of the application with respect to these numbers. We call $s_{core}(k)$ the scalability of the application when using k cores on a single node, and $s_{node}(k)$ the scalability of the application when deployed on k nodes. The scalability is defined as a value between 0 and 1, 1 representing a perfect scaling of the application across the specified number of cores or nodes, that is, a task distributed on twice as many resources will take half the time to complete. We call n_{iter} the number of iterations that the application executes, and T_{base} the time to complete one iteration on one single core of one node. Putting all together,

$$T_{sim} = \frac{T_{base} \times n_{iter}}{(n_{core} \times s_{core}(n_{core}))(n_{nodes} \times s_{nodes}(n_{nodes}))}. \tag{5.2}$$

This formula assumes an independence between the scalability across cores and across nodes. The scalability functions s_{core} and s_{node}, as well as T_{base}, have to be profiled by running the simulation on different numbers of cores and nodes without I/O.

Estimating the Power Usage

A node can be either computing or idle. These two states lead to two power usage metrics P_{max} and P_{idle}. Their respective values can be obtained through hardware profiling, and will be used to estimate the power consumption P_{sim} of the simulation during its run.

Since communication tasks constitute a very small part of computation-intensive applications, we consider that its related power consumption does not change significantly with the different number of nodes involved. Modeling the energy actually consumed by communications remains outside the scope of this work. The effect of communication phases on the simulation's run time is however still present in our model through the scalability functions.

When using dedicated nodes, the nodes running the simulation have a power behavior different from that of a dedicated node. We call c_n the number of nodes used by the simulation, and d_n the number of dedicated nodes. We derive:

$$P_{sim} = \frac{c_n P_{max} + \frac{1}{2}(P_{idle} + P_{max})d_n}{c_n + d_n}. \tag{5.3}$$

To simplify our model, we set the power consumption of a dedicated node to the mean between P_{max} and P_{idle}. The results obtained when validating our model in Section 5.3.3 shows that our model remains good despite this simplification. A more precise model would require to take into account the power usage when a node is performing I/O, and the ratio between I/O time and idle time in dedicated nodes.

Using dedicated cores, all nodes are used for computation. We argue that the presence of dedicated cores does not significantly changes the power consumption of the node, especially when the nodes feature a large number of cores. Therefore our model defines its power consumption as:

$$P_{sim} = P_{max}. \tag{5.4}$$

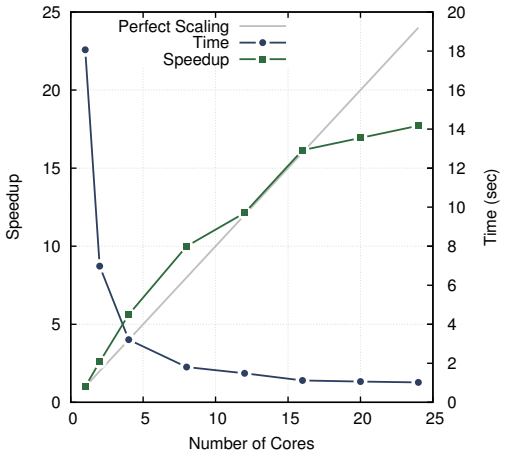
5.3.2 Application and Hardware Profiling

Our model is based on several parameters that depend on the considered application (the efficiency functions s_{core} and s_{node}), and on the hardware on which the application runs (the power usage P_{max} and P_{idle}). We perform a set of microbenchmarks to obtain these values. We provide profiling results on Grid'5000's *parapluie* cluster, described in Section 5.2.1.

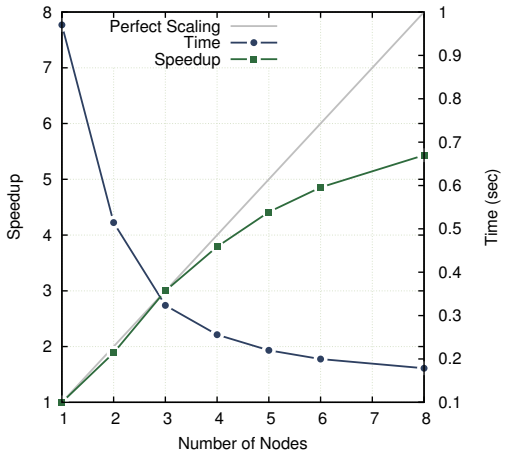
Application Scalability

To assess the scalability of CM1, we run it on different numbers of cores and nodes. Figure 5.8 (a) shows the run time and the speedup as a function of the number of cores. Figure 5.8 (b) presents the run time and the speedup as a function of the number of nodes. Similar strong-scaling tests (i.e., measured of performance with different numbers of nodes for a fixed global domain size) have been conducted by the developers of CM1.[1]

[1] http://www2.mmm.ucar.edu/people/bryan/cm1/pp.html



(a) Scalability with respect to the number of cores within a node.



(b) Scalability with respect to the number of nodes.

Figure 5.8: Scalability of CM1 with respect to the number of cores and with respect to the number of nodes. Experiments done on the Rennes site of Grid'5000.

From these runs, we obtain $T_{base} = 18.1$ seconds as the computation time with one core on one node. The efficiency functions s_{core} and s_{node} can be derived by dividing the measured speedup by the perfect speedup.

Power Usage

To profile P_{idle} and P_{max} we use the PDUs attached to some of the nodes. We retrieve the power measurements using the Simple Network Management Protocol (SNMP) with a resolution of 1 second.

Figure 5.9 (a) presents the idle power consumption of a set of 8 nodes instrumented with such PDUs. Figure 5.9 (b) shows the power consumption of the same 8 nodes when they run the simulation.

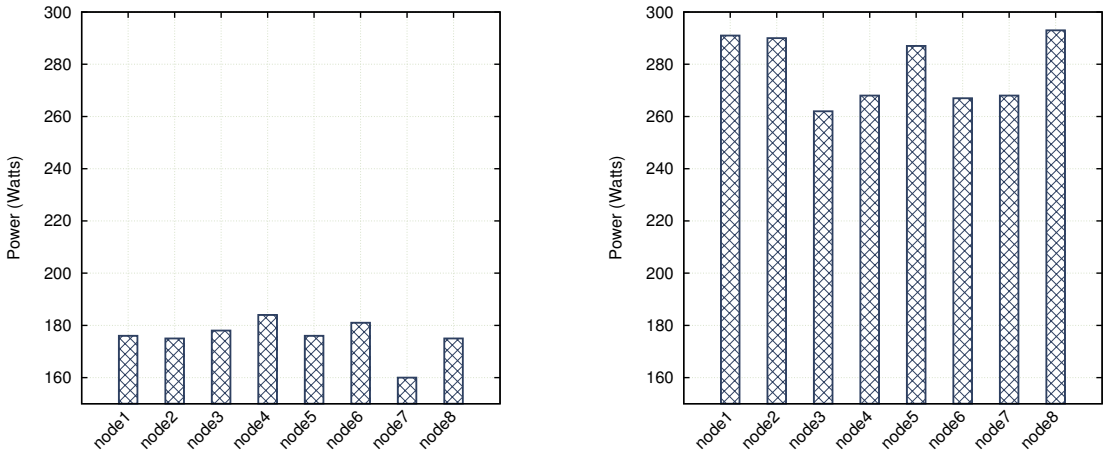
After performing these measurements, we take their average as values for P_{idle} and P_{max}. The significant variance in the power used by the nodes of the same cluster shows the importance of profiling on several nodes. A long-running HPC application can amortize the cost of our profiling approach, which is performed only once for a particular hardware.

5.3.3 Experimental Validation

Rennes Cluster

In order to validate our model, we perform a set of experiments on Rennes' *parapluie* cluster of Grid'5000. We configure CM1 to run 2520 iterations on 16 nodes (384 cores), writing data every 30 iterations. 3 other nodes are used by the PVFS file system.

Damaris uses one of four configurations: dedicating one (DC(ONE)) or two cores (DC(TWO)), and dedicating one (DN(15:1)) or two nodes (DN(7:1)). We left out the time-

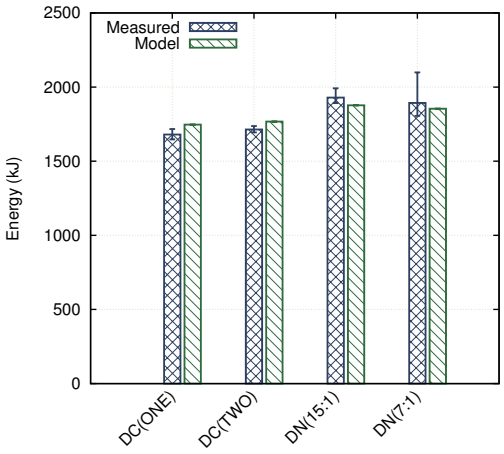


(a) Idle power usage.

(b) Average power usage during the simulation.

Figure 5.9: Power usage of the *parapluie* cluster of Grid'5000 when the nodes are idle and when they run the simulation

Figure 5.10: Observed and estimated energy consumption with different I/O approaches on the Rennes site of Grid'5000. Error bars represent minimum and maximum measured values our of five runs.



partitioning approach, as we already demonstrated that it performs worse than all other approaches.

We measure the energy consumption for each configuration. Each experiment is repeated five times. Figure 5.10 shows the results of these experiments and the estimations provided by the model. The worst relative difference we observe between our model and the experimental results is 4%, when employing one dedicated core per node (i.e., DC(ONE)). We observe a greater variability with the DN(7:1) configuration. This was due to a few iterations during which the network bandwidth of the system dropped, most probably because of contention with other users of the cluster, causing dedicated nodes to spend more time writing data than the time needed for the application to complete 30 iterations.

One major result here is that our model is able to predict the best I/O approach among the four proposed ones. In this case, dedicating one core per node appears to be the best approach in terms of energy consumption, which conforts our core intuition underlying the

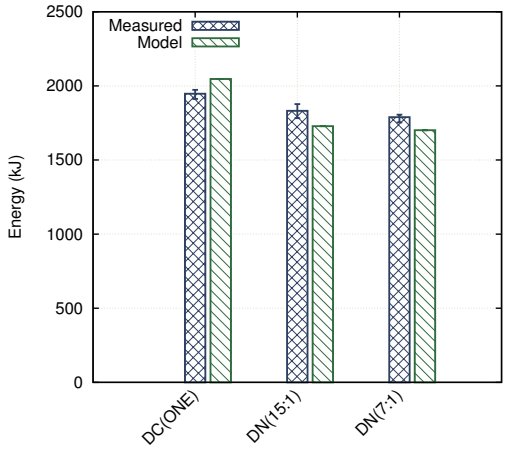


Figure 5.11: Observed and estimated energy consumption with different I/O approaches on the Nancy site of Grid'5000. Error bars represent minimum and maximum measured values our of five runs.

Site	Approach	Accuracy
Rennes	Dedicated Core (ONE)	96.0%
	Dedicated Cores (TWO)	96.9%
	Dedicated Nodes (15:1)	97.3%
	Dedicated Nodes (7:1)	98.0%
Nancy	Dedicated Core (ONE)	95.0%
	Dedicated Nodes (15:1)	94.3%
	Dedicated Nodes (7:1)	95.0%

Table 5.2: Accuracy of the model on Rennes and Nancy sites with all configurations tested.

Damaris approach, initially designed to leverage the potential benefits of using dedicated I/O cores.

Nancy Cluster

To further test our model's accuracy, we conducted a set of experiments on the *graphene* cluster of Grid'5000. We run CM1 on 32 nodes (128 cores) with 6 additional nodes for PVFS. We used the configuration of CM1 presented earlier. Damaris was configured to run with either one dedicated core per node (DC(ONE)), or one or two dedicated nodes (respectively DN(15:1) and DN(7:1)). We left out the configuration featuring two dedicated cores, as it has a large performance impact on a 4-core/node cluster (the performance of CM1 would drop by half). We employed the same type of profiling method as described in Section 5.3.2.

Each experiment is repeated five times. Figure 5.11 shows the observed energy consumption for each approach, as well as the estimations of our model. The worst relative difference observed is 5.7%, when using one dedicated node (DN(15:1) configuration). Our model is again able to predict the best configuration. In this case, the best configuration consists of using one dedicated node. Here given the small number of cores per node, dedicating one core (i.e. 1 out of 4) to I/O is not the best approach.

Table 5.2 summarizes the accuracy of our model on the two platforms and with all tested configurations.

5.4 Discussion and Related Work

5.4.1 Profiling Energy Consumption of HPC Simulations

Gamell et al. [37] provide a power model for the in situ analysis of the S3D turbulent combustion code. They investigate the roles of the system's architecture, the algorithm design and various deployment options. In their power model, they use the Byfl compiler analysis tool [99] to obtain the application's behavior. While they explore the power behaviors of the S3D code under different scenarios, they don't address the impact of these different scenarios in terms of performance.

Kamil et al. [59] extrapolate the power consumption of the machine from that of a single rack, with AC to DC conversion in mind. They use the High Performance Linpack (HPL) benchmark, which they claim to have a similar power behavior to other compute-intensive scientific workloads. However, they also indicate that HPL is not an ideal workload for performance measurements. Therefore, while they target power efficiency in their work, their contribution is limited to the exploration of power behaviors of computation-intensive workloads. Additionally, they run the benchmarks for three minutes, which from our experience is not sufficient to represent an application's power behavior.

Song et al. [121] present a power performance profiling framework, PowerPack, to study the power behavior of HPC Challenge benchmarks (HPCC). They focus on applications' memory access patterns and the impact of the system size on the energy efficiency. They find that workloads that have high temporal and spatial locality spend little time waiting for data and consume more processor power compared to other workloads. In their HPCC tests, they observe that memory is the second largest power consumer after CPU. For the energy profiling, they find that embarrassingly parallel codes achieve better energy efficiency as the size of the system increases. However, for codes that are not embarrassingly parallel, the energy consumption increases faster than the performance of the workload. With the MPI_FFT code they exemplify an energy consumption proportional to the number of node, while performance remains sub-linear. Therefore, they indicate that the size of the system is an important factor to consider along with application characteristics when trying to achieve efficient energy consumption.

5.4.2 Saving Energy

Some researchers have considered methods for saving energy on HPC machines. Orgerie et al. conducted a survey on the methods for improving the energy efficiency in large-scale systems [96]. They discuss the methods to evaluate and model the energy consumed by computing and network resources. They indicate that system energy consists of two parts: static and dynamic. The former one depends on the system size and type of components while the latter one results from the usage of the resources. They claim that we can improve the energy efficiency by minimizing the static part and by obtaining more performance in proportion to the dynamic part of the system. They model the energy consumption of computing resources, then propose several techniques to save energy, such as Dynamic Voltage Frequency Scaling (DVFS) [60, 73, 118, 137], software improvements [123] and hardware capabilities [13]. For networking resources, they find that switch fabrics are an important part of the power consumption of the network, e.g., 90% for IBM InfiniBand routers. Similarly to

computing resources, the energy consumption of networking devices is not proportional to their usage.

Laros et al. [68] present the impact of CPU frequency and network bandwidth scaling on energy efficiency. They apply static changes in the CPU frequency to save energy at the cost of performance degradations. They find that the impact of the CPU frequency scaling depends on the type of workload. While computation-intensive workloads suffer from a big degradation of their performance, the energy consumption of communication-intensive workloads can be greatly improved.

A similar work has been carried out by Springer et al. [122]. They demonstrate that significant potential exists for saving energy in HPC applications without sacrificing performance. They apply dynamic frequency scaling by shifting the gears which represent the different levels of CPU frequencies. They observe that well-tuned programs such as NAS benchmarks can benefit from their approach especially during their idle time resulting from communications. The biggest contribution of their work is to be able to switch energy gears dynamically by observing the pressure on the memory and the location of MPI calls in the program to obtain better energy efficiency.

Fault-tolerance protocols can also be subject to optimizations of their energy efficiency. Diouri et al. [20] estimate the energy usage of different fault-tolerance protocols, including protocols based on checkpoint/restart. They study the influence of various parameters such as the checkpointing interval, number of processes, message size and number, etc. as well as hardware parameters (number of cores per node, disk type, memory, etc.). They use a calibration approach that inspired our profiling approach, in order to take into consideration the specific hardware used in their energy estimations. Besides energy estimations, they also apply power saving techniques to improve energy efficiency. Contrary to our work however, they do not study different I/O approaches such as using dedicated cores or nodes, and the impact of these approaches on energy consumption.

Most of the studies on energy efficiency in HPC target the entire system's power. Gamell et al. [37] separate the network component in their model, however they make assumptions for the related component type since power information for every component (in particular the NIC) is not available. Son et al. [119] target the power usage of disks. They use the SPEC2000 floating benchmark suite and generate statistical data for performance and energy consumption via a simulator similar to DiskSim. They apply proactive disk power management and also make use of code restructuring, which results in up to 43% of energy savings compared to traditional power management. This work inspired ours by showing us that software-driven approaches can be more efficient than existing hardware solutions for energy efficiency.

5.4.3 Power Measurement Methods

The aforementioned works also differ in the methodology that they apply for power measurements. Kamil et al. [59] investigate various power measurement methodologies such as line meters, clamp meters, integrated meters and power panels, and opt for power panels in their work. Other methods can be applied, including voltage regulator models that provide current and voltage readings at node level [68], cluster specifications [37], simulators [119] and wattmeters [20]. While there is a wide range of options for measuring the power, most of them are subject to measurement errors. Therefore, power measurements are generally

multiplied to reduce the impact of the measurement error. We also follow the same trend in our work.

5.5 Conclusions

Power consumption has started to severely constrain the design of HPC systems and starts influencing software solutions as well. As the amount of data produced by large-scale simulations explodes, it becomes necessary to find solutions to I/O that are not only fast, but also energy efficient.

Our detailed study of three I/O approaches, all implemented in the Damaris framework, reveals significant differences in the performance of the CM1 application as well as its energy consumption. Three factors at least contribute to such differences. First, the adopted I/O approach and its configuration. Second, the output frequency. Third, the architecture on which the application runs, and in particular the number of cores per multicore node.

As choosing the right I/O approach to save energy for a particular scenario can be difficult, we provided a model that helps scientist estimate the energy consumption of their application on a particular platform and under different data management approaches. The accuracy of our model (96.1% on average) and its validation with the CM1 application shows that it can effectively guide the user toward the most energy-efficient configuration.

This work opens room for energy-saving approaches. In particular, dedicated resources could benefit from DVFS techniques, or be more productively used by enabling compression to reduce the amount of data and thus lower the energy consumption originating from data transfers.

Chapter 6

CALCioM: Mitigating I/O Interference through Cross-Application Coordination

Contents

As of August 2014, the top five supercomputers have all more than 500,000 cores [130]. This tremendous computational power offers the possibility to run scientific simulations at very large scale and with high accuracy. But sustained Petascale (and Exascale in a few years) is not achieved by running applications one at a time. The real power of a million-core machine comes from the increased number of applications that can run concurrently.

An important challenge at such a large scale is to deal with the data deluge coming from these applications. Chapters 3 and 4 have proposed solutions to improve I/O and visualization performance respectively, for a single application. Unfortunately, when several concurrent applications access a shared parallel file system in an uncoordinated manner, storage servers have to deal with interleaved requests coming from different sources, which often breaks the access patterns optimized by each application individually. This negatively impacts the I/O performance of these applications and increases the variability of their I/O performance. We call this particular phenomenon *cross-application I/O interference*.

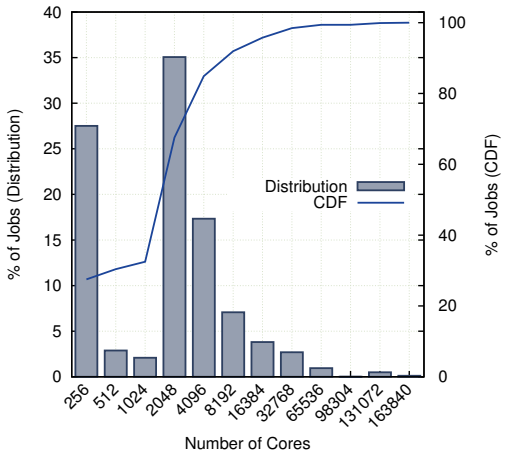
In this chapter, we propose the CALCioM approach to solve this problem. CALCioM is radically different from traditional approaches where applications are optimized individually, disregarding potential cross-application interference, and where interference-avoiding strategies are left to the file system's scheduler, with no information on the constraints or freedom of each application and no way to differentiate I/O requests. In contrast, CALCioM provides a communication layer so that applications can expose their I/O behavior and coordinate with one another in order to avoid interfering. We specifically study three coordination strategies: interfere, serialize, and interrupt accesses, which are made possible through cross-application communications. We observed that these strategies are all suboptimal in different contexts yet complement each other in a way that makes a dynamic selection desirable, especially when applications present different I/O behaviors and requirements.

6.1 I/O Interference: an Increasingly Important Issue

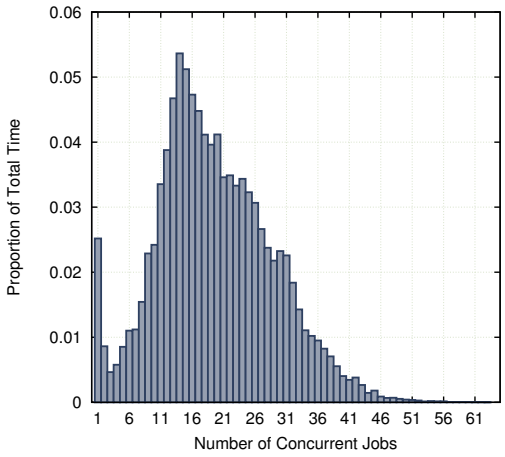
Interference can be defined as a *performance degradation observed by an application in contention with other for the access to a shared resource*. In the context of I/O for HPC, the shared resource is the parallel file system. This section tries to grasp how frequently I/O interference occurs in a system. It then provides tools to analyze I/O interference, and examples of its effect on real platforms.

6.1.1 Probability of Concurrent Accesses

Although computer scientists generally argue that their machines have been designed mainly to run applications at full scale (i.e., large applications), current machines are already used by many relatively small applications at the same time. For example, Figure 6.1 (a) shows the distribution of job sizes on Argonne's Intrepid machine. Half of the jobs on this platform indeed run on less than 2,048 cores (i.e., 1.25% of the full machine); this assertion remains true when weighting the jobs by their duration, that is, half of the machine time is used by applications smaller than 2,048 cores. Given this observation, it becomes interesting to compute the probability that at least two applications interfere with one another.



(a) Distribution of job sizes



(b) Number of concurrent jobs

Figure 6.1: Job sizes and number of concurrent jobs by time unit on Intrepid, extracted from the Parallel Workload Archive [100], using *ANL-Intrepid-2009-1.swg* (8 months of job scheduler's traces, from January 2009 to September 2009).

The number of applications that run concurrently at any given moment on a supercomputer can be denoted as a discrete random variable $X \in \mathbb{N}$. Figure 6.1 (b) shows the distribution followed by X on ANL's Intrepid, and the corresponding CDF. The proportion of time spent doing I/O by any application can also be seen as a random variable $\mu \in [0, 1]$. As a first approximation, we assume the independence between X and μ, that is, an application will not spend more time in I/O as a result of a different number of applications running concurrently. This assumption is optimistic and ignores the fact that the more applications run, the more likely they are to interfere and therefore, the higher μ becomes. But it allows us to compute a lower bound on the probability that several applications interfere.

At a given moment, we denote Y the random variable representing the number of applications currently in I/O phase. Knowing that n applications run on the machine at this moment, we can compute $\mathbb{P}(Y = k|X = n)$ the probability that k applications are concurrently in I/O phase, for $k \leq n$:

$$\mathbb{P}(Y = k|X = n) = \binom{n}{k} \mathbb{E}(\mu)^n (1 - \mathbb{E}(\mu))^{n-k} \tag{6.1}$$

Indeed each application has an independent probability $\mathbb{E}(\mu)$ to be in I/O phase, thus the number of applications in I/O phase follows a binomial distribution. We can then derive the probability $\mathbb{P}(Y > 1)$ that two or more applications are in I/O phase.

$$\begin{aligned}
\mathbb{P}(Y > 1) &= 1 - \mathbb{P}(Y = 0) - \mathbb{P}(Y = 1) \\
&= 1 - \sum_{n=0}^{+\infty} \mathbb{P}(X = n)\mathbb{P}(Y = 0|X = n) - \sum_{n=1}^{+\infty} \mathbb{P}(X = n)\mathbb{P}(Y = 1|X = n) \\
&= 1 - \sum_{n=0}^{+\infty} \mathbb{P}(X = n)(1 - \mathbb{E}(\mu))^n - \sum_{n=1}^{+\infty} \mathbb{P}(X = n) n \mathbb{E}(\mu)(1 - \mathbb{E}(\mu))^{n-1}
\end{aligned} \tag{6.2}$$

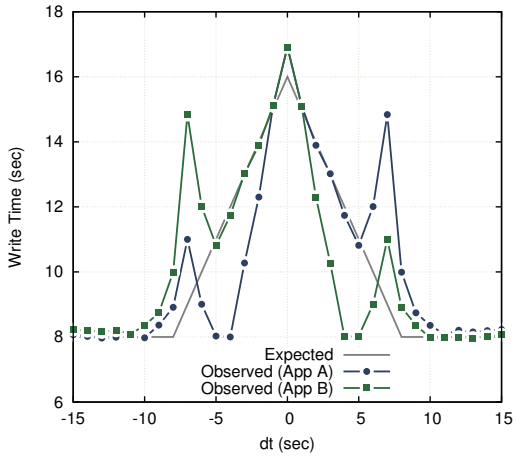


Figure 6.2: Experiments done on Grid'5000 (Nancy site) with PVFS deployed on 35 nodes; two applications of 336 processes each write 16 MB per process in a contiguous collective pattern. A (red) starts at the reference date (0), B (blue) starts at an arbitrary date dt with respected to the reference date.

As an example, assuming that the average portion of time spent in I/O by applications is as small as $\mathbb{E}(\mu) = 10\%$, and using the distribution shown in Figure 6.1 (b), the probability to observe concurrent accesses as computed using Equation 6.2 is $\mathbb{P}(Y > 1) = 48\%$, that is, *at least two applications are competing for the access to the file system about half of the time*. This makes cross-application interference frequent enough to motivate our research.[1]

6.1.2 Studying I/O Interference: a Methodology

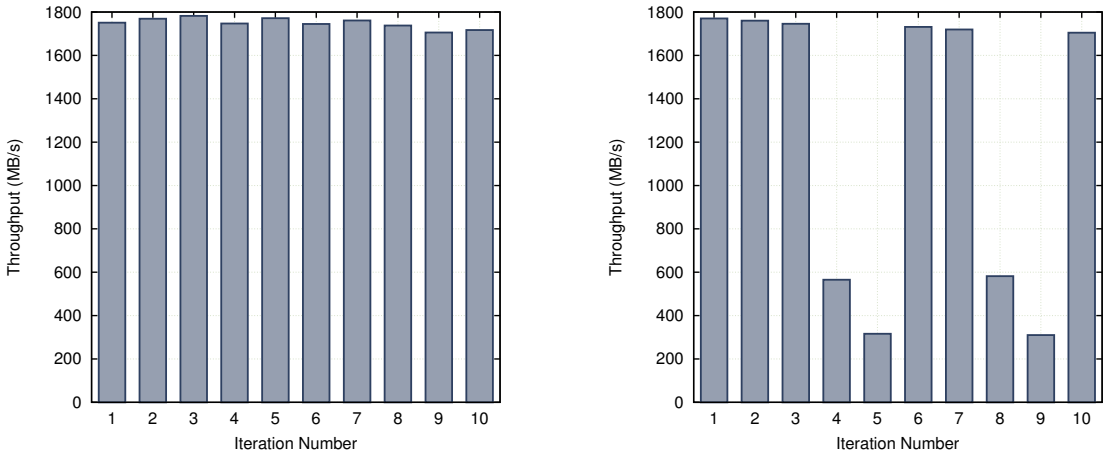
Throughout this chapter we will consider two applications A and B. To study the interference between these applications, we introduce the concept of Δ-graphs and interference factors, which are described in this section.

Δ-graphs

Application A starts writing at a reference date $t = 0$; application B starts at a date $t = dt$, and we measure the performance (for example, the time spent in an I/O phase) of A and B as a function of dt. A single experiment with a particular value of dt gives us a point in the graph. A set of experiments with different values of dt allows us to plot the measured performance as a function of dt for each application. If $dt < 0$, B starts its access before A (as a result, the Δ-graph of the pair of applications (A, B) is the mirror of the Δ-graph of (B, A)).

An example of a Δ-graph is shown in Figure 6.2, which reports experiments done on the Nancy site of Grid'5000. Here two instances of the same application run on the same number of cores. From this example, we observe that when two applications compete for the access to the file system with the same I/O load, the first one to arrive is favored, although it still observes a degradation of its write time. One can easily compute and display the expected interference as a piecewise linear function, assuming a proportional sharing of resources between the two applications. This theoretical performance is also plotted in the figure (the

[1] Note that the formula presented here differs from the one presented in our paper [27], which corresponds to the probability to see at least one application performing I/O at any given moment, that is, the probability for an application entering its I/O phase to observe another application already in its I/O phase and to interfere with it, which is different from the probability presented in this thesis.



(a) Without Interference (b) With Interference

Figure 6.3: Experiments done on the Nancy site of Grid'5000 with 35 PVFS servers across an Infiniband network: (a) one instance of IOR runs on 336 cores and writes every 10 seconds; (b) another instance is started on 336 other cores and writes every 7 seconds (the figure represents the observed throughput of the first instance only).

term "Δ-graph" has been chosen after its shape). When considering three applications the Δ-graph becomes a surface in a 3D graph, and is thus arguably more difficult to display.

Interference Factor

In the following, we will either consider the I/O time as a reference metric or use an *interference factor*, defined for a single application as the measured access time divided by the time it would require without the contention with the other application:

$$I = \frac{T}{T_{alone}} > 1$$

I is arguably more appropriate to study interference because it gives an absolute reference for a non-interfering system: $I = 1$. Moreover, it allows the comparison of applications that have different size or different I/O requirements.

I could be computed for other metrics as well, such as the energy consumption: $I = \frac{E}{E_{alone}}$. This metric depends on the application considered but also on the platform and other applications running simultaneously; it is therefore a context-dependent measure.

6.1.3 Impact of Interference on I/O Optimizations

Cross-application interference can have a severe impact on I/O optimizations at several levels of the I/O stack. As an example, Figure 6.3 shows the consequences of cross-application interference on a caching mechanism. Here two instances of the IOR benchmark [116] write periodically, one with a 10 seconds delay between each write, the other one with a 7 seconds delay. Kernel caching is enabled in the storage backend, so that applications see a higher

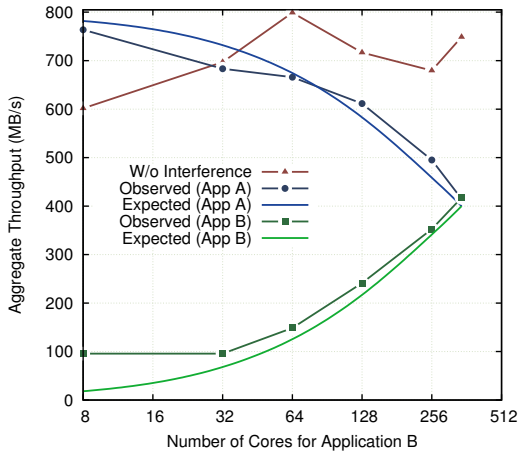


Figure 6.4: Experiments done on Grid'5000 (Nancy site) with PVFS deployed on 35 nodes; *A* runs on 336 processes; size of *B* varies; each process writes 16 MB. Both applications start at the same time.

throughput than what the disks actually provide; the file system indeed caches the writes and flushes them in disks later, when the application is not writing anymore. When the two applications happen to write at the same time (iterations 4, 5, 8, and 9), none of them benefits from the cache, and their performance drops dramatically.

6.1.4 From Diversity to System-wide Inefficiency

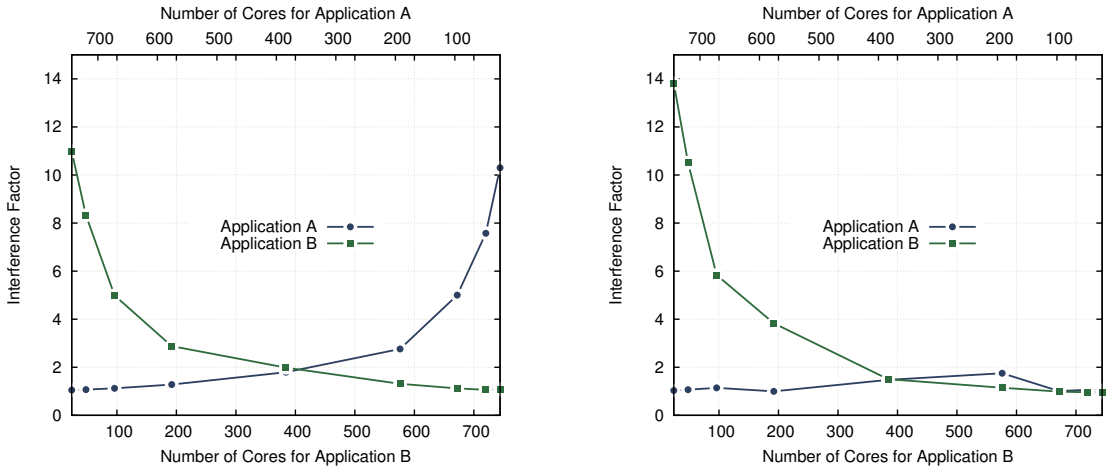
Different applications usually run on different numbers of cores, for different durations and access different amounts of data in a different manner. They also have different resource constraints and I/O requirements. For example, the CM1 atmospheric simulation on Blue Waters synchronously writes snapshot files every 3 minutes, for an amount of 23 MB/core (See Chapter 3). The NAMD chemistry simulation, on the other hand, writes trajectory files of a few bytes per core every second through a designated set of output processes, and in an asynchronous manner.[2] These behaviors and the I/O requirements that they imply cannot be captured by the storage system, which sees only incoming raw requests.

This diversity among applications and lack of knowledge that the file system has from them can lead to some applications being impacted more than they should by other applications. As an example, Figure 6.4 shows what happens to the aggregate throughput when a small application interferes with a bigger one. When application *B* runs on 8 cores while *A* runs on 336, *B* observes a 6× decrease of throughput compared with *B* running alone on 8 cores. As the number of cores used by *B* increases up to 336, *A* becomes more and more impacted.

Figure 6.5 illustrates the interference factor when two applications of different sizes write at the same time (a), or with a 5-second delay (b). Two conclusions can be derived from these figures.

- A small application is more impacted by a big one than vice-versa. More generally, an application with little I/O will observe a larger impact (relative to its I/O) when competing against an application with larger I/O.

[2]This information was gathered through discussions with the Blue Waters PRAC (http://www.ncsa.illinois.edu/BlueWaters/prac.html) users.



(a) dt = 0 sec (b) dt = 5 sec

Figure 6.5: Experiments done on Grid'5000 (Rennes site). The total number of cores is 768, Applications A and B share this numbers of cores (e.g. when A runs on 744 cores, B runs on 24 cores). The two graphs show the interference factor for different size of application given $dt = 0$ or $dt = 5$.

- A small delay between the I/O phases can make an important difference in the observed interference factor, preventing a small application from being impacted by a bigger one (left part of Figure 6.5(b)).

More important than the performance of each application individually, *cross-application interference leads to a decrease of system wide efficiency*. Depending on a given metric to measure this efficiency (for example, the sum of run time of all applications, the number of FLOPs used for actual science, etc.), it is desirable to find ways to decrease these interference factors. Doing so, however, requires some knowledge about each application's I/O behavior and requirements.

6.2 Mitigating Interference within the CALCioM Framework

Having shown that I/O interference happens frequently and potentially has a high impact on applications' I/O performance, we here propose strategies to overcome this problem. We then introduce our CALCioM framework, which integrates these strategies.

6.2.1 Interference-avoiding Strategies

Cross-application interference can have a big impact on the performance of some applications, in particular given the diversity of sizes and I/O requirements. This performance impact results in a suboptimal use of the machine. In order to mitigate interference between two applications, several strategies can be envisioned.

Serializing Accesses on a First-Come-First-Served Basis: With this strategy, only the application that arrives second in its I/O phase is impacted in a way proportional to the remaining access time of the first application. This policy requires either to give applications a (potentially dangerous) `lock` function, to ensure that the file system is accessed one application at a time, or to give them a way to *know that another application is currently doing I/O* and that there will be no advantage in interfering with it.

Interrupting an Application's Access: In this situation, the application that arrived first is impacted. Indeed, if its access can be paused quickly enough, the second application will immediately get access to the file system and will not be impacted. This strategy specifically requires a way for an application to *be interrupted*, that is, to *know that another application arrived and is expected to do I/O.*

Allowing Interference: When the interference is low enough (for instance, between two small applications) and the performance decrease can be afforded by all applications involved, then letting the applications interfere can also be a valid choice in some cases and lead to better performance than trying to schedule them.

Discussion: Adapting Dynamically to the Best Strategy

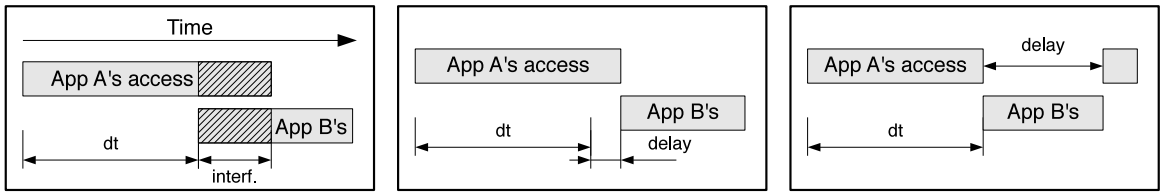
Each strategy having its own advantages and drawbacks, a mechanism can be implemented to select the best option at run time depending on information exchanged between applications. The choice of a strategy over another should be made on the basis of a system wide efficiency metric. For instance, if our goal is to minimize the sum of interference factors $f = \sum_{X \in App} I_X$, we will try to avoid the case of a small application being largely impacted by a big one, by serializing the big one after the small one or by interrupting the big one to favor the small one.

The first three strategies are presented in Figure 6.6. These strategies all require that an application becomes aware of other applications running on the system, or at least the properties of ongoing I/O operations, and even have a way to contact other applications to exchange these properties. To this end, we designed a coordination approach. This approach is illustrated by the CALCioM framework, described in the next section. It includes all these strategies and allows applications to communicate with one another in order to implement them.

Note that these strategies can naturally be generalized to more than two applications. The adaptive strategy would then consist of either choosing a place in a queue of applications that have requested access to the storage system, or interrupting the one currently accessing it.

6.2.2 CALCioM: Design Principles

CALCioM provides a way for applications to communicate with one another in order to make a decision on the best I/O scheduling strategy. Deciding could be done by the applications themselves or enforced by a system-provided entity (this detail is outside the scope of this work, as our goal is to show the possibilities offered by the sharing of information



(a) Interference (b) First-Come-First-Served (c) Interruption

Figure 6.6: Three possible policies to deal with interference: (a) let applications A and B interfere, both will be impacted; (b) serialize one application after the other, giving the advantage to the one that started its access first and impacting the second one only; and (c) interrupt one application for the benefit of another one, impacting the first one only.

between applications through a common communication layer). CALCioM seeks the optimization of a set of concurrent applications, rather than optimizing each application individually, and thus considers the set of applications running concurrently rather than each application individually. Figure 6.7 summarizes the the CALCioM framework.

A design choice central to our approach is that CALCioM *does not* give to the user a *lock* function to prevent multiple applications from accessing the file system at the same time. Nor does it offer a way for an application to force the interruption of another one. CALCioM *only provides the means by which applications can communicate*. CALCioM can be integrated in the I/O stack of applications and use the information exchanged by different applications to make a decision on the their behavior.

CALCioM works with knowledge acquired in each layer of the I/O stack of each application. It considers the I/O stack as a whole instead of a set of layers (application, I/O library, MPI-I/O, file system) to be optimized individually. For instance, CALCioM will get from the application level how many files (or how many bytes) are intended to be written and from MPI-I/O the series of raw requests to the file system, the targeted storage servers, the number of rounds of collective buffering, or other such information.

As an example of CALCioM-enabled behavior, consider an application A writing a large amount of data. As another application B starts an I/O phase, it contacts A with some information regarding its expected I/O operations, for example, a well-optimized write of a small amount of data. If, targeting the optimization of a given metric, A (or a centralized entity) considers that stopping and letting B execute its access will lead to better overall performance, it will contact B back with this decision. B will proceed with its I/O. When B finishes, it contacts A back and A resumes its own I/O operation.

6.2.3 Architecture and API

The communication between different applications and the gathering of information on I/O behaviors are done only through one process in each application (typically rank 0 in MPI_COMM_WORLD) or a reduced set of processes. This process, called a *coordinator*, is responsible for gathering information from other processes inside the application, for interacting with other applications, and for sending orders back to inner processes on accesses to be performed. CALCioM is thus hierarchical in the sense that an application can internally

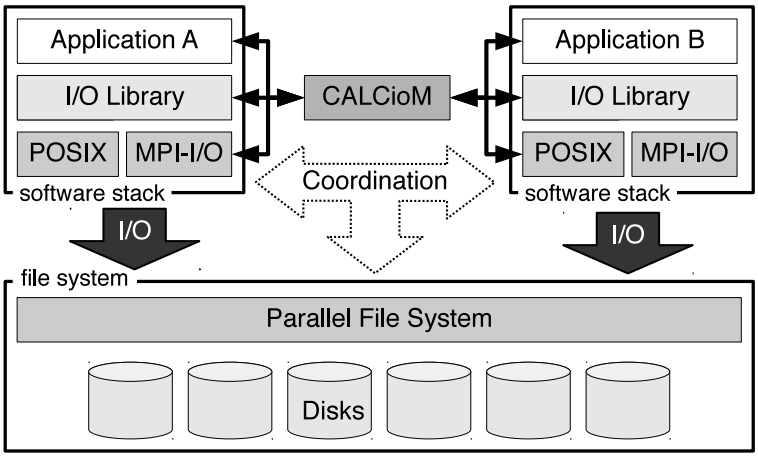


Figure 6.7: Schema of the CALCioM approach; a communication layer between independent applications that can be leveraged at every level of the I/O stacks to communicate information on the I/O behavior.

have its own way of managing I/O, and CALCioM acts as the root of each application's particular I/O management system.

Design and Implementation Options

CALCioM can be designed by using MPI as underlying communication layer in order to be platform independent. Indeed MPI already provides functions to build a communicator across multiple applications (`MPI_Comm_{connect,accept}`, `MPI_Port_{open,close}`), as applications start and leave. Yet, since an application cannot know when another one will try to contact it, these connection primitives should be made non-blocking, either by extending the MPI standard with `MPI_Comm_{iconnect,iaccept}` or by calling these functions from a thread (something that is also done by Zounmevo et al. [152]). Such a thread would be required only in coordinator processes, that is, one extra thread per application.

In a production system, one might want to implement these primitives at a system level, in order to improve their security, and to back up the coordination algorithm with a centralized entity to enforce the decisions taken on the basis of the I/O behaviors. For large-scale systems it might also be more effective to perform coordination via a separate service running on the system, rather than the peer-to-peer approach used in our prototype. Systems such as BlueGene/Q running the operating system in a spare core [45] would offer a good way of providing fully asynchronous, system-level communications with other applications.

CALCioM's API

CALCioM provides a simple API to application/library/middleware developers. These functions have to be called by all processes doing I/O.

- `calciom_prepare(MPI_Info info)` adds more information about the future I/O accesses. In order to be generic, it uses an MPI_Info structure, which contains a set of

(key,value) pairs, to represent knowledge on the application's I/O behavior. As examples of values that can be leveraged, in Section 6.3 we communicate the number of files, the number of rounds of collective buffering and the amount of data transferred per round. A call to `calciom_complete()` will later unstack this information.

- `calciom_inform()` sends the information to the set of running applications currently doing I/O, as well as applications interrupted or waiting. Suggestions of authorizations are eventually sent back by these applications.
- `calciom_check(int* authorized)` checks whether the application is "allowed" to access the file system, based on other applications' responses. That is, either this application is alone, or other applications have reached a consensus (which may have been enforced by a centralized entity) that it is best for this application to perform its I/O now.
- `calciom_wait()` explicitly waits for all the other applications to agree that this application should do its I/O access.
- `calciom_release()` ends a step in the I/O access, checks for pending requests from other applications, reevaluates the global strategy (if new information has been sent), and responds to other applications. A new call to `calciom_inform` is necessary before the next I/O access.

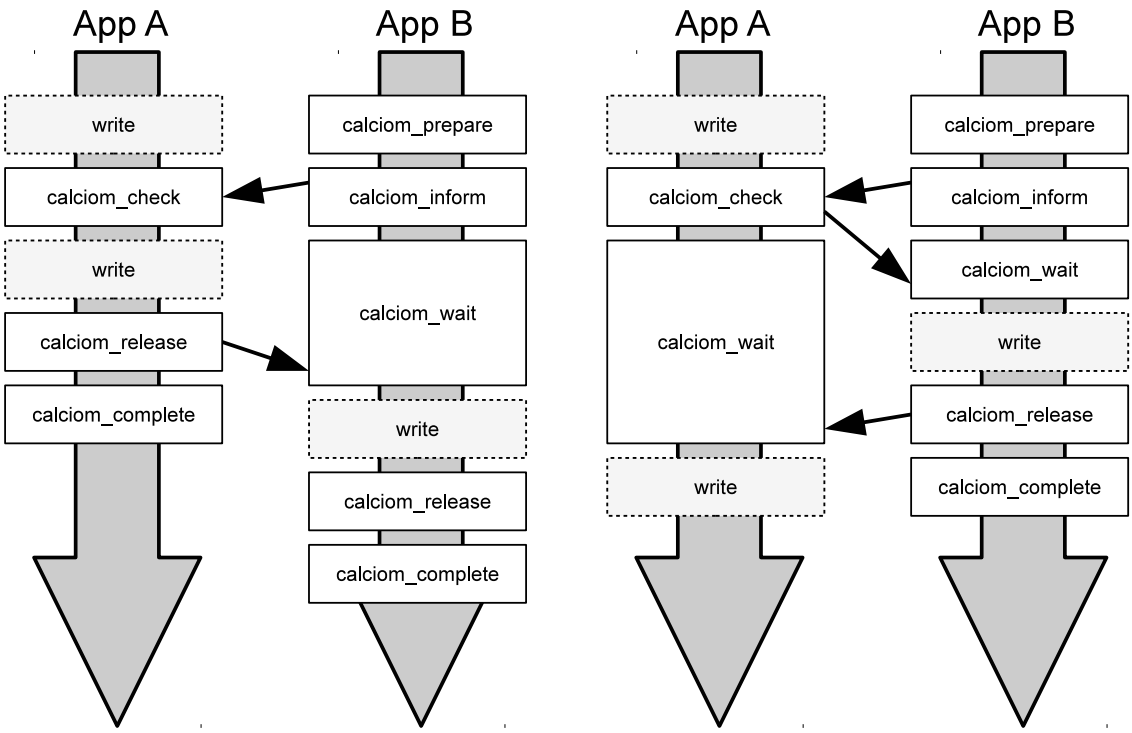
In coordinators, all these functions perform communications with other applications. Other processes perform communications with the coordinator of their application. Retrieving the list of other running applications is done through communications with the machine's job scheduler when the job starts and finishes.

Examples of Usage

Figure 6.8 shows the communications done through CALCioM in two scenarios. In Figure 6.8 (a) application *B* requests the access to the file system to application *A*, which is already writing. Application *A*, aware that *B* is expected to perform I/O, sends a notification using `calciom_release` when it is done writing. In Figure 6.8 (b) application *B* requests the access to the file system to application *A*, which is already writing. Application *A* still has some write calls to make but letting *B* write is considered more efficient. It thus answers *B*, which can write. When *B* finishes writing, it sends a notification back to *A*, which restarts writing.

Integration Level

The API presented above is intended to be used at each level of the I/O stack, from the application level down to the MPI-I/O implementation. `calciom_inform`, `calciom_check`, `calciom_wait` and `calciom_release` can be used at the low level between each atomic request to the file system, surrounding a complex *write* operation or even an entire I/O phase. The reason for also offering these functions to application and library developers is that they can also observe the load of the storage stack at any point in the program and decide to schedule their operations differently (for instance, starting a new iteration of computation and coming back to the I/O phase later). This is, however, beyond the scope of this work.



(a) Protocol for FCFS Serializations (b) Protocol for Interruptions

Figure 6.8: CALCioM's protocols for serialization and interruptions. (a) Application B waits for application A to call `calciom_release` before writing. (b) Application B interrupts application A between two atomic `write` calls.

An Holistic View of the I/O Stack

The location of the calls to CALCioM's API gives different degrees of freedom in adapting the I/O behavior; using these functions only between each file access at the application level gives fewer opportunities for the application to be interrupted upon request from another application, for example. Calling CALCioM's API at the lowest levels of the I/O stack (i.e. within POSIX or atomic MPI-I/O calls) allows a complex I/O phase to be interrupted at any moment between small, atomic I/O requests. However, a low-level-only implementation will not have sufficient information on how many files will later be accessed and how they will be accessed. Therefore it will not be able to give enough information to other applications.

These constraints force that each level of the I/O stack, including the applications and high-level I/O libraries, use at least `calciom_prepare` and `calciom_complete` to provide enough information for a better understanding of the application's I/O behavior. Therefore, not only the application developer has to use CALCioM in the application code, it also has to be integrated within I/O libraries by their respective developers.

Additionally, CALCioM will have no effect if only a small number of applications use it. An application that does not use CALCioM will still interfere with concurrent applications, whether these applications use CALCioM or not.

In the light of these requirements, the reader may object that it will be difficult to convince every application or library developers to instrument their existing code with CALCioM's API. Fortunately, we will see in Chapter 7 a solution that provides all the information on the I/O behavior without this development effort. Thus, while we did explicitly use CALCioM's API at every level of the I/O stack in the experiments presented in the next section, *it is possible to implement CALCioM in such a way that it does not require any change in the application's code, nor any I/O library.*

6.3 Experimental Evaluation

The following experimental campaign aims to present four different policies that CALCioM offers: (1) let applications interfere, (2) wait for another concurrent application to complete its I/O, (3) interrupt an application's access for the benefit of another one, and (4) dynamically select one of the above policies to optimize the machine wide efficiency.

6.3.1 Platforms and Methodology

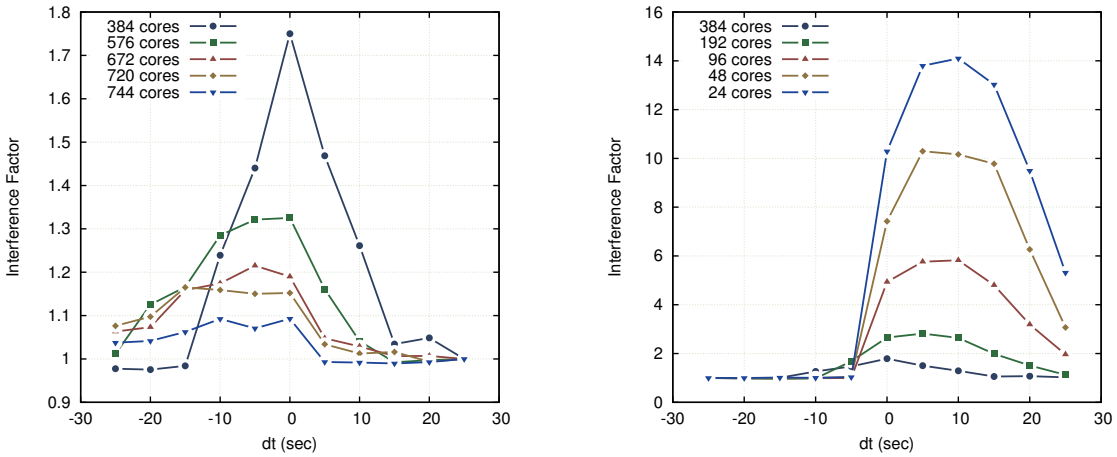
The study of cross-application interference requires reserving a full machine in order not to impact (or be impacted by) other applications. We choose the following machines for this purpose.

Surveyor

Surveyor is a 4096-core (1024 nodes) BlueGene/P supercomputer at Argonne, running at 13.6 TeraFlops. It exposes a 4-node PVFS2 shared file system for high-performance I/O. Surveyor consists of one rack of Argonne's Intrepid machine [54, 67] and therefore shares the same architecture. Note that Surveyor's PVFS2 file system is not shared with Intrepid; thus, reserving the full machine ensured that at worst only a user connected to the frontend of Surveyor could interfere with our experiments.

Grid'5000

Grid'5000 [53] was already extensively described and used in Chapters 3 to 5. We mainly used the Rennes site, more specifically the *parapluie* cluster (40 nodes featuring 2 AMD 1.7 GHz CPUs, 12 cores/CPU, 48 GB RAM) and *parapide* (25 nodes featuring 2 Intel 2.93 GHz CPUs, 4 cores/CPU, 24 GB RAM, 434 GB local disk). We leverage the InfiniBand network that connects all nodes of these two clusters. The OrangeFS file system (a branch of PVFS2) version 2.8.3 was deployed on 12 nodes of *parapide*, using an *ext3* backend file system on local disks with caching disabled in order to avoid the huge performance drop observed in Section 6.1. Reserving these two clusters and deploying our own file system ensured us to be the only users of the IB switch as well as the file system at the time of the experiments.



(a) Δ−graph of App A (big) (b) Δ−graph of App B (small)

Figure 6.9: Experiments done on Grid'5000. A total of 768 cores is split into two groups of N (App B) and $768 - N$ (App A) cores, for $N \in \{24, 48, 96, 192, 384\}$. Each application writes 16 MB (8 strides of 2 MB) per process.

Application

Using real-life applications to evaluate cross-application interference is arguably not appropriate because (1) it is difficult to differentiate inner and outer causes of performance degradations in applications that exhibit a complex access pattern, (2) they may not be representative of generic interference patterns that applications with perfectly optimized I/O would exhibit, and (3) we need a way to control precisely the moment when these applications perform I/O. Therefore, we developed a simplified[3] version of the IOR benchmark [116] that starts by splitting its set of processes into groups running independently on different nodes. This IOR-like benchmark allows us to control the access patterns of each group of processes (for example, contiguous or strided with a specified number of blocks and block sizes, in a way similar to IOR). For this work specifically, our study focused on collective write operations and write/write interference between two applications only.

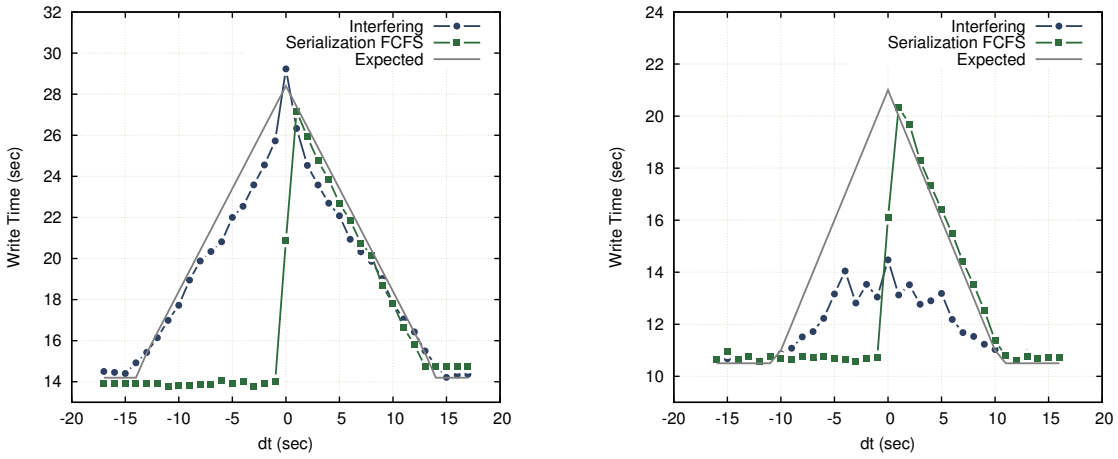
6.3.2 Interfere or Serialize Accesses?

Our first experiments aim to illustrate the potential advantage of serializing I/O accesses of two applications as opposed to letting them interfere.

Benefits of Avoiding Interference through Serialization

Small Application vs. Large Application: Figure 6.9 completes our study of interference initiated in Section 6.1 between applications running on different numbers of cores.

[3]Our version does not provides all the backends such as HDF5 that IOR provides, and I/O patterns are hard-coded instead of being configurable.



(a) 2× 2048 cores (b) 2× 1024 cores

Figure 6.10: Experiments done on Surveyor. Two applications of the same size write 32 MB per process using a contiguous pattern: (a) the applications are big enough to interfere with each other; (b) the applications are smaller and the interference is not as high as expected.

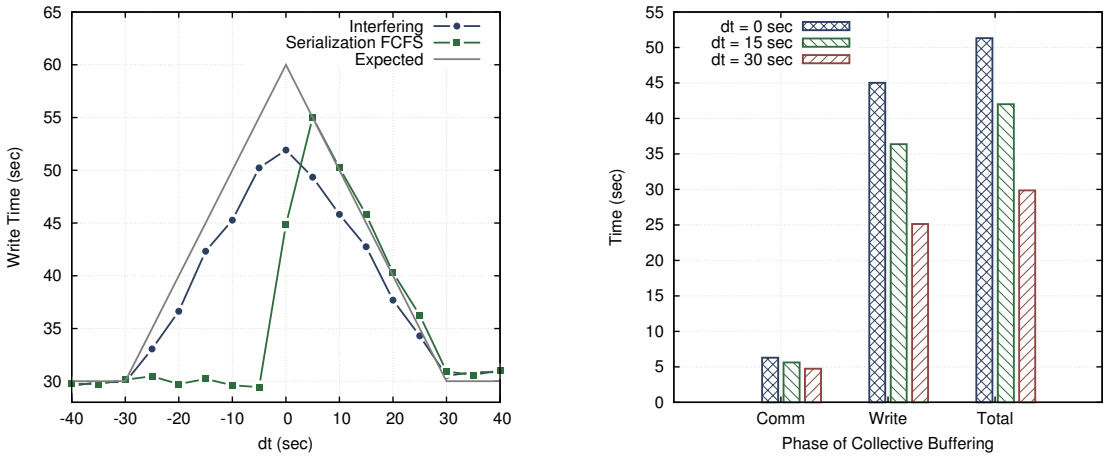
We observe that the small application (B) is a lot more impacted than the big one, with an interference factor going up to 14 for a 24-core instance competing with a 744-core instance. On the left part of the graphs ($dt < 0$), B manages to write before A starts writing, which prevents them from interfering. On the right part, however, B starts while A is already writing, thus leading to interference. Provided that we try to minimize the sum of write times, or the sum of interference factors, regardless of the number of cores on which the applications run, a smarter strategy consists of having instance A wait for B to have completed its write before starting its own operation (i.e., being on the left side of the Δ-graphs as often as possible). This is possible only if B starts writing before A and A has a way to know that B is writing, in which case the decision to wait for B to complete is taken either by A or by a system-provided entity that enforces the decision.

Yet given a time interval $[t_1, t_2]$ during which both A and B are expected to complete exactly one I/O phase, we can show that the probability for B to start writing while A already started (in which case B will either have to interfere with A or be serialized after it) follows Equation 6.3.

$$\mathbb{P}(dt < 0) = \frac{T_{A(alone)}}{t_2 - t_1} \tag{6.3}$$

The bigger the difference in size between the applications, the less likely relying only on an FCFS (First-Come-First-Served) policy will allow us to achieve our target of system wide efficiency.

Equally-sized Applications: Figure 6.10 (a) shows two accesses from instances of the same size serialized one after the other. As opposed to the case where these applications interfere, only the application that accesses second is impacted and experiences a performance degradation that is equivalent to that of an interference with the first application. The application



(a) $\Delta-$graph (b) Phases of collective buffering

Figure 6.11: Experiments on Surveyor. Two applications of the same size (2048 cores each) write 16 MB per process using a strided pattern (16 blocks of 1 MB per process), triggering the collective buffering algorithm. Figure (a) shows the $\Delta-$graph when the application interfere and when their accesses are serialized one after the other. Figure (b) shows how each of the two phases behave: the communication phase is almost not impacted (it is impacted as a side-effect of the variability in the write phase of different processes), while the write phase is the most impacted.

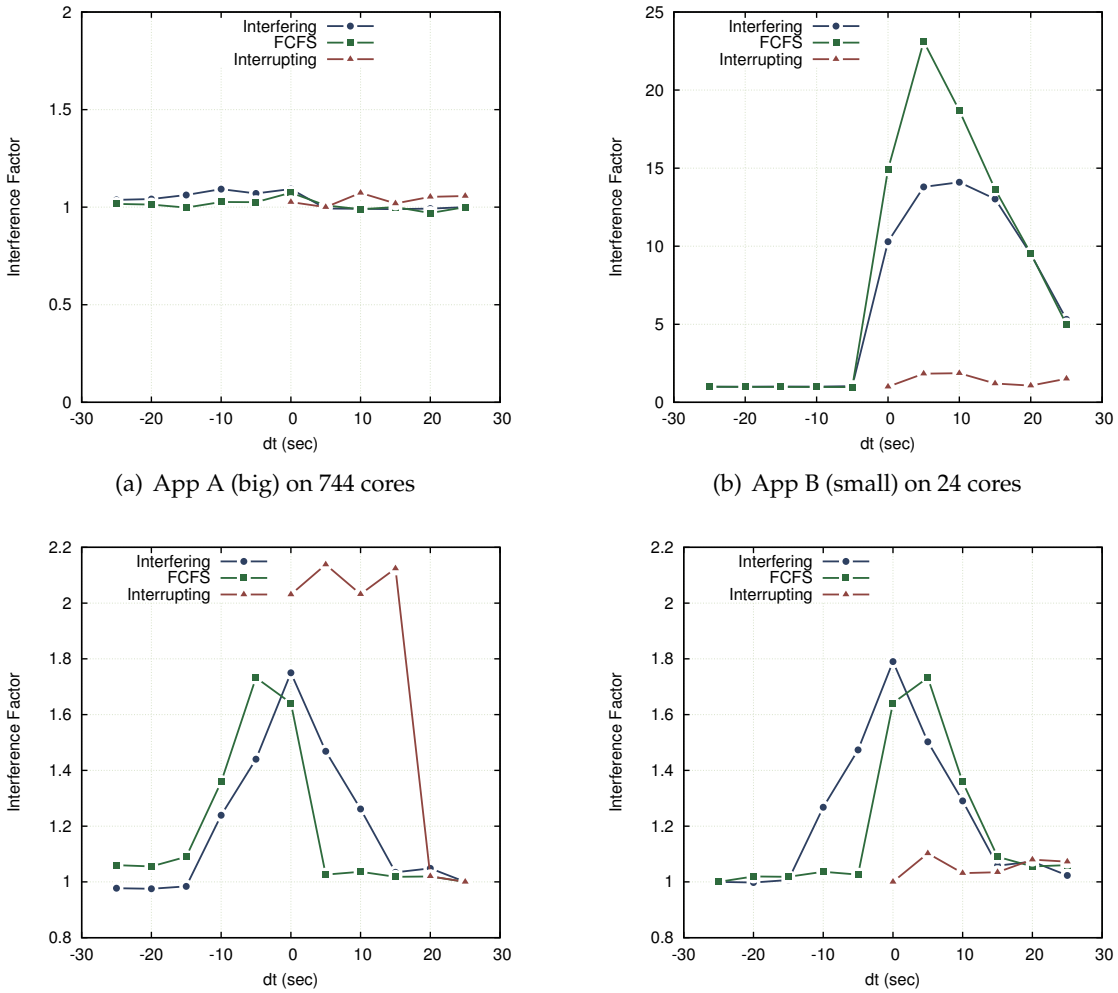
writing first, however, is not impacted anymore, hence *leading to a better overall system performance*.

Limitations of the Fist-Come-First-Served Strategy

The limitations of the FCFS strategy are however numerous. Some of them are illustrated in the following experiments.

Small Application vs. Small Application: Figure 6.10 (b) presents a case where the applications have the same size, but this size being small, the compound A+B tolerates rather well the interference. Serializing the accesses will benefit only the first one, at the expense of the second.

Using Strided Patterns: In Figure 6.11 (a), each instance uses a collective, strided access pattern. This access pattern triggers the collective buffering algorithm (also termed "two-phase I/O" and described in Chapter 2) that introduces collective communication steps. These communications are less subject to interference, as shown in Figure 6.11 (b), and therefore, serializing the accesses has a higher impact on the application arriving second than blind interference. Note that this result is observed on Surveyor, where different applications don't share their network (partitions on which they run in the supercomputer are electrically isolated). This observation was not made on Grid'5000 where the sharing of a



(a) App A (big) on 744 cores

(b) App B (small) on 24 cores

(c) App A (big) on 384 cores

(d) App B (small) on 384 cores

Figure 6.12: Experiments done on Grid'5000's Rennes site. Two applications write 8 MB per process using a strided pattern, from a different number of cores (App A runs on 744, 720, 672, 576, and 384 cores, and App B respectively on 24, 48, 96, 192, and 384, for a total of cores of 768). We show how the interference factor behaves for the 3 policies: interfere, serialize one application after the other (leading a an important performance degradation for B when B is small, as shown in Figure (b)), and interrupt A (leading to a performance degradation for A if B is of the same size, as shown in Figure (c)).

common InfiniBand switch for both I/O and communications makes communication steps interfere as well.

Delaying Small Application: The experiments presented in Figure 6.12 show that FCFS serialization has a positive effect when applications have a similar size or similar I/O requirements (Figures 6.12 (c) and (d)). However, when they have very different sizes (Figures 6.12 (a) and (b)) or very different I/O requirements, the FCFS serialization leads to an important performance degradation of the small application when this application arrives

second and is delayed after the big one. As explained above, at equivalent access frequency between the two applications, the situation of a small application accessing before the big one is less likely than its opposite. Depending on the global efficiency targeted, *it may thus be desirable in these situations that the big application be interrupted for the benefit of the small one*.

6.3.3 A Third Option: Access Interruption

By frequently calling `calciom_inform/release`, an application has the possibility to receive information from other applications more often, and also be interrupted at a finer grain. These interruptions can overcome the limitations of the FCFS serialization strategy.

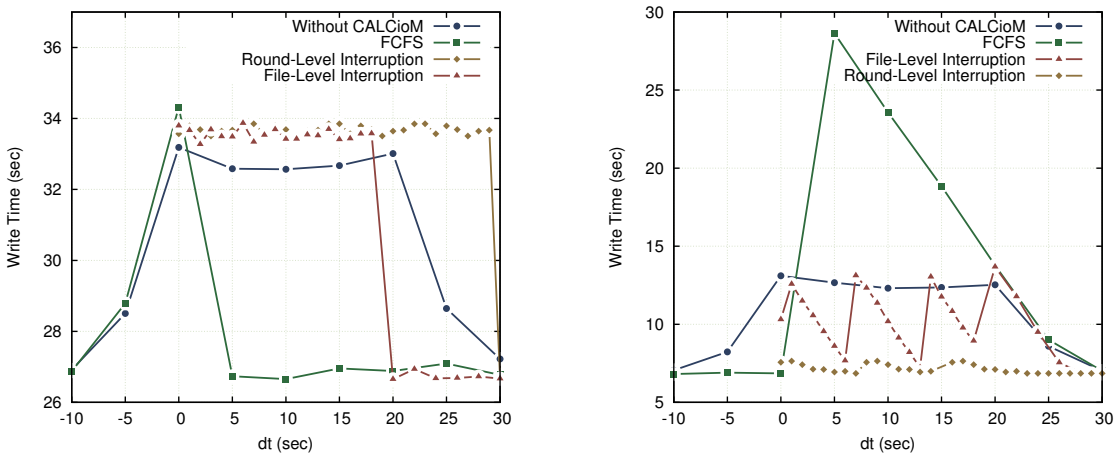
Preventing Small Applications from Being Delayed: Figure 6.12 also presents the results of experiments where the application accessing second *interrupts* the one accessing first, regardless of the size or I/O requirements of each application. These experiments are done with a strided write pattern using collective buffering, and `calciom_inform/release` are called before and after each atomic call to independent contiguous writes in a custom, CALCioM-enabled ADIO layer for ROMIO. The interruption being possible only when $dt > 0$ (e.g., there is someone to interrupt), the curves start at $dt = 0$. These figures show that, as expected, the interruption strategy has the opposite effect to FCFS serialization; it is effective when a small application interrupts a big one, but it becomes ineffective and even counterproductive when applications have a similar size.

Impact of the Integration Level: Figure 6.13 shows results on Surveyor with interference, FCFS serialization and interruptions. In these experiments, applications have the same size; however, A writes four files while B writes only one. The `calciom_inform/release` functions have been set up in two different levels: in the ADIO layer (between each round of collective buffering) or at the application level between each file. The second case leads to the "saw" pattern because A cannot be interrupted at a fine grain, and is forced to finish writing a file before being interrupted. An implementation in the ADIO layer offers more possibility for A to interrupt its access quickly enough for B not to be impacted.

6.3.4 Dynamic Choice: Interfere, Serialize, or Interrupt?

The previous sections have demonstrated the pros and cons of different policies made possible by CALCioM thanks to cross-application coordination. It also showed that the FCFS and Interruption strategies are complementary, that is, they are both optimal under different conditions.

To close the loop, we integrated all three policies in CALCioM and made it select the most appropriate strategy dynamically, based on information exchanged between applications. This selection is based on the targeted machine wide efficiency metric. In this section, we consider an example of such a metric; namely, the total number of CPU hours actually used for doing science, and show how CALCioM can select the best strategy. We thus aim at minimizing the total number of CPU hours wasted in I/O phases: $f = \sum_{X \in Apps} N_X \times T_X$ where N_X is the number of cores running application X, and T_X is the observed I/O time. Note that this metric does not necessarily favor a small application, since it weights the I/O



(a) App A (big) (b) App B (small)

Figure 6.13: Experiments done on Surveyor. App A and B run on 2048 cores each, App A writes 4 files using 4MB per process (contiguous access), App B writes only 1 such a file. These graphs show the interference factor of the two applications depending on the strategy used and on dt.

time with the amount of computing resources. However, it will favor a big application with small I/O requirements.

Theoretical Analysis: We consider the scenario presented in Figure 6.13 where $N_A = N_B =$ 2048 cores on Surveyor, and B writes four times less data than A. The case of B starting before A is trivial (A is serialized after B). Thus we consider only $dt > 0$; B either interrupts A or is serialized after it. Using the above definition of f, we can compute the expected cost of each of the policies using Equations 6.4.

$$\begin{aligned} f_{FCFS} = &\quad N_A \times T_{A_{(alone)}} + N_B \times (T_{A_{(alone)}} + T_{B_{(alone)}} - dt) \\ f_{Interrupt} = &\quad N_A \times (T_{A_{(alone)}} + T_{B_{(alone)}}) + N_B \times T_{B_{(alone)}} \end{aligned} \tag{6.4}$$

Given $N_A = N_B$ and $T_{A_{(alone)}} = 4 \times T_{B_{(alone)}}$, A should be interrupted if and only if

$$f_{Interrupt} < f_{FCFS}, \tag{6.5}$$

which translates into $dt < \frac{3}{4} T_{A_{(alone)}}$. As a result,

- If B starts first, A is serialized after B;
- If B starts before A finished writing 75% of its data, A is interrupted;
- Otherwise B is serialized after A.

Experimental Results: The result of these decisions on the value of f is summarized in Figure 6.14 (lower is better) and compared with the situation of applications simply interfering without CALCioM involved. Considering this specified metric of computational efficiency,

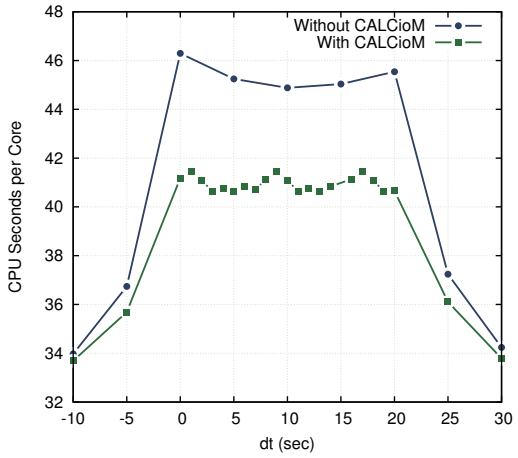


Figure 6.14: Synthesis on CALCioM's choices and impact on the specified metrics (computational efficiency). Experiment performed on the Surveyor machine (see configuration in Figure 6.13). The figure shows the CPU seconds per core wasted in I/O under interference, and with CALCioM selecting the appropriate approach depending on dt.

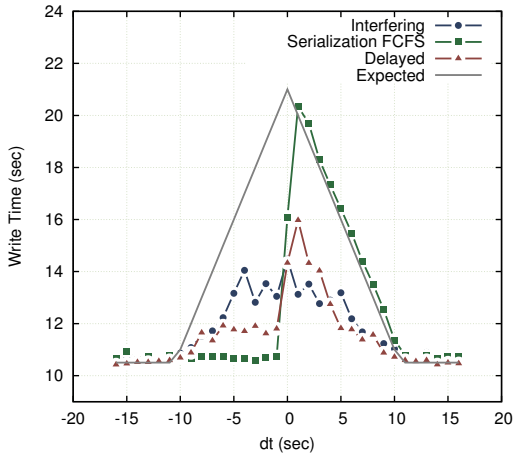


Figure 6.15: Experiment performed on the Surveyor machine. 2×1024 cores write 32MB/process (contiguous pattern). The interference is not as high as expected. As a consequence, serializing accesses is not a good decision. A tradeoff can be found by slightly delaying one of the writes.

CALCioM always manages to make a decision that improves this metric; that is, it lowers the global time wasted in I/O per core.

Beyond Serialization and Interruption: Selecting a policy does not necessarily mean simply choosing between FCFS and interruption. Indeed, in a context where the observed interference is lower than expected, as in Figure 6.15, we have shown in Section 6.3.2 that serialization (or interruption) is not a good option. More elaborate decisions could be made, such as delaying an application and allowing some degree of overlap. This decision still depends on the specified system wide efficiency metric to optimize, but it requires a better estimation of the interference by the applications, an estimation outside the scope of this work.

6.4 Discussion and Related Work

In this section we present and discuss related work. CALCioM is fundamentally different from all approaches that address I/O interference. These approaches are indeed based on scheduling requests in the parallel file system, while CALCioM proposes *cross-application*

coordination by having each individual application communicate their I/O behavior.

6.4.1 Application-Side I/O Scheduling

The approach closest to ours has been proposed by Lofstead et al. [75], but addresses cross-process I/O interference within a single application. Based on the observation that within a single application, processes already interfere with one another, they introduce an adaptive I/O approach in which the processes of an application are gathered in groups. Each group writes in a particular storage server, and one process in each group is chosen to coordinate the accesses issued by all the other processes in the group. This drastically reduces I/O variability within a single application. CALCioM targets the same goal but at machine scale, across multiple applications. This task is inherently more difficult because of the lack of knowledge that applications have about one another, the diversity of their I/O workloads and the fact that applications come and leave, making the set of entities to coordinate dynamic.

6.4.2 Server-Side I/O Scheduling

I/O scheduling techniques implemented in parallel file systems aim at lowering disk-head movements caused by unrelated requests (i.e., achieving better *data locality*). They also try to better distribute I/O requests across multiple data servers. These objectives imply (1) trying to service applications one at a time and (2) trying to force all the data servers to serve the same application at the same time, while keeping fairness across multiple applications. Our experimental evaluation clearly showed that serializing I/O requests without knowledge of the applications' I/O load can lead to machine-wide inefficiency.

Zhang et al. [147] leverage the notion of a "reuse distance" to state whether it is worthwhile for a data server to wait for an application's new I/O request or to service other applications requests. In contrast, CALCioM coordinates all running applications without the need for requests to carry an ID, or for the file system to wait arbitrarily for potential new requests to arrive. Using CALCioM, the file system is, in fact, unaware of the coordination strategies implemented by the applications themselves.

Other approaches such as the one from Lebre et al. [69] provide multi-application scheduling with the goal of better aggregating and reordering requests. It also tries to maintain fairness across applications. However the proposed solution does not take into account each application's available resources and required I/O efficiency and does not check the availability of the file system to potentially change the application's behavior.

6.4.3 Application-Aware I/O Scheduling

Some research efforts consider information from the application level in order to improve their scheduling strategies.

Qian et al. [109] present a network request scheduler built in Lustre [22]. They propose to associate deadlines to requests, as well as their targeted object's identifier, in order to first service requests belonging to the same object while preventing starvation by taking deadlines into account. They propose to dynamically adapt the deadline value depending

on the load on the file system, and to add mandatory deadlines for requests that correspond to critical I/O operations (a cache becoming full in the client, for instance). The same goal is achieved by Song et al. [120], with an application's *id* instead of an object *id*. The use of deadlines helps avoiding starvation but does not aim to reduce interference in any ways. The information leveraged from clients is also very rudimentary. In contrast, exposing the I/O behavior of applications in CALCioM allows CALCioM to know precisely in which order the requests have to be issued and serviced for best overall performance.

Zhang et al. [148] propose to coordinate the schedulers of each data server in order to meet QoS requirements set by each application in terms of application run time. The required application run time is converted into bandwidth and latency bounds through machine learning techniques. The application I/O behavior must be extracted from a first run on a dedicated platform. I/O schedulers in data servers then allocate time windows to serve one application at a time in a coordinated manner. Our approach does not require machine learning techniques but requires information sharing between applications. We also aim to improve machine-wide efficiency instead of improving the QoS of single applications.

Closer to our approach is the work from Batsakis et al. [4], where the observation is made that different clients with different resource usage should be serviced differently by the file system. In there solution, clients price their requests depending on their ability to delay them (which depends on the memory usage on the client). The server also prices all requests based on its own availability. An auction mechanism is then implemented to chose whether a request should be serviced or delayed. This mechanism is constrained to asynchronous requests and involves communications between the client and the server to set up the auction. Our approach does not assume asynchronous requests.

Tanimura et al. [124] propose to reserve throughput from the storage system. Their system is implemented in the Papio file system. Applications have to define their requirements in terms of throughput either when submitting a job or at run time, and the level of service is controlled by a centralized manager. Reserving throughput may not be an effective way of improving machine wide efficiency, as it locks resources while most applications have a periodic behavior, alternating between I/O intensive phases and computation phases during which no I/O is performed. We approached the problem in a different way by giving the maximum performance possible to all applications, and by resolving interferences as they occur based on a specified metrics of platform efficiency.

Zhang et al. [146] propose an approach that couples the I/O scheduler and process scheduler on compute nodes. When an application becomes I/O intensive, processes fork to create new processes that executes the same code only to retrieve information on the future I/O requests that will be issued. These pre-execution processes are then killed and the main processes can leverage knowledge from their own future I/O requests. Their implementation is done under MPI-I/O and in PVFS. This techniques is complementary to our approach; it manages to give a prediction of future I/O behavior that could then be leveraged by CALCioM. Another techniques that does not require to spawn additional processes will be proposed in the next chapter.

To our knowledge, none of the existing approaches to I/O scheduling and I/O optimizations leverage both the facts that (1) applications can themselves communicate with one other and self-coordinate and (2) applications have different constraints related to their resources usage, I/O load and behavior, which should be taken into account when targeting machine wide efficiency.

6.5 Conclusion

Distributed systems are by nature subject to concurrency. Performance variability as a consequence of resource sharing is a well-known problem in cloud computing, for example. Cloud users share not only network bandwidth, but also the hardware on which their VMs run [88]. In this context, performance guarantees are part of the service-level agreement that also defines the pricing model of the platform; hence, interference has economical consequences. Pu et al. [107], for example, provide a study of interference specifically for I/O workloads in the cloud.

In the supercomputing community however, the lack of an underlying pricing model, along with the fact that computing resources are fully dedicated to a single job at a given moment, did not motivate much analysis of cross-application interference. Yet cross-application contention is mentioned by Skinner and Kramer [117] as one of the five main causes of performance variability in HPC systems, in particular at the level of parallel file systems, which remains the main shared resource of the platform and thus the main point of contention between applications. In their own words, cross-application contention is in fact *one of the most complex manifestations of performance variability on large-scale parallel computers*.

Uselton et al. [134] also mention that the high variability observed in the I/O performance of HPC applications is caused by factors coming from both inside and outside the application, which makes its analysis even more challenging.

Cross-application interference in HPC systems, and more particularly in their I/O system, is therefore an important problem that can affect the efficiency of the entire machine. This problem will be even more important with Exascale machines that will allow running more applications in a concurrent manner. In this chapter we explored the effect of cross-application contention on their I/O performance. We propose the CALCioM approach, which provides a means by which independent applications can communicate with one another in order to coordinate their I/O strategy, targeting system wide efficiency. We illustrated the usefulness of our approach through experiments on two platforms: Argonne's Surveyor and the French Grid'5000 testbed. For example, CALCioM is able to prevent a $14\times$ slowdown of a small application competing with a larger one, at a negligible cost for the latest, by allowing the interruption of its ongoing I/O operations. CALCioM opens a wide range of new possible scheduling optimizations through the sharing of I/O properties between applications. We intentionally focused our study on interference between two applications only, as displaying interference factors in the context of more than two applications is arguably difficult.

An important limitation of CALCioM was described in Section 6.2.3: CALCioM requires a global view of the I/O behavior of the application and thus, the I/O stack has to be instrumented at each level, from the application down to the low-level POSIX and MPI-I/O layers. Additionally, we left aside the question of *how an application knows about its own I/O*, assuming that this question could be partially answered at each level of the I/O stack. In the following chapter, we provide an answer to this question through the Omnisc'IO approach, which *transparently captures and models any HPC application's I/O behavior*.

Chapter 7

Modeling and Predicting I/O: the Omnisc'IO Approach

Contents

THE effectiveness of an approach like CALCioM, presented in the previous chapter, strongly depends on a certain level of knowledge of the I/O access patterns. As explained in Chapter 2, this kind of knowledge can also be useful to other techniques such as prefetching, caching, and scheduling [47, 131]. Prefetching and caching indeed require the location of future accesses (i.e., spatial behavior), while I/O scheduling leverages estimations of I/O requests interarrival time (i.e., temporal behavior). For example, the

work by Boito et al. [143] shows a 46.3% performance improvement when the file system's I/O scheduler leverages trace-based prediction of future access patterns. The key challenges inherent in these techniques thus include the proper comprehension and exploitation of the application's I/O behavior within the I/O stack itself [48, 47]. This makes *modeling and predicting the applications' I/O behavior of utmost importance*.

7.1 Limitations of Current Approaches to I/O Prediction

Most of the works aiming to predict the I/O patterns of HPC applications either predict the spatial location of future accesses, or their date, but rarely both at the same time. Additionally, many of them do not work without prior knowledge of the applications: they either require trace-based offline training or multiple pre-executions of the application to build an accurate model. Table 7.1 lists a number of such approaches. While our proposed approach is not the only one to satisfy all criteria, the approach proposed by Zhang et al. [146] have some drawbacks that will be discussed in Section 7.4.

Our work addresses the limitations of current prediction systems and takes a step forward toward intelligent I/O management of HPC applications in next-generation post-Petascale supercomputers [55] that is capable of run-time analysis and adaptation to the I/O behavior of applications. To this end, this chapter presents the design and implementation of Omnisc'IO, a grammar-based approach for modeling the I/O behavior of *any* HPC application. Omnisc'IO leverages this model to predict *when* future I/O operations will occur (i.e., predict the inter-arrival time between I/O requests), as well as *where* and *how much* data will be accessed (i.e., predict the file being accessed as well as the location – offset and size – of the data within this file).

The intuition behind Omnisc'IO is that, on one hand, while statistical models are appropriate mostly for phenomena that exhibit a random behavior, the (mostly) deterministic behavior of HPC applications, inherent from their code structure, makes other representations of their I/O behavior possible. On the other hand, formal grammars, as natural models to form a sequence of symbols, have been widely applied to areas of text compression, natural language processing, music processing, and macromolecular sequence modeling [62, 91]. Therefore, an approach based on formal grammars is suitable for I/O behavior modeling, since it detects the hierarchical nature of the code that produced the I/O patterns, with its nested loops and stacks of function calls. *To the best of our knowledge, grammar-based models have never been used in the context of HPC applications. Omnisc'IO is the first prediction system that adopts this appealing approach*.

Omnisc'IO solves the main limitation of CALCioM, which requires that every level of the I/O stack be instrumented with CALCioM's functions in order to provide information on the I/O behavior of the application. In contrast, Omnisc'IO is *transparently* integrated into the POSIX and MPI I/O stacks and *does not require any modification in applications or higher-level I/O libraries*. It works *without any prior knowledge* of the application, and *it converges to accurate predictions of the I/O behavior within a couple of iterations* of the simulation. Omnisc'IO can be applied at the core of many I/O optimizations, including CALCioM but also any prefetching, caching, or scheduling system as well.

In order not to undermine the generality of our approach, this chapter does not present the use of Omnisc'IO in a particular context (i.e., prefetching, caching, or scheduling). The

Table 7.1: List of approaches to I/O prediction, with the nature of the predictions (temporal or spatial), run-time learning of the model, underlying method and subsequent usage.

Approach	Temporal pred.	Spatial pred.	Run-time learning	Method	Usage
Oly and Reed [94]	No	**Yes**	No (trace-based)	Markov models	Prefetching
Chen et al. [15]	No	**Yes**	**Yes**	Speculative execution	Prefetching
He et al. [47]	No	**Yes**	No (several runs)	Knowledge graphs	Prefetching
Kroeger and Long [64]	No	**Yes**	**Yes**	Partitioned context modeling	None
Gniady et al. [41]	No	**Yes**	**Yes**	Stack frames	Caching
Madhyastha and Reed [80]	No	**Yes**	No (trace-based)	Neural networks	None
			No (several runs)	Hidden Markov models	
He et al. [48]	No	**Yes**	**Yes**	LZ77-inspired	Prefetching
Tran and Reed [131]	**Yes**	No	**Yes**	ARIMA models	Prefetching
Byna et al. [9]	**Yes**	**Yes**	No (several runs)	I/O signatures	Prefetching
Zhang et al. [146]	**Yes**	**Yes**	**Yes**	Pre-execution process	Scheduling
Omnisc'IO	**Yes**	**Yes**	**Yes**	Sequitur grammars	None

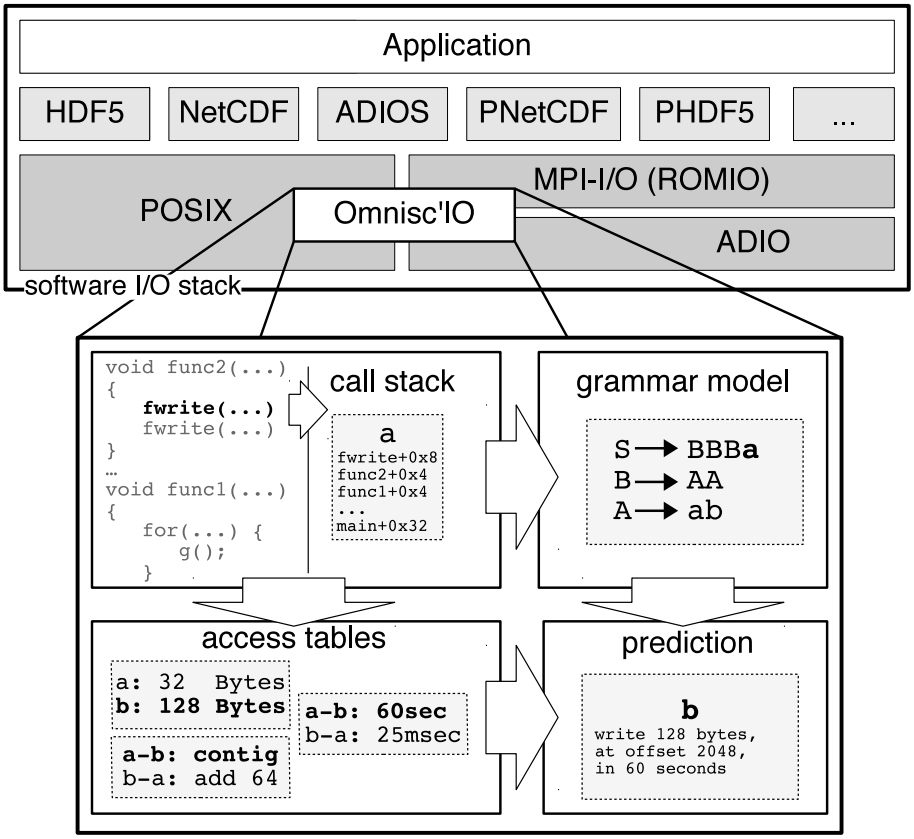


Figure 7.1: Overview of the Omnisc'IO approach, its architecture (bottom) and integration within the I/O stack (top).

previous chapter already demonstrated the benefits of exposing an application's I/O behavior in the context of cross-application coordination. Other researchers have shown the benefits of applying I/O predictions to enhance the performance in prefetching, caching, or scheduling techniques [94, 131, 9, 80]. We focus our study on the prediction capabilities of Omnisc'IO rather than its effective application in any of the aforementioned optimizations.

7.2 The Omnisc'IO Approach

This section first gives an overview of Omnisc'IO, then dives into its technical and algorithmic details.

7.2.1 Overview of Omnisc'IO

Figure 7.1 presents an overview of Omnisc'IO. Omnisc'IO captures each atomic request to the file system (open, close, read, write) in a transparent manner within the POSIX and MPI-I/O layers, without requiring any change in the application or I/O libraries. At each operation, Omnisc'IO operates as follows.

1. The *context* in which the operation is executed is extracted by recording the call stack of the program (upper-left part of Figure 7.1). This is a known techniques [41] that helps to capture the structure of the code that issues the I/O operations. The context is abstracted as a *context symbol* (*a* in the figure).

2. A grammar-based model of the stream of context symbols (upper-right part of Omnisc'IO's architecture in Figure 7.1) is updated by using the Sequitur algorithm. Sequitur has been applied to text compression in the past [62] because of its ability to detect several occurrences of substrings in a text and to store them into grammar rules. We have adapted it to model the repetitive behavior of an HPC application, represented as a stream of context symbols. The application of Sequitur to the field of application behavior modeling is novel and constitutes part of our contribution.

3. Spatial (size, offset, file) and temporal (interarrival time) access patterns are recorded in tables associating context symbols or transitions among symbols with access patterns (lower-left part of the architecture in Figure 7.1). The intuition is (for the example of the access size) that a given context symbol will often be associated with the same access size or with a reduced number of sizes whose sequence can also be learned.

4. We improved on the Sequitur algorithm to *make predictions* of future context symbols. It then becomes easy to predict the characteristics of future accesses by looking up the access patterns associated with the predicted context symbols in the aforementioned tables (lower-right part of Figure 7.1) Turning Sequitur into a prediction system is a challenge that, to the best of our knowledge, has never been addressed before. The algorithmic details of our prediction model therefore constitutes another important part of our contribution.

7.2.2 Algorithmic and Technical Description

As shown in Figure 7.1, Omnisc'IO is integrated within the POSIX I/O layer and in MPI-I/O. The following sections provide more details on the four steps described above.

Tracking Applications' Behavior

To give a context to each atomic I/O operation, we use the libc `backtrace` function to retrieve the list of stacked program counters (array of `void*` pointers). When called within a function `f`, this list of addresses represents the series of return addresses that leads from `f` back to `main`. Different calls to `f` in distinct places in the program (or libraries) lead to different *call-stack traces*. Omnisc'IO calls `backtrace` within wrappers of atomic I/O functions, stores the returned array in a dictionary, and associates it with a unique integer. In the following, such integers are called *context symbols* and represent the context in which an I/O operation occurs.

Omnisc'IO is based on the observations that (1) a particular context is likely to be associated with fixed parameters (e.g., two calls to `write` within the same context are likely to involve the same amount of data); (2) transitions between two contexts can also be associated with fixed parameters (tracking the evolution of the offset can be done by tracking transitions between contexts) and with little-varying transition times; and, most importantly, (3)

the stream of context symbols is eventually predictable, and a model of it can be built at run time.

In our prototype, we overloaded the POSIX I/O functions (`write`, `read`...) and the libc functions (`fwrite`, `fread`...) using a preloaded shared library. In MPI-I/O we added an intermediate layer within the ADIO layer in ROMIO, a popular implementation of MPI-I/O [126], to track the lowest-level I/O functions that access files metadata (open, close...) and atomic functions that access contiguous blocks of data and that are used by more elaborate I/O algorithms.

While working at the lowest level of the I/O stack is necessary to capture the I/O behavior at a fine grain (i.e., a series of atomic requests to the file system), the use of backtraces lets Omnisc'IO have an information also on the upper layers that issued the I/O, including I/O middleware, libraries and finally the application itself. This is specifically what allows Omnisc'IO to overcome the limitation of CALCioM, shown in Chapter 6, which required an instrumentation of each and every layer of the application's I/O stack by its developers.

Learning the Grammar of the Application: While capturing a stream of symbols representing the behavior of the application, we aim to predict the next symbols given past observations. Omnisc'IO models the stream of symbols using a context-free grammar. This grammar is learned at run time using an algorithm inspired by Nevill-Manning and Witten's *Sequitur* algorithm [92].

As background, a context-free grammar G is a quadruple $(\Sigma, \mathbb{V}, \mathcal{R}, S)$, where Σ is a finite set of *terminal* symbols (in our case the symbols defined by the call stack traces); $\mathbb{V}$ is a finite set of *non terminal* symbols disjoint from Σ; $\mathcal{R}$ is a finite relation from $\mathbb{V}$ to $(\mathbb{V} \cup \Sigma)^*$, usually written as a set of rules in the form $A \rightarrow x_1...x_k$, where $x_i \in (\mathbb{V} \cup \Sigma)$; and $S \in \mathbb{V}$ is a starting symbol. In the following, we call x_i the *nested symbols* of A.

Sequitur builds a context-free grammar from a stream of symbols by updating the grammar at each input. It starts with a single rule S. At each new input x, it appends x to the end of rule S and recursively enforces two constraints:

> **Digram uniqueness:** Any sequence of two symbols $ab \in (\mathbb{V} \cup \Sigma)^2$ (digram) cannot appear more than once in all rules. If one does, a new rule $R \rightarrow ab$ is created and replaces all instances of the digram ab, and the constraints are enforced recursively.

> **Rule utility:** All rules should be instantiated at least twice. If a rule appears only once, its instance is replaced with the content of the rule, the rule is deleted, and the constraints are enforced recursively.

Examples of context-free grammars are given in Table 7.2, some of which violate the Sequitur constraints. In the following, the grammar built from the context symbols is called the *main grammar* of Omnisc'IO. Sequitur has a linear worst-case complexity both in space and in time.

Table 7.2: Examples of context-free grammars. Lowercase letters represent terminal symbols, while uppercase letters represent rules and their instances. Example 1 is correct from a Sequitur perspective. Example 2 violates the rule utility (rule A is used only once; it should be deleted an its only instance should be replaced with its content). Example 3 violates the digram uniqueness (digram ab appears twice; a new rule $B \rightarrow ab$ can be created to replace it).

Example 1	**Example 2**	**Example 3**
$S \rightarrow abAAe$	$S \rightarrow abAe$	$S \rightarrow ababAAe$
$A \rightarrow cd$	$A \rightarrow cd$	$A \rightarrow cd$

Predictions Using the Grammar Model: Sequitur builds a grammar from a stream of symbols, but it does not predict the next incoming symbols from past observations. Therefore, we enriched the algorithm to be able to make such a prediction.

This improvement works by marking some of the terminal symbols in the grammar as *predictors*. This *predictor* characteristic is extended to non terminal symbols by using the following constraints:

Predictor nesting: A non terminal can be a predictor only if at least one of its nested symbols is a predictor.

Predictor utility: If symbol x (terminal or not) is a predictor in rule $Y \neq S$, then there exists at least one rule Z such that an instance of Y is a predictor in Z.

These constraints enforce that (1) if the grammar contains at least one predictor, then rule S contains at least one predictor, and (2) all the terminal predictors of the grammar can be reached from a predictor in S (proofs of these properties are trivial). The relations linking predictors together form a direct acyclic graph within the tree structure of the grammar. These two structural properties have to be carefully preserved when updating the grammar.

In order to make predictions from the set of predictors, two operations are defined, respectively, to update the set of predictors and to find new ones.

Updating predictors: We call *incrementing a predictor* the operation that consists of unmarking a symbol previously marked as a predictor and marking as a predictor the symbol that immediately follows it in the rule where it appears. *Updating predictors* consists of first unmarking all terminal predictors that did not correctly predict the last input, enforcing the predictor's constraints, and then incrementing all remaining terminal predictors. If a predictor is the last symbol of a rule, then non terminal predictors that reference it are incremented recursively. Examples of this operation are shown in Table 7.3, where predictor symbols are marked in red and underlined.

Discovering predictors: If all predictors have been removed because none of them correctly predicted the last input, a new step is necessary to rebuild a set of predictors. This step

Table 7.3: Predictors incrementation matching a given input. The predictors are marked in red and underlined. In the first input, a does not match and disappears from the set of predictors, c matches and is incremented to d, and A stays a predictor. In the second example, d matches but has no successor in rule A; thus A is incremented to e in rule S. The resulting models correspond to the grammars before the input is appended and Sequitur's constraints are applied.

Before Update	**Input**	**After Update**
$S \rightarrow \underline{a}bA\underline{A}e$	c	$S \rightarrow abA\underline{A}e$
$A \rightarrow \underline{c}d$		$A \rightarrow c\underline{d}$
$S \rightarrow abA\underline{A}e$	d	$S \rightarrow abAA\underline{e}$
$A \rightarrow c\underline{d}$		$A \rightarrow cd$

Table 7.4: Discovery of new predictors matching the last input (b, appended at the end of rule S). The predictors are marked in red and underlined. The symbol b becomes a predictor wherever it appears, and recursively any rule that leads to an occurrence of b becomes a predictor. The predictors are then updated to predict the next expected input (here c or e).

Before Discovery	**After Discovery**	**After Update**
$S \rightarrow abAAeb$	$S \rightarrow a\underline{b}A\underline{A}e\underline{b}$	$S \rightarrow ab\underline{AA}e\underline{b}$
$A \rightarrow cdb$	$A \rightarrow cd\underline{b}$	$A \rightarrow \underline{c}db$

is completed by navigating through the grammar and setting as predictors all symbols matching with the last symbol of rule S (after insertion of the last input). Parent rules are also set as predictors recursively wherever they appear. Note that the last symbol of rule S may be a non terminal, which forces new predictors to be searched only within the context of its corresponding rule and thus reduces the number of predictors and narrows down the prediction. An example of this operation is shown in Table 7.4. After the discovery of these new predictors, an *update* is necessary.

The *prediction* of the model corresponds to the set of terminal symbols marked as predictors after inserting a new input, updating the predictors, and enforcing the constraints. Although statistical methods could be used to weight each predicted symbol with a probability of appearance, considering equal probability for all predicted symbols appeared to be sufficient to achieve good results in our experiments.

To implement our algorithm, we reused the simple C++ code provided by the authors of Sequitur.[1] For comparison, the original code has 358 lines, whereas our improved version has 982 (without counting the I/O wrappers and the code related to access tables, which is explained later).

[1] http://www.sequitur.info/sequitur_simple.cc

Context-aware Access Behavior

The final step in Omnisc'IO is the actual bookkeeping of per-context access behavior. This is done differently for each type of tracked metrics.

Tracking Access Sizes: Access sizes are tracked on a per-context-symbol basis, so that predicting the next context symbol helps predicting the next access size. As will be shown in Section 7.3, most context symbols are always associated with the same access size each time they are encountered in an execution, making it easy to predict the exact size of the next accesses given a correct prediction of the next context symbols.

For the minority of symbols associated with several access sizes, Omnisc'IO keeps track of all access sizes encountered and builds a grammar from this sequence of sizes. The sizes constitute the terminal symbols of this grammar, which we call a *local size grammar*. The local size grammar associated with a context symbol is updated whenever the context symbol is encountered, and it evolves independently of the *main grammar* and independently of local size grammars attached to other symbols. It can then be used to make predictions of the size.

If the number of different access sizes observed for a given symbol is too large (typically larger than a configurable constant N), the local size grammar is replaced with simple average, minimum and maximum values that are updated whenever the context symbol is encountered. For our experiments, after analyzing the distributions of different access sizes per symbol, we choose $N = 24$.

More elaborate methods could be implemented to predict the access sizes for context symbols that exhibit apparently random sizes. We show in Section 7.3, however, that the three cases presented above have been sufficient to cover the behavior of all our applications.

Tracking Offsets: Many prefetching systems, including those implemented in the Linux kernel [139], are based on the assumptions of consecutive accesses; that is, the next operation is likely to start from the offset where the previous one ended. As we will show in our experiments, this assumption is held for the POSIX-based applications that we tested, but it fails for applications that use a higher-level library such as HDF5. Indeed such libraries often move the offset pointer backward or forward to write headers, footers, and metadata.

To predict the offset of the next operation, we define the notion of *offset transformation*. An offset transformation can (1) leave the offset as it was at the end of the previous operation (*consecutive access transformation*), (2) set it to a specific absolute value (*absolute transformation*), or (3) set it a value relative to the offset after the previous operation (*relative transformation*). Since it is not possible at low level to distinguish between absolute and relative transformations, Omnisc'IO uses absolute transformations only for operations that reset the offset to 0 (`open` and `close`). All other nonconsecutive offset transformations are considered relative to the previous offset.

Omnisc'IO associates transitions between context symbols with offset transformations the same way it associates context symbols with access sizes. For instance, if symbol B follows A in the execution and A left the offset at a value from which B starts, then the transition $A \rightarrow B$ is associated with a *consecutive access transformation*. When a transition encounters different types of offset transformations, Omnisc'IO builds a *local offset grammar* for the transition. Local offset grammars are the counterpart of local size grammars for offset

transformations. If the grammar associated with a transition grows too large (more than 24 symbols in our experiments), Omnisc'IO switches back to always predicting a *consecutive offset* transformation for this transition of context symbols.

Tracking Files Pointers: In order for the prediction of offsets to work properly, Omnisc'IO needs to know that two consecutive operations work on the same file or that an operation works on a new file or a file that has already been accessed earlier. This is particularly important when accesses to multiple files interleave. The prediction of files accessed is done by recording opened file pointers and associating transitions between symbols with changes of file pointers. Since in our experiments the case of interleaved accesses to different files did not appear, we will not study this particular aspect further.

Tracking Interarrival Times: To keep track of the time between the end of an operation and the beginning of the next one, Omnisc'IO uses a table associating transitions between context symbols with statistics on the measured time. These statistics include the minimum and maximum observed, the average, and the variance. We prefer these statistics rather than keeping only the average because they represent the minimum required to answer (1) whether an operation will immediately follow (maximum, minimum, average, and variance close to 0); (2) whether the next operation will follow in a predictable amount of time (maximum, minimum, and average close to each other, small variance); or (3) whether the time before the next operation is more unpredictable or depends on parameters that are not captured by our system (large minimum-maximum interval, large variance). Thus, these statistics, while minimal, are able to give us confidence in the predicted interarrival time, which may be important in the context of scheduling, for example.

To quickly react to changes in interarrival times, Omnisc'IO also keeps a *weighed interarrival average time*, updated every time the transition between symbols is encountered by using the following formula,

$$\widehat{T}^{weighed}_{x\to y} \leftarrow \frac{\widehat{T}^{weighed}_{x\to y} + T}{2}, \tag{7.1}$$

where $x \to y$ is the observed transition between context symbols x and y and T is the measured interarrival time. This weighed average is more efficient at making predictions of interarrival time, especially in a context where this interarrival time varies a lot between different observations of the same transition.

Overall Prediction Process and API

At each operation, Omnisc'IO updates its models (the main grammar and the tables of access sizes, offset transformations and interarrival times). It then updates its predictors and builds the set of possible next context symbols (this set often consists of a single prediction). From these possible next symbols, a set of triplets *(size, offset, date)* is formed that can be used by scheduling, prefetching, or caching systems. Although this kind of prediction can easily be extended to the series of N next I/O operations, our experiments will focus only on the capability of Omnisc'IO to predict the next one.

Application	Field	I/O Method	I/O Behavior
CM1	Climate	HDF5+POSIX	One file per process, same I/O behavior in each process, same domain size per process.
		HDF5+MPI-I/O	
		HDF5+Gzip	
GTC	Fusion	POSIX	One file per node per iteration, number of particles varies.
Nek5000	Fluid Dynamics	POSIX	I/O phase executed by rank 0 after a reduce phase.
LAMMPS	Molecular Dynamics	POSIX	I/O phase executed by rank 0 after a reduce phase.

Table 7.5: List of applications used in our experiments and their I/O backends.

To use Omnisc'IO, any software aiming at optimizing I/O simply needs to be linked against the Omnisc'IO library and to call the following function:

```
int omniscio_next(omniscio_req** prediction, int* n)
```

This function allocates the `prediction` array and fills it with a set of predicted request structures (including the size, offset, and date) representing the expected next I/O accesses.

7.3 Experimental Evaluation

In this section, we evaluate Omnisc'IO with real applications. We first assess its capability to predict the next context symbols, and we show how the grammar grows in size as the application continues to run. We then evaluate its performance in predicting the spatial and temporal characteristics of the next operations.

7.3.1 Platform and Applications

Grid'5000

All our experiments are carried out at the Nancy site of the French Grid'5000 testbed [6]. The applications run on *griffon*, a Linux cluster consisting of 92 Intel Xeon L5420 nodes (8 cores per node, 736 cores in total), using Mpich 3.0.4. The OrangeFS 2.8.7 parallel file system [12] is deployed on 12 nodes of the *graphene* cluster, which consists of Intel Xeon X3440 nodes. All nodes, including the file system's, are interconnected through a 20G InfiniBand network. Grid'5000 was selected because it gives us a complete control over the software stack. In particular, our experiments required the modify the code of Mpich to integrate Omnisc'IO, a modification that can hardly be done on a production machine.

Applications

The list of applications used is presented in Table 7.5. These applications are real-world codes representative of applications running on current supercomputers. They have been

used on NCSI's Kraken [63] and NCSA's Blue Waters [5] for CM1, ORNL's Titan [129] for GTC and LAMMPS, and ANL's Intrepid and Mira [83] for Nek5000.

We run these applications on 512 cores of Grid5000, except for Nek5000, which we run on 32 cores. These applications are written in Fortran except for LAMMPS (C++). Most of them use a POSIX I/O interface. To show the generality of our approach with respect to higher level I/O libraries, CM1 [8, 7] (already extensively used in Chapters 3 to 5) uses the HDF5 I/O library over the default (POSIX) I/O driver, as well as the MPI-I/O driver provided by pHDF5, and Gzip compression over the default POSIX driver. CM1 writes one file per process per I/O phase. The domain decomposition in CM1 is such that the amount of data remains the same over time and across processes. The use of compression exemplifies the case of varying data size in a nonvarying domain decomposition. GTC [43] writes one file per node per iteration, but the amount of data varies between files as particles move from one process to another. Like CM1, the domain decomposition in Nek5000 [98] does not vary over time, but the I/O phase is executed only by the rank 0 after a *reduce* phase. LAMMPS [66, 105] also sends data to the rank 0 process only. This process then writes each set of particles (of potentially different sizes) contiguously in a single file.

Although in CM1 and GTC all processes write data, we consider the results of Omnisc'IO only on process rank 0 (for applications that issue I/O from all processes, these results are in fact identical in all processes as they execute the same code and thus exhibit the same behavior). We first evaluate how well our algorithm manages to predict future context symbols based on past observations. We then evaluate the ability of Omnisc'IO to predict the location (offset and size in the file) of the next I/O operations. We also evaluate its ability to predict when future accesses will happen.

7.3.2 Experiments

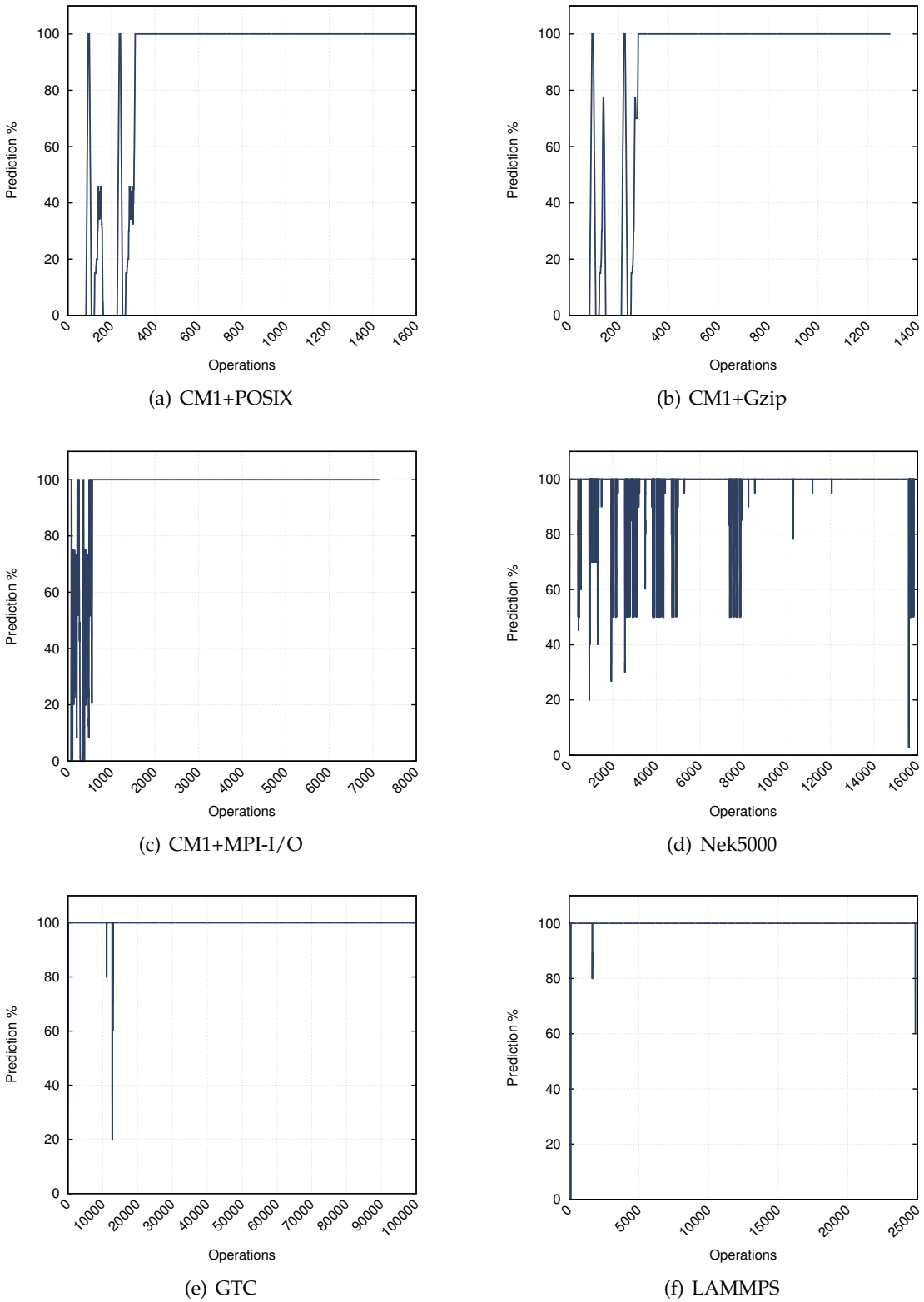
Our experiments consist of running each application, and at each I/O operation use Omnisc'IO to predict the characteristics of the next operation, that is, the date, the location and size, and the symbol. This prediction is then compared with the observed next operation.

7.3.3 Results Discussion

Context Prediction

Since Omnisc'IO is based on predicting context symbols, our first analysis aim to show how well it performs this task. We use a sliding window of ten operations and report the percentage of correct predictions. When Omnisc'IO predicts several possible next symbols, they are weighed as $\frac{1}{\text{number of predicted symbols}}$. For instance if Omnisc'IO predicts that the next symbol will be either a or b and the real next symbol is b, then this prediction is weighed $\frac{1}{2}$. For CM1 and GTC, which run for long periods of time, we show only several iterations starting from the beginning of the run.

Results: Figure 7.2 shows the context prediction capabilities of Omnisc'IO for all six use-cases. For all three configurations of CM1 as well as for LAMMPS and GTC, Omnisc'IO converges to a perfect (steady 100%) prediction of symbols after the first iteration. The variation



(a) CM1+POSIX

(b) CM1+Gzip

(c) CM1+MPI-I/O

(d) Nek5000

(e) GTC

(f) LAMMPS

Figure 7.2: Context prediction capability of Omnisc'IO over the run of each application. Configurations (a), (b), (c), (e), and (f) exhibit a clear learning phase after which Omnisc'IO makes perfect predictions ((e) and (f) exhibit a drop of prediction at the end of the first iteration).

observed during the first iteration corresponds to the moment the grammar starts detecting the innermost loops.

Omnisc'IO seems to learn GTC's behavior (Figure 7.2(e)) fast: the reason is that GTC's I/O phase consists of a loop over all particles, which is easily modeled in the grammar after the first two particles are written out. The prediction quality drops at the end of the first iteration when Omnisc'IO does not predict the end of this loop and the file being closed. This mistake is never repeated in later iterations. The same pattern appears in LAMMPS.

The case of Nek5000 is more interesting: although it writes periodically the exact same amount of data, the grammar model does not converge as fast and as perfectly as the other applications. By investigating the code of Nek5000, we found that this is due to code branches that process data in a different way depending on its content and then write it in an identical manner, leading to the creation of several symbols that are actually interchangeable in the grammar. Moving the `write` call outside the branches would help remove this indeterminism. Because we claimed our solution works with no prior knowledge of the application and without the involvement of the application developer, we did not apply this code modification.

We also observe a drop in prediction quality at the end of the LAMMPS and Nek5000 runs. This drop is due to the final results being output in a section of the code different from the one used for the periodic checkpoints. Thus these symbols, which appear the first time at the end of the execution could not have been predicted by any model.

Cost of a Failed Prediction

A failed prediction leads to searching new predictors within the grammar, instead of simply updating existing ones. This operation is linear in the size of the grammar (number of symbols). A failed prediction also has an effect on the system that leverages the prediction. For instance, a prefetching algorithm would read unnecessary data and/or fail to read the data that is actually needed by the program. The real cost would therefore depend on how much the incorrect operation consumes resources that could be used more productively.

Grammar Size and Memory Footprint

We then estimated the memory footprint of our approach. This memory footprint, as will be explained, is mainly dependent on the size of the grammar, which we evaluate hereafter.

Results: Figure 7.3 shows the evolution of the size of the main grammar as a function of the number of operations. One can clearly distinguish a first *learning phase* during which Omnisc'IO discovers the behavior of the application. This phase corresponds to the first iteration (potentially preceded by an input phase). It is followed by a *stationary regime* during which the model is updated in a mostly logarithmic manner. All the applications considered here exhibit this logarithmic growth of the grammar size after the learning phase. GTC's grammar growth is logarithmic as well, but it exhibits a staircase pattern. This is due to a variable number of particles written at each checkpoint, which leads to a variable number of `writes` and thus prevents Omnisc'IO from grouping these writes into large rules. That said, after 100,000 accesses the grammar has only 450 symbols.

The memory footprint is directly linked to the size of the main grammar (a symbol in our implementation is a 100-byte C++ object, making the grammar consume 26 KB in the case of CM1+POSIX, for example), and the number of entries in the tables (one entry per symbol or per transition, accounting also for a few bytes. CM1+POSIX uses 198 symbols, for example). This part of the memory footprint does not increases after the learning phase. The memory footprint of Omnisc'IO is thus correlated mainly with the grammar size and does not exceed a few hundreds of kilobytes.

Prediction of Sizes

We analyzed how many different access sizes were associated with each context symbol. We found that the vast majority of symbols were associated with just one size, potentially different for each symbol (171 symbols out of 183 for CM1 using HDF5 are associated with one size, and similar numbers with GZIP and pHDF5, 12 out of 17 for GTC, and all 38 of them for Nek5000). LAMMPS had the most interesting distribution, with 123 symbols associated with a unique size (yet potentially different for each symbol), and one unique symbol associated with a different size at almost every appearance. This distribution is due to the fact that all n processes send their set of particles to the rank 0 process, which writes them into a file in n successive `write` calls. As the number of particles varies between processes and between checkpoints, this leads to the variation in observed sizes.

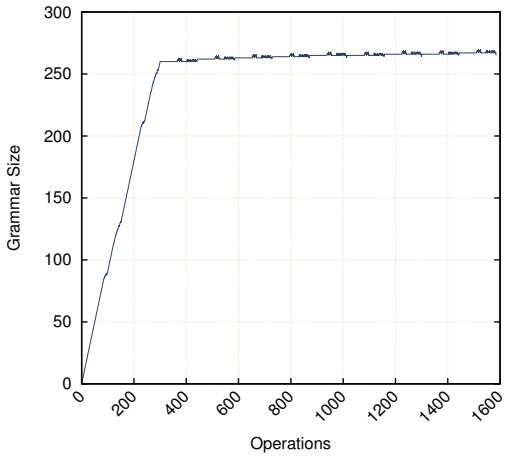
To evaluate the prediction of sizes, we use the relative error as a metric, as shown in Equation 7.2.

$$E_{size} = \frac{|size_p - size_o|}{size_o}, \tag{7.2}$$

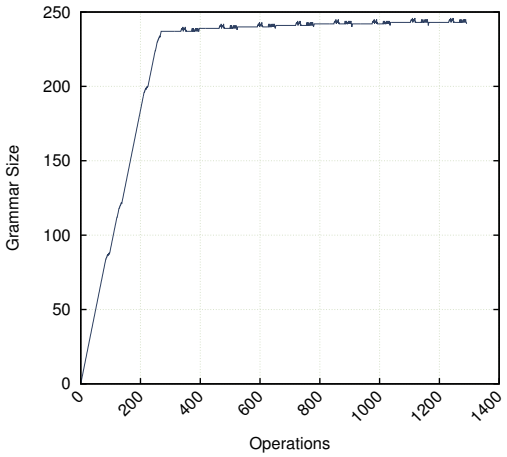
where $size_p$ is the predicted size and $size_o$ is the observed size. Intuitively, if the predictions are always such that $E_{size} \leq \epsilon$, then allocating $1 + \epsilon$ times the predicted size (in a caching system, for example) will always be enough to cover the need for the next operation. In a system like CALCioM, $1 + \epsilon$ times the predicted size represents an upper bound of the size expected to be accessed by a process. This information can be exchanged with other applications for a better estimation of their potential interference.

Results: Figure 7.4 shows the relative error observed for all six cases. In all but Nek5000, the error goes to 0 or close to 0 after the learning phase. Errors observed in Nek5000 match the incorrect predictions of context symbols. In LAMMPS, the prediction is very close but not equal to 0. The reason is that the number of particles written (and thus the size of each write) varies slightly from one write to another. Thus, after trying to build a local size grammar out of those random sizes, Omnisc'IO falls back to keeping track of the average only.

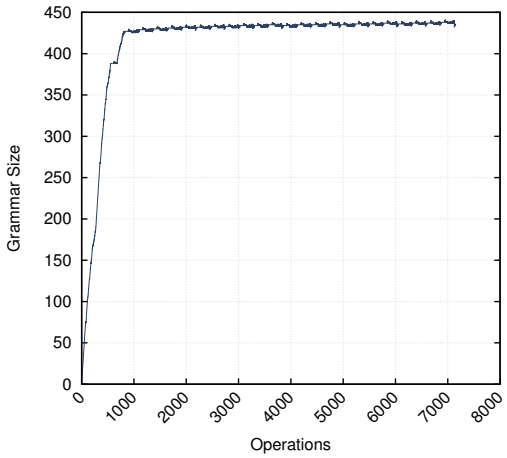
Note that the graphs are cut down to a maximum relative error of 5, whereas the observed errors can be of up to several thousands. For instance, if Omnisc'IO predicts a `write` or 5,000 bytes while the application actually writes only 2, the relative error is 2,499.



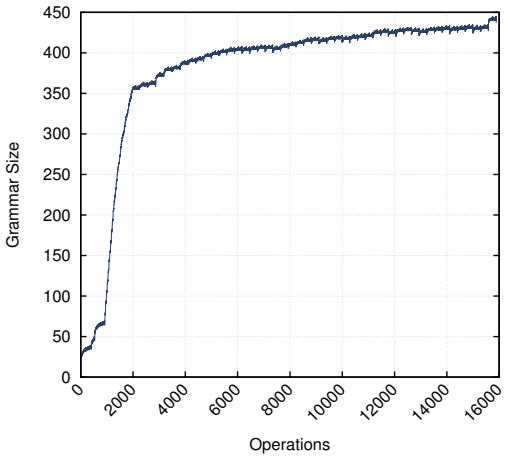
(a) CM1+POSIX



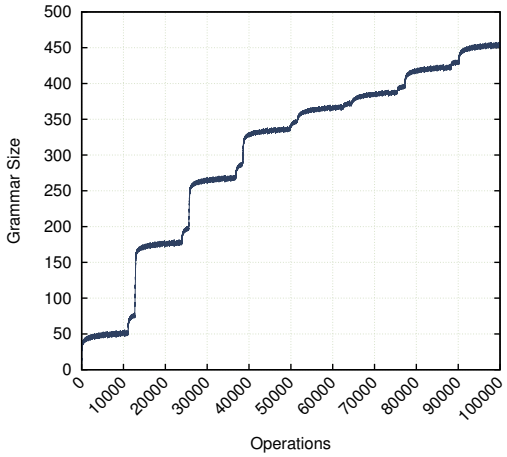
(b) CM1+Gzip



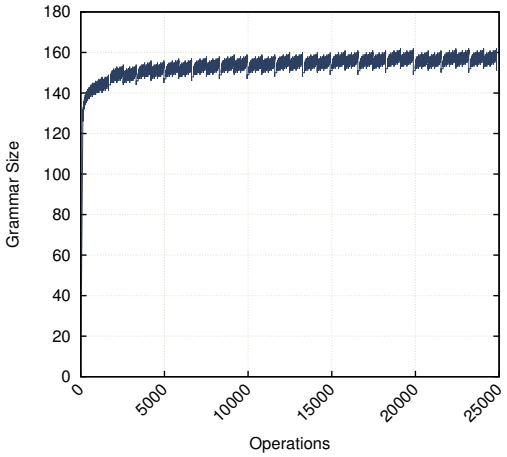
(c) CM1+MPI-I/O



(d) Nek5000

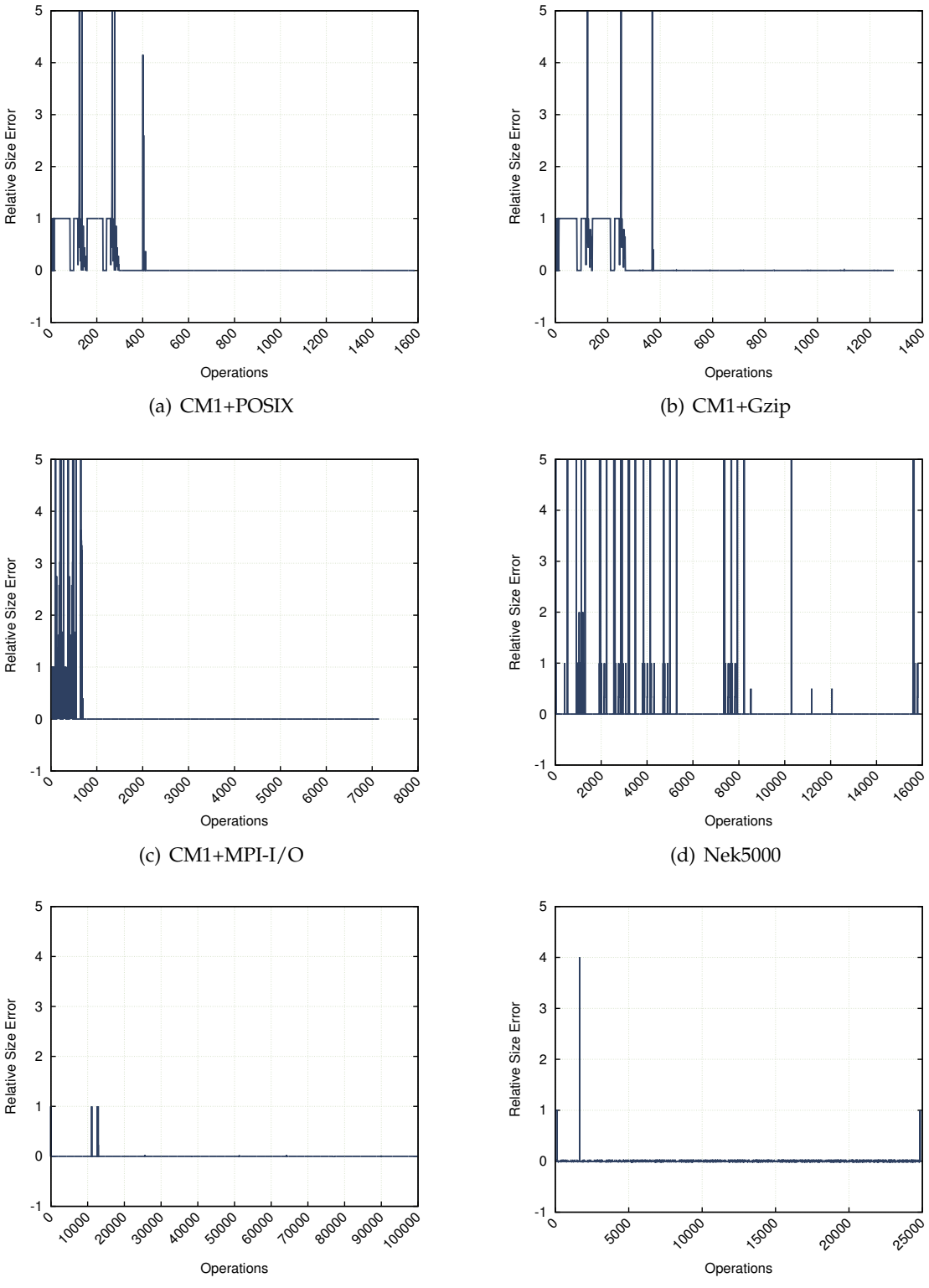


(e) GTC



(f) LAMMPS

Figure 7.3: Evolution of main grammar size (sum of the length of each rule, in number of symbols).



(a) CM1+POSIX

(b) CM1+Gzip

(c) CM1+MPI-I/O

(d) Nek5000

(e) GTC

(f) LAMMPS

Figure 7.4: Relative error in the prediction of access sizes in all simulations.

Table 7.6: Proportion of correct offset prediction using a naive *contiguous offsets* approach, and using Omnisc'IO, rounded to closest 0.1%.

Application	Contiguous Accesses	Omnisc'IO
CM1 (POSIX)	47.4%	92.2%
CM1 (Gzip)	53.2%	83.0%
CM1 (MPI-I/O)	72.7%	98.0%
Nek5000	99.4%	99.7%
GTC	99.9%	100%
LAMMPS	99.9%	100%

Prediction of Offsets

We consider that an offset prediction is either correct or incorrect. When our algorithm makes several predictions for the next context symbol (and therefore several predictions of offset), correct predictions are weighed accordingly. We compare our solution with the classical *contiguous access* estimation [139], which consists of always predicting that the next offset will follow the previous access.

Results: Table 7.6 shows the proportion of contiguous accesses in our set of applications as well as the proportion of correct predictions made by Omnisc'IO. In all cases, Omnisc'IO achieves a better prediction of offsets than does the naive approximation based on contiguous accesses. It is especially better suited when using a high-level I/O library such as HDF5 in CM1, since it manages to model and predict the portion of accesses that are non-contiguous. In particular, the prediction of offset in CM1 using HDF5 goes from 47.4%, when using a contiguous access estimation, to 92.2% with Omnisc'IO.

Hit Ratio

We also combined the prediction of sizes and offsets to measure how accurately our solution can predict the location of the next access. This information forms a predicted segment $S = [\![x_{start}, x_{end}]\!]$. The segment effectively accessed by the next I/O operation is denoted $S_0 = [\![y_{start}, y_{end}]\!]$. The *hit ratio* of S with respect to S_0, denoted $H(S|S_0)$, is computed using Equation 7.3.

$$H(S|S_0) = \begin{cases} \dfrac{100 \times |S \cap S_0|}{\max(x_{end}, y_{end}) - \min(x_{start}, y_{start})} \\ 100 \quad \text{if} \quad S = S_0 = \emptyset \end{cases} \tag{7.3}$$

This metrics yields the percentage of overlap between the two segments with respect to the distance between their extrema: $H(S|S_0) = 100 \iff S = S_0$. Since our approach may propose several potential next locations, this formula is extended to multiple segment $S_1 \dots S_n$ by considering the average of $H(S_i|S_0)$ for $i \in [\![1, n]\!]$.

Table 7.7: Average hit ratio achieved by Omnisc'IO, rounded to closest 0.1%.

Application	Hit Ratio
CM1 (POSIX)	84.6%
CM1 (Gzip)	79.5%
CM1 (MPI-I/O)	96.0%
Nek5000	98.6%
GTC	100%
LAMMPS	99.4%

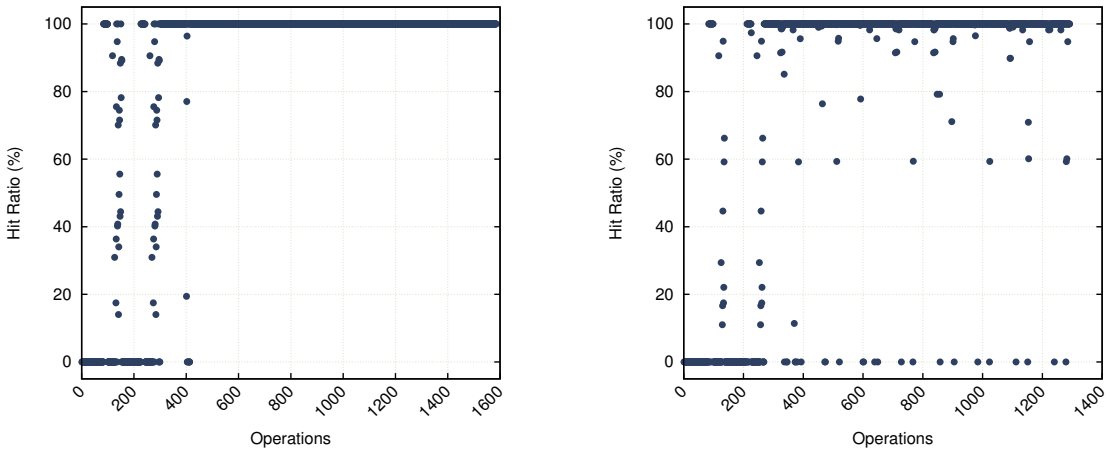
Results: Figure 7.5 shows the results obtained with our simulations, and Table 7.7 presents the average hit ratio over the course of the entire run for each application. Note that for CM1+POSIX and CM1+MPI-I/O, Omnisc'IO holds a perfect hit ratio after the learning phase. Although the hit ratio in LAMMPS also seems to be perfect, it is actually slightly lower than 100% because of the small error made in the prediction of the size (see earlier explanation in Section 7.3.3). The lowest hit ratio achieved in our experiments was that of CM1+Gzip (79.5%), which, considering the study made on the prediction of offsets and sizes in earlier sections, is explained mainly by incorrect predictions of offsets. Our guess is that HDF5 writes compressed data by blocks of predictable size but jumps back and forth in a more unpredictable manner to update metadata.

Temporal Prediction

Temporal prediction involves estimating the time between the end of an I/O operation and the beginning of the next one (interarrival time). We evaluate the temporal prediction capabilities of Omnisc'IO by computing the absolute difference between the predicted and the measured interarrival times.

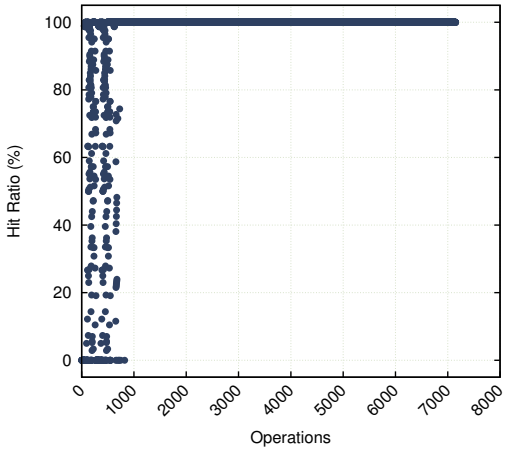
Results: For qualitative analysis, Figure 7.6 presents the series of observed interarrival times between consecutive operations, along with the predictions made by Omnisc'IO. We note that Omnisc'IO is efficient at discriminating *immediate transitions* (low transition times, which can be used as a hint that two consecutive operations belong to the same I/O phase) from *distant transitions*, (corresponding to computation and communication phases that last much longer).

Figure 7.7 presents the absolute difference between observed and predicted transition times on a logarithmic scale. For readability reasons, we consider only the 1,000 last operations of each run, that is, during the stationary regime. Table 7.8 reports the average of absolute difference over the course of each run (in its entirety, and not restricted to the stationary regime). We also compare the performance of Omnisc'IO with the *immediate reaccess* estimation used by some I/O schedulers (e.g., [147]), which consists in assuming that the next I/O operation is likely to immediately follow the current one (i.e., interarrival time are always estimated to 0) and use a time window during which a potential new operation is expected). In all situations, Omnisc'IO appears to be very good at predicting the interarrival time of I/O accesses. In particular, the average difference between the predicted and observed interarrival time is bellow a microsecond for LAMMPS, and at worst 0.199 seconds for CM1+Gzip, as opposed to 0.003 and 0.791 seconds, respectively, when considering an immediate reaccess estimation.

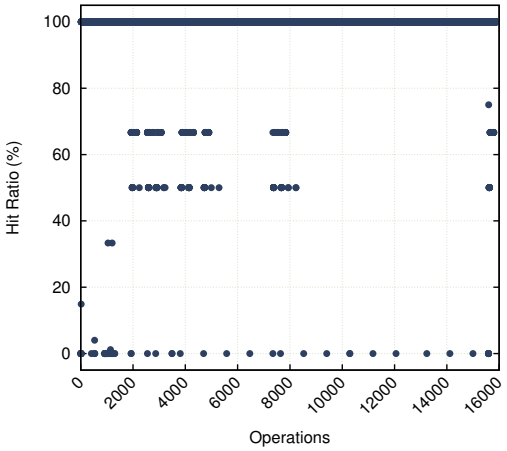


(a) CM1+POSIX

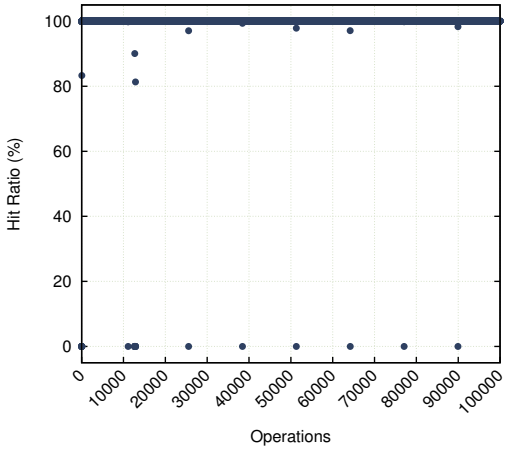
(b) CM1+Gzip



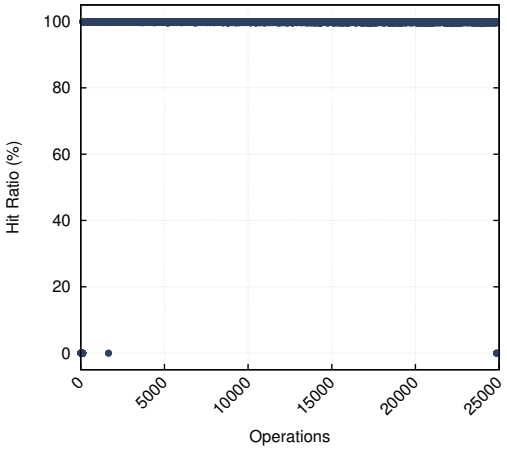
(c) CM1+MPI-I/O



(d) Nek5000

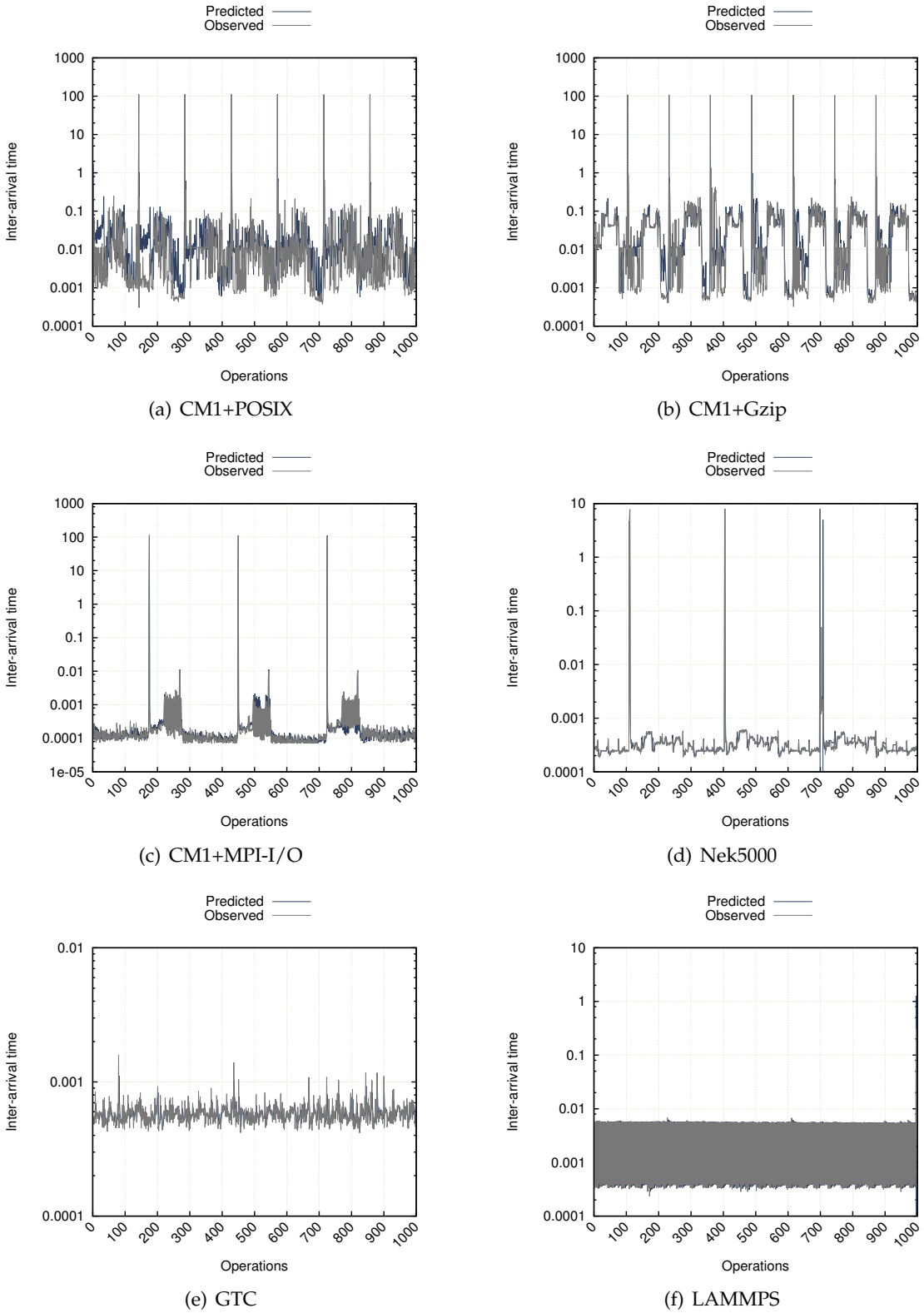


(e) GTC



(f) LAMMPS

Figure 7.5: Measurement of the hit ratio using Omnisc'IO to predict the location of the next accessed segment, as a function of the number of the number of operations completed.



(a) CM1+POSIX

(b) CM1+Gzip

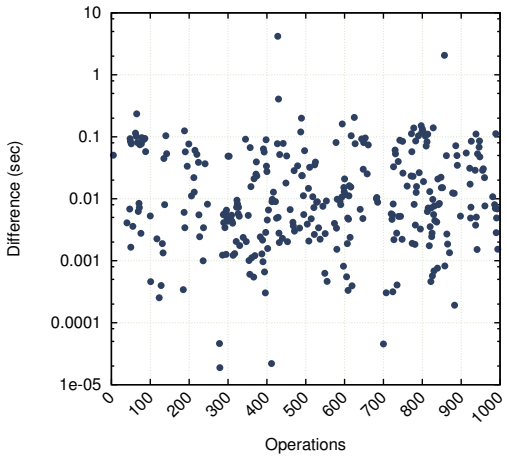
(c) CM1+MPI-I/O

(d) Nek5000

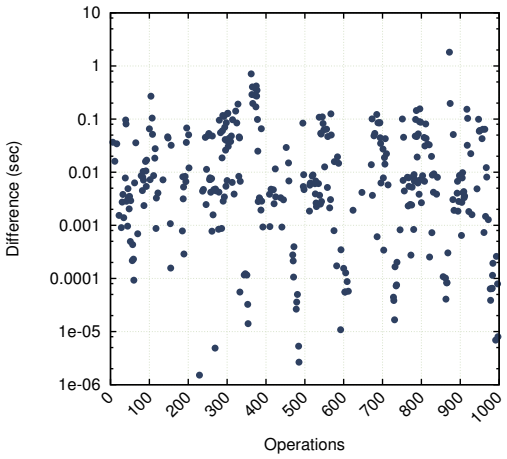
(e) GTC

(f) LAMMPS

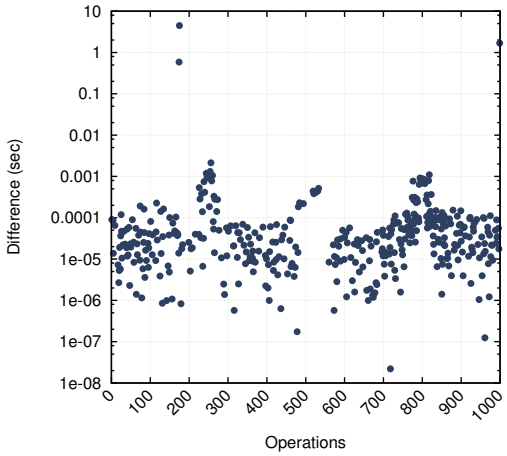
Figure 7.6: Matching between observed and predicted interarrival times of I/O events.



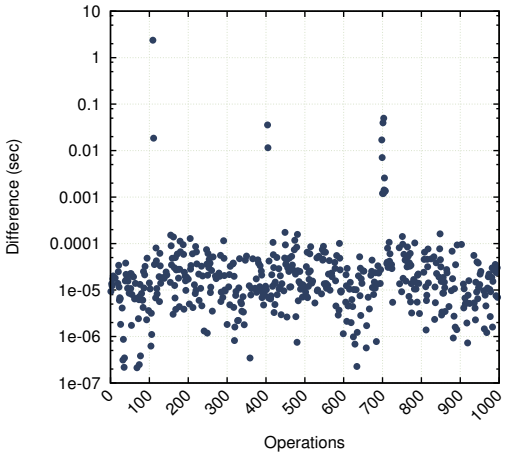
(a) CM1+POSIX



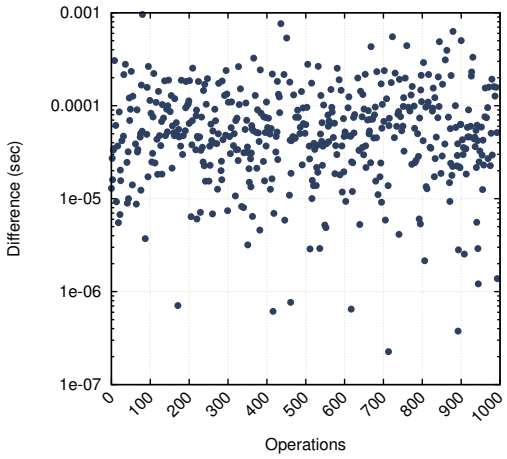
(b) CM1+Gzip



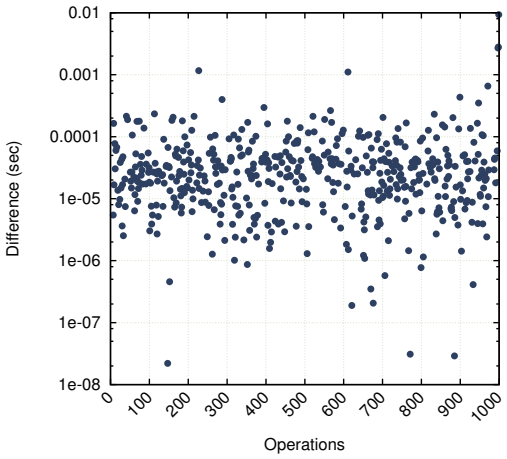
(c) CM1+MPI-I/O



(d) Nek5000



(e) GTC



(f) LAMMPS

Figure 7.7: Difference between predicted and observed interarrival times of I/O events.

Table 7.8: Average time difference between predicted and observed interarrival times (rounded to closest millisecond), and comparison with an *immediate re-access* estimation.

Application	Time Difference	Immediate Reaccess
CM1 (POSIX)	0.197 sec	0.735 sec
CM1 (Gzip)	0.199 sec	0.791 sec
CM1 (MPI-I/O)	0.060 sec	0.406 sec
Nek5000	0.012 sec	0.049 sec
GTC	0.001 sec	0.006 sec
LAMMPS	0.000 sec	0.003 sec

Table 7.9: Overhead of Omnisc'IO in the run time of each application (in microseconds per operation).

Application	Average Overhead	Std. dev.
CM1 (POSIX)	20.51 μsec	18.27 μsec
CM1 (Gzip)	20.20 μsec	15.56 μsec
CM1 (MPI-I/O)	19.95 μsec	14.50 μsec
Nek5000	23.44 μsec	18.96 μsec
GTC	19.03 μsec	27.79 μsec
LAMMPS	22.10 μsec	14.72 μsec

Note that combining the prediction of interarrival times and context symbols makes it possible to give an estimation of how many accesses will happen within a given time window and how many consecutive operations will occur before the end of the I/O phase. This type of information is especially important in the context of approaches like CALCioM, which needs to know *when* an application can be interrupted and *how many* accesses remain (together with the parameters of these accesses) before the end of an I/O phase.

Run-time Overhead

The run-time overhead of Omnisc'IO on a commodity hardware is presented in Table 7.9. This overhead of a few microseconds is negligible compared with the time taken by the I/O operations themselves (a few milliseconds to several seconds). However, since Omnisc'IO works at the level of atomic, contiguous operations, these I/O operations can be made asynchronous to hide the overhead of Omnisc'IO behind the I/O time.

7.3.4 Limitations of Our Approach

Like all systems, Omnisc'IO has limitations. As it leans on the repetitiveness of I/O patterns, any nonperiodic application (e.g., applications that write their results only once at the end of their run) will make Omnisc'IO incapable of discovering repetitive structures in the I/O pattern. To deal with such applications, however, Omnisc'IO can save its model into files and reload it before the next run.

As noted in Section 7.3.3, Omnisc'IO is sensitive to branches in the code that depend on the content of the data. It is arguably more difficult to solve this problem, since it would require for Omnisc'IO to know on which specific part of the entire simulation's data the branch depends.

7.4 Discussion and Related Work

This section discusses related work in the context of grammar-based modeling as well as spatial and temporal I/O prediction.

7.4.1 Grammar-based Modeling

The first work related to ours is Sequitur [92]. As explained above, Sequitur is designed to build a grammar from a sequence of symbols and has been used mainly in the area of text compression [62], but also natural language processing, music processing, and macromolecular sequence modeling [91]. The repetitive periodic I/O behavior of HPC applications [94] is a very good candidate application for Sequitur. To our knowledge, our approach is the first to take advantage of a grammar-based model to not only modeling but also making real-time predictions (through improvements of the Sequitur algorithm) of the application's I/O pattern.

7.4.2 I/O Patterns Prediction

Spacial and temporal I/O access prediction is a challenge commonly addressed in the context of prefetching, caching, and scheduling. Prefetching and caching indeed require a prediction of the location of future accesses [139], while I/O scheduling leverages estimations of I/O requests' interarrival time. While these domains have been investigated for decades in the context of commodity computers [70], we restrict our study of related works mostly to their use in the HPC area, where applications have different (mostly more regular) I/O behavior.

Spatial Predictions

Most of the work on spatial I/O patterns predication is done to assist I/O prefetching using various approaches, including Markov models [94], speculative execution [15], and knowledge accumulation [47]. These studies require either prior knowledge of the application or several runs before the model converges. Moreover, the predictions were evaluated by mean of performance improvements in a particular context such as prefetching. Our work focuses on providing a general approach that can predict both spatial and temporal I/O patterns of any HPC applications, at run time. Its evaluation focuses on its prediction capability, and our results can therefore be transferred to any of the aforementioned applications.

Kroeger and Long [64] studied several spatial access pattern modelings techniques, some of which are inspired by text compression algorithms such as variants of PPM (prediction by partial matching). The contexts (or symbols) used in these models are parameters of system-level I/O calls (i.e., file name, offset, size, etc.). Our solution builds a model of the program's structure using backtraces and keeps statistics only on the access parameters. Moreover, it can predict *when* the next operations are going to happen, which the aforementioned approaches cannot do.

Gniady et al. [41] also use stack frames to optimize the prediction of disk accesses, using existing pattern prediction techniques in the operating system. Their solution is used to improve caching. We designed our own prediction model based on the Sequitur algorithm that

builds a grammar-based model of the behavior of the application, and we applied this model not only to predict spatial access patterns, but also interarrival times of I/O operations.

Madhyastha and Reed [80] use artificial neural networks (ANNs) and hidden Markov models (HMMs) to classify access patterns in order to improve adaptive file systems. In their paper, the authors show that ANNs are incapable of predicting future access patterns, while HMMs need to be trained by using access patterns from several previous executions. Again, the challenge of predicting *when* future accesses will occur is not addressed. Our solution based on grammar models is able to converge at run time without prior execution of the application and can make predictions of both spatial and temporal access patterns.

Closer to our approach is the work by He et al. [48], who propose an approach to spatial I/O pattern detection to improve metadata indexing in PLFS. Their approach considers a sequence of *(offset,size)* access parameters and tries to find repetitive patterns in the differences between consecutive accesses, using a method inspired by the LZ77 sliding window algorithm. They also apply their algorithm to pattern-aware prefetching. While Omnisc'IO targets the same goal, it differs in the underlying algorithm used (Sequitur-based versus LZ77-inspired). Our approach also leverages stack traces to build a model of the program's behavior, whereas the solution proposed by He et al. works on the sequence of *(offset,size)* pairs.

Temporal Prediction and Scheduling

Prediction of temporal access pattern has been investigated by Tran and Reed [131] using ARIMA time series to model inter-arrival time between I/O requests. While the authors propose a solution that builds the model at run time, such statistical models need a large number of observations in order to converge to a good representation and, thus, good predictions. While ARIMA-based methods are effective at file system level when no knowledge can be retrieved from the application, we have shown that accurate predictions of interarrival times are possible at the application level without the need for such stochastic methods.

Byna et al. [9] propose a notation called I/O signatures to assist I/O prefetching. I/O signatures describe the historic access pattern including the spatiality, request size, repetitive behavior, temporal intervals, and type of I/O operation. I/O signatures are stored persistently and can be used only in later runs. Therefore contrary to Omnisc'IO, their approach requires at least a first run to retrieve some knowledge about the application. Additionally, this a priori knowledge becomes useless if the application's configuration or its scale changes.

Zhang et al. [146] couple I/O schedulers with process schedulers on compute nodes. When an application enters an I/O phase, it spawns new processes that pre-execute the code in order to find future I/O accesses while the main processes are waiting for the first access to complete. The knowledge of future accesses is then leveraged by the main processes. Considering the trend toward smaller operating systems with only restricted features, this kind of approach is likely not to be applicable in future machines with no preemptive process scheduler. Besides, running multiple processes in a single core is likely to increase run-time variability.

Several schedulers have been proposed that leverage some knowledge from the applications. The network request scheduler from Qian et al. [109], built in Lustre [22], associates

deadlines to requests. A similar design is proposed by Song et al. [120]. These schedulers are not based on any prediction, however, and could be greatly improved by knowledge extracted by Omnisc'IO on future access patterns. This knowledge can indeed help decide which application should be given priority to access the file system given its future access pattern. The scheduler proposed by Lebre et al. [69] aims at aggregating and reordering requests while trying to maintain fairness across applications, a task that would undoubtedly be easier with any kind of prediction of future incoming I/O requests.

7.5 Conclusion

The unprecedented scale of today's supercomputers forces researchers to consider new approaches to data management. In particular, self-adaptive and intelligent I/O systems that are capable of runtime analysis, modeling, and prediction of applications I/O behavior with little overhead and memory footprint will be of utmost importance to optimize prefetching, caching, or scheduling techniques.

7.5.1 Achievements of the Omnisc'IO Approach

Partially based on limitations of the scheduling and coordination approaches illustrated by CALCioM in Chapter 6, and their requirement for a component that transparently provides information on any application's I/O behavior, we have proposed an approach to I/O predictions based on formal grammar. This approach, called Omnisc'IO, precisely addresses the challenges of modeling and predicting the spatial and temporal access patterns of any HPC application. Omnisc'IO builds a model of I/O behavior using formal grammars. It is transparent to the application, has negligible overhead in time and memory, and converges at run time without prior knowledge of the application.

Our evaluation of Omnisc'IO with four real applications, in a total of six different scenarios, showed that Omnisc'IO converges quickly to a stable model capable of predicting both the date and location of future I/O accesses, achieving a near-perfect hit ratio (from 79.5% to 100% in our experiments) and interaccess time estimation (up to 0.199 sec of average absolute difference with the observed interaccess time).

7.5.2 Omnisc'IO as a Building Block for a Smart I/O Stack

Omnisc'IO represents a step toward the smart I/O stack presented in Chapter 2. In addition to complementing our CALCioM approach by extracting information on the I/O behavior, Omnisc'IO could be used by many other techniques including prefetching, caching and scheduling.

Additionally, Omnisc'IO produces a very compact and yet very precise model of an application's I/O behavior. This compact representation leveraging formal grammars makes it very suitable for a use in discrete event simulations of large-scale HPC storage systems.

These many possible use cases makes Omnisc'IO an excellent candidate as a building block for a smart I/O stack, that is, an I/O system that dynamically models, understands and adapts to the behavior of the applications that use it.

Finally, Omnisc'IO is a proof that, while I/O performance suffers from an increasing variability in post-Petascale machines, *the source of this I/O activity is predictable*. This regularity can thus be leveraged in the future to achieve better I/O performance, lower I/O variability, and maybe close the gap between computation and storage performance at Exascale.

Chapter 8

Conclusion and Perspectives

Contents

As HPC resources exceeding millions of cores become a reality, science and engineering codes invariably must be modified in order to efficiently exploit these resources. An important challenge in maintaining high performance is the presence of high variability in the effective I/O performance observed by large-scale simulations. This variability comes from different sources.

1. I/O contention between processes of a single application already affect the performance of the application and the variability of this performance. With large machines and larger simulations, it becomes necessary to find approaches that reduce the number of writers and optimize accesses to the parallel file system.

2. When coupling a simulation with a visualization software, in situ visualization tasks tend to negatively impact the performance of the simulation. This problem is further amplified in the context of interactive in situ visualization. It is arguably more difficult to solve because it involves multiple components (the simulation and the visualization software) that have to share resources efficiently without impacting each other.

3. Finally, I/O interference between different applications is an increasingly important problem, as we tend to run more and more concurrent applications on post-Petascale platforms. Contrary to the two previous sources of performance variability, I/O interference can hardly be solved simply through application-level optimizations.

Our work tackled all three aspects of this I/O performance variability in a number of contributions that we describe in the next section. These contributions also opened new research directions that are explained in the latest section.

8.1 Achievements

8.1.1 Using Dedicated Cores for Data Services in Large Scale Simulations

The Damaris approach: As a first step toward addressing the challenges of I/O variability in HPC simulations, we introduced the Damaris approach. Damaris leverages the increasing number of cores per node in recent supercomputers by proposing to dedicate one or a few cores in each node for data processing and I/O. It efficiently uses shared memory as a communication medium between cores running the simulation and dedicated cores. Additionally, its high-level, XML-based data description and its plugin system make Damaris a very flexible tool. An implementation of Damaris was developed that is subject to a registration at the APP (Agence de Protection des Programmes). As we finish this manuscript, this implementation reaches its version 1.0 with almost 20,000 lines of code. This implementation was used in several contributions of this manuscript.

Hiding the I/O variability with Damaris: First, we used Damaris to offload I/O tasks in dedicated cores, and compared the resulting performance with the two standard approaches to I/O in HPC simulations: the file-per-process and the collective I/O approaches. By gathering I/O operations in a reduced number of cores and by avoiding synchronization between these cores, Damaris was able to completely hide all I/O-related costs, and in particular the I/O variability. Our experiments using the CM1 atmospheric simulation on up to 9216 cores of the Kraken supercomputer showed that Damaris can achieve a 15 times higher throughput compared with the collective I/O approach. Damaris also dramatically reduces the application run time. Observing that dedicated cores still remain idle a large fraction of the time, we implemented several improvement, including an overhead-free data compression that achieved up to 600% compression ratio.

In situ visualization support in Damaris: We further leveraged the time spared by Damaris on dedicated cores by extending it to support in situ visualization. We evaluated our framework, called Damaris/Viz, with the CM1 atmospheric simulation and the Nek5000 computational fluid dynamic code, on the Grid'5000 and Blue Waters platforms. We showed that Damaris/Viz can fully hide the performance variability induced by in situ visualization tasks, even in scenarios involving interactions with a user. Besides, Damaris/Viz reduces visualization-related code modifications to a minimum in existing simulations. It also adapts to the needs of users through a unified connection with several existing visualization packages, including VisIt. Finally, it leverages the existing double-buffering techniques implemented in simulations to optimize its memory usage.

Investigating the tradeoffs between performance and energy consumption: Finally, Damaris served as a framework for studying the tradeoffs between performance and energy consumption in HPC simulations. We integrated in Damaris the option to use dedicated nodes instead of dedicated cores. We used the CM1 application on Grid'5000 to evaluate its performance and its energy consumption under various configurations of Damaris (time partitioning, dedicated cores and dedicated nodes), and various parameters of the architecture and the simulation, including the number of cores per node and the frequency of output. Based on these experiments, we formulated a model of the energy consumption for computation-intensive simulations when using different I/O approaches. Ou model achieved a 96.1% average accuracy. Its validation with the CM1 application showed that it can effectively guide the user toward the most energy-efficient configuration.

8.1.2 Addressing Cross-Application I/O interference

The main limitation of Damaris in solving the challenges of I/O interference is that it works at the level of a single application, or in the coupling between a simulation and a visualization software. With larger machines, more and more applications run concurrently and tend to interfere with one another when accessing a shared parallel file system. To specifically address this issue, we proposed the CALCioM approach.

The CALCioM approach: CALCioM consists of a communication layer across otherwise independent applications. This communication layer enables cross-application coordination to avoid I/O interference. In particular, we studied several coordination strategies that can be implemented within CALCioM, as well as ways to make a dynamic selection of the best strategy in order to optimize the efficiency of the full machine. CALCioM was evaluated on Grid'5000 and on ANL's Surveyor machines with the IOR benchmark. Our experiments showed that cross-application coordination can dramatically reduce the I/O interference. In particular, we were able to prevent a 14 times decrease of I/O performance for a small application in contention with a bigger one, and to substantially increase the machine wide efficiency through a dynamic selection of the best coordination strategy.

8.1.3 Predicting Spatial and Temporal I/O Patterns

The effective implementation of CALCioM raised the challenge of transparently predicting the spatial and temporal I/O patterns of any HPC simulation. To this end, we proposed the Omnisc'IO approach.

The Omnisc'IO approach: Omnisc'IO precisely aims to model and predict the spatial and temporal I/O patterns of any HPC simulation. It uses a grammar-based model built using an enhanced version of the Sequitur algorithm. It builds its model at run time with no prior knowledge of the application, and does not require any modification of the applications and I/O libraries. Our extensive experimental campaign using four application –CM1, GTC, Nek5000 and LAMMPS– in a total of six scenarios on Grid'5000, demonstrated that Omnisc'IO converges to a stable model in a couple of iterations only. This model is capable of predicting both the date and location of future I/O accesses, achieving a near-perfect hit

ratio (from 79.5% to 100% in our experiments) and interaccess time estimation (up to 0.199 sec of average absolute difference with the observed interaccess time).

Omnisc'IO constitutes a first step toward the design of a smart I/O stack, that is, an I/O stack that is capable of adapting to the applications' behavior and leverage knowledge about this behavior to improve its performance.

8.2 Prospects

Our present work naturally opens a number of perspectives. This section lists the most promising ones. We divide these perspectives in two sections: the first one groups potential contributions related to Damaris, while the second one lists the directions opened by our work on CALCioM and Omnisc'IO.

8.2.1 Prospects Related to the Damaris Approach

As an I/O approach, Damaris is relatively complete and has demonstrated its excellent capabilities in improving I/O performance while reducing I/O variability. Therefore, apart from engineering work to make Damaris even more flexible and adaptable, research perspectives related to Damaris focus on two aspects: improving its support for in situ visualization, and further studying the energy consumption of diverse I/O approaches. With respect to these aspects, we present hereafter two directions that we plan to investigate.

Smart In Situ Visualization with Damaris: Our study of in situ visualization using Damaris and CM1 revealed that in some simulations such as climate models, an important fraction of the data produced by the simulation does not actually contain any part of the phenomenon that are of interest to scientists. When visualizing this data in situ, it thus becomes possible to lower the resolution of non-interesting parts in order to increase the performance of the visualization process, an approach that we call "smart in situ visualization". Challenges to implement smart ISV include automatically discriminating relevant and non-relevant data within the simulation while this data is being produced. This detection should be made without user intervention and be fast enough to not diminish the overall performance of the visualization process. The plugin system of Damaris together with its existing connection with the VisIt visualization software provide an excellent ground to implement and evaluate smart ISV. This perspective is part of the PhD subject of Lokman Rahmani.[1]

Tradeoff Between Compression and Energy Consumption: Chapter 5 studied the tradeoff between performance and energy consumption in the context of Damaris. In Chapter 3 we have seen that the time spared by dedicated cores in Damaris can be leveraged to compress the data prior to storing it. An immediate question that can be asked is to which extent does compression in Damaris impacts this energy/performance tradeoff. On one hand, compression reduces the amount of data transferred and thus, the network traffic, which leads to lower energy consumption from data movements. On the other hand, compressing data

[1] PhD student at ENS Rennes / IRISA within the KerData team.

requires more computation time and higher energy consumption as a result of data movement in the local memory hierarchy. We can thus expect again a tradeoff between energy, performance and compression level.

8.2.2 Prospects Related to CALCioM and Omnisc'IO

The CALCioM and Omnisc'IO approaches both open thrilling research directions. CALCioM illustrated the unconventional idea that applications could communicate with one another in order to coordinate their I/O behavior and avoid interfering. Yet it remains to be evaluated with real applications at large scale. On the other hand, Omnisc'IO provides a very effective way to predict spatial and temporal behaviors, and remains to be applied in optimization techniques such as caching, prefetching, or scheduling. We highlight hereafter three main perspectives.

Coupling CALCioM and Omnisc'IO: Omnisc'IO provides predictions on the I/O behavior of applications. These predictions can benefit the effectiveness of CALCioM as well as its transparent integration into the I/O stack. Therefore, an important direction in the near future will be to couple the two in order to come closer to the notion of a "smart I/O stack", described in Chapter 2. The evaluation of such a coupling is however challenging. Indeed running and controlling several applications at the same time is a daunting task. It also requires to reserve an entire Petascale machine in order to prevent any interference with other users' applications. Fortunately we can leverage the models of I/O behaviors produced by Omnisc'IO to replay realistic interference scenarios in event-driven simulations. More generally we plan to use Omnisc'IO to improve caching and perfecting techniques.

Studying Communication Interference: Discussions with researchers from the University of Illinois at Urbana Champaign brought to our attention the fact that recent supercomputers like Blue Waters are not only subject to I/O interference, but also to interference in communications between the processes of the applications. It is indeed frequent to observe some applications impacting others because of their heavy collective communication patterns. One potential research perspective would thus consists of leveraging CALCioM not only to avoid I/O interference, but to prevent communication interference as well.

Cross-Application Interference and Energy Consumption: Last but not least, while we studied the energy consumption of different I/O approaches at the level of individual applications, the effect of cross-application interference on energy consumption remains to be investigated. With the 20 MW power budget set by DARPA as a target for Exascale machines, it would not be surprising in the near future to not only allocate CPU hours to projects, but also fixed energy budgets. While this would drive the adoption of energy-saving techniques, it would become necessary to investigate the effect of cross-application interference on the energy consumption so that projects with a small energy budget don't get unfairly impacted by large applications that interfere with them.

Bibliography

[1] H. Abbasi, J. Lofstead, F. Zheng, K. Schwan, M. Wolf, and S. Klasky. "Extending I/O Through High Performance Data Services". In: *Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER '09)*. New Orleans, Louisiana, USA, Sept. 2009. DOI: 10.1109/CLUSTR.2009.5289167.

[2] *ADIOS, Oak Ridge National Laboratory (ORNL)*. https://www.olcf.ornl.gov/center-projects/adios/.

[3] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. "Scalable I/O Forwarding Framework for High-Performance Computing Systems". In: *Proceedings of the IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER '09.* Sept. 2009. DOI: 10.1109/CLUSTR.2009.5289188.

[4] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey. "CA-NFS: a Congestion-Aware Network File System". In: *ACM Transactions on Storage (TOS)* 5.4 (2009), p. 15.

[5] *Blue Waters supercomputer, National Center for Supercomputing Applications (NCSA)*. http://www.ncsa.illinois.edu/BlueWaters/.

[6] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, et al. "Grid'5000: A Large Scale and Highly Reconfigurable Experimental Grid Testbed". In: *International Journal of High Performance Computing Applications (IJHPCA)* 20.4 (2006), p. 481. ISSN: 1094-3420.

[7] G. H. Bryan and J. M. Fritsch. "A Benchmark Simulation for Moist Nonhydrostatic Numerical Models". In: *Monthly Weather Review* 130.12 (2002), pp. 2917–2928. DOI: 10.1175/1520-0493(2002)130<2917:ABSFMN>2.0.CO;2.

[8] G. Bryan. *CM1*. http://www.mmm.ucar.edu/people/bryan/cm1/.

[9] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. "Parallel I/O Prefetching using MPI File Caching and I/O Signatures". In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '08)*. Nov. 2008. DOI: 10.1109/SC.2008.5213604.

[10] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. "24/7 Characterization of Petascale I/O Workloads". In: *Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER '09)*. IEEE. 2009.

[11] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. "Small-File Access in Parallel File Systems". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)* (2009). DOI: 10.1109/IPDPS.2009.5161029.

[12] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. "PVFS: a Parallel File System for Linux Clusters". In: *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*. Atlanta, Georgia: USENIX Association, 2000.

[13] E. V. Carrera, E. Pinheiro, and R. Bianchini. "Conserving Disk Energy in Network Servers". In: *Proceedings of the 17th annual ACM International Conference on Supercomputing (ICS '03)*. ACM. 2003, pp. 86–97.

[14] J Chen, A Choudhary, S Feldman, B Hendrickson, C. Johnson, R Mount, V Sarkar, V White, and D Williams. "Synergistic Challenges in Data-Intensive Science and Exascale Computing". In: *DOE ASCAC Data Subcommittee Report, Department of Energy Office of Science* (2013).

[15] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, and W. Gropp. "Exploring Parallel I/O Concurrency with Speculative Prefetching". In: *Proceedings of the 37th International Conference on Parallel Processing (ICPP '08)*. IEEE. 2008, pp. 422–429.

[16] C. M. Chilan, M Yang, A. Cheng, and L. Arber. "Parallel I/O Performance Study with HDF5, a Scientific Data Package". In: *TeraGrid 2006: Advancing Scientific Discovery* (2006).

[17] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, et al. "Extreme Scaling of Production Visualization Software on Diverse Architectures". In: *IEEE Computer Graphics and Applications* (2010), pp. 22–31. ISSN: 0272-1716.

[18] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp. "Non-Contiguous I/O through PVFS". In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER '02)*. Chicago, Illinois, USA: IEEE Computer Society, 2002, p. 405. ISBN: 0-7695-1745-5. DOI: 10.1109/CLUSTR.2002.1137773.

[19] P. M. Dickens and R. Thakur. "Evaluation of Collective I/O Implementations on Parallel Architectures". In: *Journal of Parallel and Distributed Computing (JPDC)* 61.8 (2001), pp. 1052–1076. ISSN: 0743-7315. DOI: 10.1006/jpdc.2000.1733.

[20] M. E. Diouri, G. L. T. Chetsa, O. Glück, L. Lefevre, J.-M. Pierson, P. Stolf, and G. Da Costa. "Energy Efficiency in High-Performance Computing With and Without Knowledge of Applications and Services". In: *International Journal of High Performance Computing Applications (IJHPCA)* 27.3 (2013), pp. 232–243.

[21] C. Docan, M. Parashar, and S. Klasky. "Enabling High-Speed Asynchronous Data Extraction and Transfer Using DART". In: *Concurrency and Computation: Practice and Experience* (2010), pp. 1181–1204. ISSN: 1532-0634. DOI: 10.1002/cpe.1567.

[22] S. Donovan, G. Huizenga, A. J. Hutton, C. C. Ross, M. K. Petersen, and P. Schwan. "Lustre: Building a File System for 1000-node Clusters". In: *Proceedings of the 2003 Linux Symposium*. Citeseer. 2003.

[23] M. Dorier. "SRC: Damaris - Using Dedicated I/O Cores for Scalable Post-Petascale HPC Simulations". In: *Proceedings of the International Conference on Supercomputing (ICS '11)*. ICS '11. Tucson, Arizona, USA: ACM, 2011, p. 370. ISBN: 978-1-4503-0102-2. DOI: 10.1145/1995896.1995953.

[24] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O". In: *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER '12)*. Beijing, China: IEEE, Sept. 2012.

[25] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. *Damaris: Leveraging Multicore Parallelism to Mask I/O Jitter*. English. Research Report RR-7706. INRIA, Apr. 2012, p. 36.

[26] M. Dorier, R. Sisneros Roberto, T. Peterka, G. Antoniu, and B. Semeraro Dave. "Damaris/Viz: a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework". In: *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV '13)*. Atlanta, Georgia, USA, Oct. 2013.

[27] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim. "CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination". English. In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '14)*. Phoenix, Arizona, USA, May 2014.

[28] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross. "Omnisc'IO: A Grammar-Based Approach to Spatial and Temporal I/O Patterns Prediction". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE, ACM. New Orleans, United States, Nov. 2014.

[29] M. Dreher, B. Raffin, et al. "A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations". In: *ACM/IEEE International Symposium on Cluster, Cloud and Grid Computing (CCGrid '14)* (2014).

[30] M. Dreher, J. Prevoteau-Jonquet, M. Trellet, M. Piuzzi, M. Baaden, B. Raffin, N. Férey, S. Robert, and S. Limet. "Exaviz: A Flexible Framework to Analyse, Steer and Interact with Molecular Dynamics Simulations". In: *Faraday Discussions* (2014).

[31] D. Ellsworth, B. Green, C. Henze, P. Moran, and T. Sandstrom. "Concurrent Visualization in a Production Supercomputing Environment". In: *IEEE Transactions on Visualization and Computer Graphics (TVGC)* 12.5 (2006), pp. 997–1004. ISSN: 1077-2626. DOI: 10.1109/TVCG.2006.128.

[32] A. Esnard, N. Richart, and O. Coulaud. "A Steering Environment for Online Parallel Visualization of Legacy Parallel Simulations". In: *Proceedings of the IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications (DS-RT '06)*. IEEE. 2006, pp. 7–14.

[33] *EzViz, ERDC DSRC*. http://daac.hpc.mil/software/ezViz/.

[34] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. Jansen. "The ParaView Coprocessing Library: A Scalable, General Purpose In Situ Visualization Library". In: *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV '11)*. 2011.

[35] M. Folk, A. Cheng, and K. Yates. "HDF5: A File Format and I/O Library for High Performance Computing Applications". In: *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '99)*. 1999.

[36] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir. *Scheduling the I/O of HPC Applications under Congestion*. Rapport de recherche RR-8519. INRIA, Apr. 2014.

[37] M. Gamell, I. Rodero, M. Parashar, J. C. Bennett, H. Kolla, J. Chen, P.-T. Bremer, A. G. Landge, A. Gyulassy, P. McCormick, S. Pakin, V. Pascucci, and S. Klasky. "Exploring Power Behaviors and Trade-offs of In-situ Data Analytics". In: *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. Denver, Colorado: ACM, 2013. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503303.

[38] L. Ganesh, H. Weatherspoon, M. Balakrishnan, and K. Birman. "Optimizing Power Consumption in Large Scale Storage Systems". In: *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS '07)*. San Diego, California, USA: USENIX Association, 2007.

[39] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. "MPI-2: Extending the Message-Passing Interface". In: *Proceedings of the International Conference on Parallel Processing (Euro-Par '96)*. Springer. 1996, pp. 128–135.

[40] E. Gibney. *Model Universe Recreates Evolution of the Cosmos*. http://www.nature.com/news/model-universe-recreates-evolution-of-the-cosmos-1.15178. News. 2014.

[41] C. Gniady, A. R. Butt, and Y. C. Hu. "Program-counter-based Pattern Classification in Buffer Caching". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI '04)*. San Francisco, California, USA: USENIX Association, 2004.

[42] *Green500 List of Supercomputers*. http://www.green500.org/.

[43] *GTC version 1, Plasma Theory Group, UCIrvine*. http://phoenix.ps.uci.edu/GTC/.

[44] J. Hamilton. *Cost of Power in Large-Scale Data Centers*. http://perspectives.mvdirona.com/2008/11/28/CostOfPowerInLargeScaleDataCenters.aspx. Nov. 2008.

[45] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. "The IBM Blue Gene/Q Compute Chip". In: *Micro, IEEE* 32.2 (2012), pp. 48–60.

[46] A. Harmon. *Breakthrough Simulation: A Supercell Producing a Long-Track EF5 Tornado*. http://www.isgtw.org/feature/breakthrough-simulation-supercell-producing-long-track-ef5-tornado, viewed on July 2014. 2014.

[47] J. He, X.-H. Sun, and R. Thakur. "KNOWAC: I/O Prefetch via Accumulated Knowledge". In: *Proceedings of the 2012 IEEE International Conference on Cluster Computing (CLUSTER '12)*. Washington, DC, USA: IEEE Computer Society, 2012. ISBN: 978-0-7695-4807-4. DOI: 10.1109/CLUSTER.2012.83.

[48] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun. "I/O Acceleration with Pattern Detection". In: *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '13)*. ACM. 2013, pp. 25–36.

[49] M. Hereld, M. E. Papka, and V. Vishwanath. "Toward Simulation-Time Data Analysis and I/O Acceleration on Leadership-Class Systems". In: *Proceeding of the IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV '11)*. Providence, RI, 2011.

[50] *Hierarchical Data Format HDF5*. http://www.hdfgroup.org/HDF5/.

[51] A. Hoisie and V. Getov. "Extreme-Scale Computing - Where 'Just More of the Same' Does Not Work". In: *Computer* 42.11 (Nov. 2009), pp. 24–26. ISSN: 0018-9162. DOI: 10.1109/MC.2009.354.

[52] J. Hruska. *DARPA Summons Researchers to Reinvent Computing*. http://www.extremetech.com/computing/116081-darpa-summons-researchers-to-reinvent-computing, viewed on June 2014. 2012.

[53] *INRIA Grid'5000*. http://www.grid5000.fr.

[54] *Intrepid supercomputer, Argonne National Laboratory*. https://www.alcf.anl.gov/intrepid.

[55] F. Isaila, J. Garcia, J. Carretero, R. B. Ross, and D. Kimpe. "Making the Case for Reforming the I/O Software Stack of Extreme-Scale Systems". In: *Preprint ANL/MCS-P5103-0314, Argonne National Laboratory* (2014).

[56] F. Isaila, J. G. Blas, J. Carretero, R. Latham, and R. Ross. "Design and Evaluation of Multiple Level Data Staging for Blue Gene Systems". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2010). ISSN: 1045-9219. DOI: 10.1109/TPDS.2010.127.

[57] C. Johnson, S. Parker, C. Hansen, G. Kindlmann, and Y. Livnat. "Interactive Simulation and Visualization". In: *Computer* 32.12 (1999), pp. 59–65.

[58] D. B. Johnston. *First-of-a-kind supercomputer at Lawrence Livermore available for collaborative research*. https://www.llnl.gov/news/newsreleases/2014/May/NR-14-05-02.html. News Release. 2014.

[59] S. Kamil, J. Shalf, and E. Strohmaier. "Power Efficiency in High Performance Computing". In: *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, (IPDPS '08)*. IEEE. 2008.

[60] N. Kappiah, V. W. Freeh, and D. K. Lowenthal. "Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs". In: *Proceedings of the 2005 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '05)*. IEEE Computer Society. 2005.

[61] C. Karakoyunlu and J. Chandy. "Techniques for an Energy Aware Parallel File System". In: *Proceedings of the International Green Computing Conference (IGCC '12)*. June 2012. DOI: 10.1109/IGCC.2012.6322247.

[62] J. Kieffer and E. hui Yang. "Grammar-Based Codes: A New Class of Universal Lossless Source Codes". In: *IEEE Transactions on Information Theory* 46.3 (May 2000), pp. 737–754. ISSN: 0018-9448. DOI: 10.1109/18.841160.

[63] *Kraken supercomputer, National Institute for Computational Sciences (NICS)*. http://www.nics.tennessee.edu/computing-resources/kraken.

[64] T. Kroeger and D. Long. "The Case for Efficient File Access Pattern Modeling". In: *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS '99)*. 1999, pp. 14–19. DOI: 10.1109/HOTOS.1999.798371.

[65] D. Q. Lamb. *Supercomputers Can Save U.S. Manufacturing*. http : / / www . scientificamerican . com / article / big - computers - for - little/, viewed on May 2014. Mar. 2012.

[66] *LAMMPS, Sandia National Laboratory*. http://lammps.sandia.gov/.

[67] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. "I/O Performance Challenges at Leadership Scale". In: *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '09)*. Portland, Oregon, USA: ACM, 2009. ISBN: 978-1-60558-744-8. DOI: 10 . 1145 / 1654059 . 1654100.

[68] J. H. Laros III, K. T. Pedretti, S. M. Kelly, W. Shu, and C. T. Vaughan. "Energy Based Performance Tuning for Large Scale High Performance Computing Systems". In: *Proceedings of the 2012 Symposium on High Performance Computing (HPC '12)*. Orlando, Florida, USA: Society for Computer Simulation International, 2012. ISBN: 978-1-61839-788-1.

[69] A. Lebre, G. Huard, Y. Denneulin, and P. Sowa. "I/O Scheduling Service for Multi-Application Clusters". In: *Proceedings of the IEEE International Conference on Cluster Computing and Workshops (CLUSTER '06)*. Sept. 2006.

[70] H.-Y. Li, C. S. Xie, and Y. Liu. "A New Method of Prefetching I/O Requests". In: *Proceedings of the International Conference on Networking, Architecture, and Storage, (NAS '07)*. July 2007, pp. 217–224. DOI: 10.1109/NAS.2007.3.

[71] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. "Parallel netCDF: A High-Performance Scientific I/O Interface". In: *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '03)*. IEEE. 2003. ISBN: 1581136951.

[72] M. Li, S. Vazhkudai, A. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman. "Functional Partitioning to Optimize End-to-End Performance on Many-Core Architectures". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society. 2010.

[73] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. "Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '06)*. IEEE. 2006.

[74] LLNL. *Getting Data Into VisIt, https://wci.llnl.gov/codes/visit/manuals.html*. 2006.

[75] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf. "Managing Variability in the IO Performance of Petascale Storage Systems". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. Washington, DC, USA: IEEE Computer Society, 2010. ISBN: 978-1-4244-7559-9. DOI: 10.1109/SC.2010.32.

[76] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)". In: *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. CLADE '08. Boston, MA, USA: ACM, 2008. ISBN: 978-1-60558-156-9. DOI: 10.1145/1383529.1383533.

[77] K.-L. Ma. "In Situ Visualization at Extreme Scale: Challenges and Opportunities". In: *IEEE Computer Graphics and Applications* 29.6 (2009), pp. 14–19. ISSN: 0272-1716. DOI: 10.1109/MCG.2009.120.

[78] K.-L. Ma, C. Wang, H. Yu, and A. Tikhonova. "In-situ Processing and Visualization for Ultrascale Simulations". In: *Journal of Physics: Conference Series* 78.1 (2007).

[79] X. Ma, J. Lee, and M. Winslett. "High-Level Buffering for Hiding Periodic Output Cost in Scientific Simulations". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 17 (2006), pp. 193–204. ISSN: 1045-9219. DOI: 10.1109/TPDS.2006.36.

[80] T. Madhyastha and D. Reed. "Learning to Classify Parallel Input/Output Access Patterns". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 13.8 (2002), pp. 802–813. ISSN: 1045-9219. DOI: 10.1109/TPDS.2002.1028437.

[81] P. Malakar, V. Natarajan, and S. S. Vadhiyar. "An Adaptive Framework for Simulation and Online Remote Visualization of Critical Climate Applications in Resource-constrained Environments". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. Washington, DC, USA: IEEE Computer Society, 2010. ISBN: 978-1-4244-7559-9. DOI: 10.1109/SC.2010.10.

[82] *MatplotLib*. http://matplotlib.org/.

[83] *Mira supercomputer, Argonne National Laboratory*. http://www.alcf.anl.gov/mira.

[84] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. "Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. Los Alamitos, CA, USA: IEEE Computer Society, 2010. ISBN: 978-1-4244-7559-9. DOI: 10.1109/SC.2010.18.

[85] K. Moreland, R. Oldfield, P. Marion, S. Jourdain, N. Podhorszki, V. Vishwanath, N. Fabian, C. Docan, M. Parashar, M. Hereld, et al. "Examples of In Transit Visualization". In: *Proceedings of the 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities (PDAC '11)*. ACM. 2011.

[86] *MPI Standard, MPI Forum*. http://www.mpi-forum.org/.

[87] *Mpich*. http://www.mpich.org.

[88] R. Nathuji, A. Kansal, and A. Ghaffarkhah. "Q-clouds: Managing Performance Interference Effects for QoS-Aware Clouds". In: *Proceeding of ACM European Conference on Computer Systems (EuroSys '10)*. Apr. 2010, pp. 237–250.

[89] *Nautilus supercomputer, National Institute for Computational Sciences (NICS)*. http://www.nics.tennessee.edu/computing-resources/nautilus.

[90] *NetCDF, Unidata*. http://www.unidata.ucar.edu/software/netcdf/.

[91] C. G. Nevill-Manning. "Inferring Sequential Structure". PhD thesis. 1996.

[92] C. G. Nevill-Manning and I. H. Witten. "Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm". In: *Journal of Artificial Intelligence Research* 7.1 (Sept. 1997), pp. 67–82. ISSN: 1076-9757.

[93] A. Nisar, W. keng Liao, and A. Choudhary. "Scaling Parallel I/O Performance Through I/O Delegate and Caching System". In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '08)*. 2008. DOI: 10.1109/SC.2008.5214358.

[94] J. Oly and D. A. Reed. "Markov Model Prediction of I/O Requests for Scientific Applications". In: *Proceedings of the 16th ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '02)*. ACM. 2002, pp. 147–155.

[95] *OpenMP*. http://openmp.org/.

[96] A.-C. Orgerie, M. D. d. Assuncao, and L. Lefevre. "A Survey on Techniques for Improving the Energy Efficiency of Large-Scale Distributed Systems". In: *ACM Computing Surveys (CSUR)* 46.4 (2014), p. 47.

[97] B. O'shea, G. Bryan, J. Bordner, M. Norman, T. Abel, R. Harkness, and A. Kritsuk. "Introducing Enzo, an AMR Cosmology Application". In: *Adaptive Mesh Refinement - Theory and Applications*. Vol. 41. Lecture Notes in Computational Science and Engineering. 2005. ISBN: 978-3-540-27039-3.

[98] J. W. L. P. F. Fischer and S. G. Kerkemeier. *Nek5000 Web page http://nek5000.mcs.anl.gov*. 2008.

[99] S. Pakin and P. McCormick. *Byfl: Compiler-based Application Analysis*. https://github.com/losalamos/Byfl. 2013.

[100] *Parallel Workload Archive*. http://www.cs.huji.ac.il/labs/parallel/workload/.

[101] *ParaView, KitWare*. http://www.paraview.org/.

[102] C. Patel, R. Sharma, C. Bash, and S. Graupner. "Energy Aware Grid: Global Workload Placement Based on Energy Efficiency". In: *Proceedings of the International Mechanical Engineering Congress and Exposition (ASME '03)*. American Society of Mechanical Engineers. 2003, pp. 267–275.

[103] C. M. Patrick, S. W. Son, and M. Kandemir. "Comparative Evaluation of Overlap Strategies with Study of I/O Overlap in MPI-IO". In: *Operating Systems Review (SIGOPS)* 42 (6 Oct. 2008), pp. 43–49. ISSN: 0163-5980. DOI: 10.1145/1453775.1453784.

[104] P. Pepeljugoski, J. Kash, F. Doany, D. Kuchta, L. Schares, C. Schow, M. Taubenblatt, B. Offrein, and A. Benner. "Low Power and High Density Optical Interconnects for Future Supercomputers". In: *Proceedings of the Optical Fiber Communication (OFC), collocated with the Fiber Optic Engineers Conference, (OFC/NFOEC '10)*. Mar. 2010.

[105] S. Plimpton. "Fast Parallel Algorithms for Short-Range Molecular Dynamics". In: *Journal of Computational Physics* 117.1 (1995). ISSN: 0021-9991. DOI: 10.1006/jcph.1995.1039.

[106] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. "MPI-IO/GPFS an Optimized Implementation of MPI-IO on Top of GPFS". In: *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '01)*. Los Alamitos, CA, USA: IEEE Computer Society, 2001. ISBN: 1-58113-293-X. DOI: 10.1145/582034.582051.

[107] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu. "Understanding Performance Interference of I/O Workload in Virtualized Cloud Environments". In: *Proceedings of the IEEE International Conference on Cloud Computing (Cloud '10)*. July 2010, pp. 51–58. DOI: 10.1109/CLOUD.2010.65.

[108] *Python/C API*. http://docs.python.org/2/c-api/.

[109] Y. Qian, E. Barton, T. Wang, N. Puntambekar, and A. Dilger. "A Novel Network Request Scheduler for a Large Scale Storage System". English. In: *Computer Science - Research and Development* 23 (3-4 2009), pp. 143–148. ISSN: 1865-2034. DOI: 10.1007/s00450-009-0073-9.

[110] M. Rasquin, P. Marion, V. Vishwanath, B. Matthews, M. Hereld, K. Jansen, R. Loy, A. Bauer, M. Zhou, O. Sahni, et al. "Electronic Poster: Co-Visualization of Full Data and In Situ Data Extracts from Unstructured Grid CFD at 160k Cores". In: *ACM/IEEE SC Companion*. ACM. 2011, pp. 103–104.

[111] M. Rivi, L. Calori, G. Muscianisi, and V. Slavnic. "In-Situ Visualization: State-of-the-Art and Some Use Cases". In: *PRACE White Paper (2012), http://www.prace-ri.eu/Visualisation* (2011).

[112] I. Sample. *Universe Recreated in Massive Computer Simulation*. http://www.theguardian.com/science/2014/may/07/universe-recreated-computer-simulation-model-big-bang, viewed on May 2014. News Article. 2014.

[113] D. C. Schmidt. *Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events*. 1995.

[114] F. Schmuck and R. Haskin. "GPFS A Shared-Disk File System for Large Computing Clusters". In: *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*. 2002.

[115] W. Schroeder, L. Avila, and W. Hoffman. "Visualizing with VTK: a Tutorial". In: *IEEE Computer Graphics and Applications* 20.5 (2000), pp. 20–27. ISSN: 0272-1716. DOI: 10.1109/38.865875.

[116] H. Shan and J. Shalf. "Using IOR to Analyze the I/O Performance for HPC Platforms". In: *Proceedings of the Cray User Group Conference (CUG '07)*. Seattle, Washington, USA, 2007.

[117] D. Skinner and W. Kramer. "Understanding the Causes of Performance Variability in HPC Workloads". In: *Proceedings of the IEEE Workload Characterization Symposium (IISWC '05)*. IEEE Computer Society, 2005, pp. 137–149. ISBN: 0-7803-9461-5. DOI: 10.1109/IISWC.2005.1526010.

[118] D. C. Snowdon, S. Ruocco, and G. Heiser. "Power Management and Dynamic Voltage Scaling: Myths and Facts". In: (2005).

[119] S. W. Son, G. Chen, M Kandemir, and A Choudhary. "Exposing Disk Layout to Compiler for Reducing Energy Consumption of Parallel Disk Based Systems". In: *Proceedings of the tenth ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. ACM. 2005, pp. 174–185.

[120] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang. "Server-Side I/O Coordination for Parallel File Systems". In: *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '11)*. Nov. 2011.

[121] S. Song, R. Ge, X. Feng, and K. W. Cameron. "Energy Profiling and Analysis of the HPC Challenge Benchmarks". In: *International Journal of High Performance Computing Applications (IJHPCA)* (2009).

[122] R. Springer, D. K. Lowenthal, B. Rountree, and V. W. Freeh. "Minimizing Execution Time in MPI Programs on an Energy-Constrained, Power-Scalable Cluster". In: *Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*. ACM. 2006, pp. 230–238.

[123] J. Steele. "ACPI Thermal Sensing and Control in the PC". In: *Proceeding of Wescon/98*. Sept. 1998, pp. 169–182. DOI: 10.1109/WESCON.1998.716441.

[124] Y. Tanimura, R. Filgueira, I. Kojima, and M. Atkinson. "Poster: Reservation-Based I/O Performance Guarantee for MPI-IO Applications Using Shared Storage Systems". In: *ACM/IEEE SC Companion*. Nov. 2012, pp. 1384–1384.

[125] R. Thakur, W. Gropp, and E. Lusk. "Data Sieving and Collective I/O in ROMIO". In: *Symposium on the Frontiers of Massively Parallel Processing* (1999), p. 182. DOI: 10.1109/FMPC.1999.750599.

[126] R. Thakur, W. Gropp, and E. Lusk. "On Implementing MPI-IO Portably and with High Performance". In: *Proceedings of the sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*. ACM. 1999, pp. 23–32.

[127] P. Thibodeau. *Scientists, IT Community Await Exascale Computers*. http://www.computerworld.com/s/article/345800/Scientists_IT_Community_Await_Exascale_Computers, viewed on May 2014. Dec. 2009.

[128] D. Thompson, N. Fabian, K. Moreland, and L. Ice. *Design Issues for Performing In Situ Analysis of Simulation Data*. Tech. rep. Technical Report SAND2009-2014, Sandia National Laboratories, 2009.

[129] *Titan supercomputer, Oak Ridge National Laboratory (ORNL)*. https://www.olcf.ornl.gov/titan/.

[130] *Top500 List of Supercomputers*. http://www.top500.org/.

[131] N. Tran and D. A. Reed. "Automatic ARIMA Time Series Modeling for Adaptive I/O Prefetching". In: *IEEE Transaction on Parallel and Distributed Systems (TPDS)* 15.4 (Apr. 2004), pp. 362–377. ISSN: 1045-9219. DOI: 10.1109/TPDS.2004.1271185.

[132] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron. "From Mesh Generation to Scientific Visualization: an End-to-End Approach to Parallel Supercomputing". In: *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '06)*. Tampa, Florida, USA: ACM, 2006. ISBN: 0-7695-2700-0. DOI: 10.1145/1188455.1188551.

[133] M. J. Turk, B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman. "yt: A Multi-Code Analysis Toolkit for Astrophysical Simulation Data". In: *The Astrophysical Journal Supplement Series* 192.1 (2011).

[134] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen, J. Shalf, K. Karavanic, and L. Oliker. "Parallel I/O Performance: From Events to Ensembles". In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '10)*. Apr. 2010. DOI: 10.1109/IPDPS.2010.5470424.

[135] *VisIt, Lawrence Livermore National Laboratory (LLNL)*. https://wci.llnl.gov/simulation/computer-codes/visit.

[136] M. Vogelsberger, S. Genel, V. Springel, P. Torrey, D. Sijacki, D. Xu, G. Snyder, S. Bird, D. Nelson, and L. Hernquist. "Properties of galaxies reproduced by a hydrodynamic simulation". In: *Nature* 7499 (2014), 177—182. DOI: 10.1038/nature13316.

[137] L. Wang, G. Von Laszewski, J. Dayal, and F. Wang. "Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS". In: *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid '10)*. IEEE. 2010, pp. 368–377.

[138] B. Whitlock, J. M. Favre, and J. S. Meredith. "Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System". In: *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization (EGPGV '10)*. Eurographics Association, 2011.

[139] F. Wu. "Sequential File Prefetching in Linux". In: *Advanced Operating Systems and Kernel Applications: Techniques and Technologies*. IGI Global, 2010, pp. 217–236. DOI: 10.4018/978-1-60566-850-5.ch011.

[140] O. Yildiz, M. Dorier, S. Ibrahim, and G. Antoniu. "A Performance and Energy Analysis of I/O Management Approaches for Exascale Systems". In: *Proceedings of the Sixth International Workshop on Data Intensive Distributed Computing (DIDC '14)*. Vancouver, Canada: ACM, 2014, pp. 35–40. ISBN: 978-1-4503-2913-2. DOI: 10.1145/2608020.2608026.

[141] H. Yu and K. Ma. "A Study of I/O Techniques for Parallel Visualization". In: *Journal of Parallel Computing* 31.2 (2005).

[142] H. Yu, C. Wang, R. Grout, J. Chen, and K.-L. Ma. "In Situ Visualization for Large-Scale Combustion Simulations". In: *IEEE Computer Graphics and Applications* 30.3 (2010), pp. 45–57. ISSN: 0272-1716. DOI: 10.1109/MCG.2010.55.

[143] F. Zanon Boito, R. Kassick, P. Navaux, and Y. Denneulin. "AGIOS: Application-Guided I/O Scheduling for Parallel File Systems". In: *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS '13)*. 2013, pp. 43–50. DOI: 10.1109/ICPADS.2013.19.

[144] F. Zhang, M. Parashar, C. Docan, S. Klasky, N. Podhorszki, and H. Abbasi. "Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform". In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '12)*. IEEE. 2012.

[145] F. Zhang, S. Lasluisa, T. Jin, I. Rodero, H. Bui, and M. Parashar. "In-situ Feature-Based Objects Tracking for Large-Scale Scientific Simulations". In: *ACM/IEEE SC Companion*. IEEE. 2012.

[146] X. Zhang, K. Davis, and S. Jiang. "Opportunistic Data-driven Execution of Parallel Programs for Efficient I/O Services". In: *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS '12)*. IEEE. 2012, pp. 330–341.

[147] X. Zhang, K. Davis, and S. Jiang. "IOrchestrator: Improving the Performance of Multi-Node I/O Systems via Inter-Server Coordination". In: *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*. IEEE Computer Society. 2010.

[148] X. Zhang, K. Davis, and S. Jiang. "QoS Support for End Users of I/O-Intensive Applications using Shared Storage Systems". In: *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC '11)*. Seattle, Washington, USA, Nov. 2011.

[149] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. "PreDatA – Preparatory Data Analytics on Peta-Scale Machines". In: *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS '10)*. 2010. DOI: 10.1109/IPDPS.2010.5470454.

[150] F. Zheng, J. Cao, J. Dayal, G. Eisenhauer, K. Schwan, M. Wolf, H. Abbasi, S. Klasky, and N. Podhorszki. "High End Scientific Codes with Computational I/O Pipelines: Improving their End-to-End Performance". In: *Proceedings of the 2nd International Workshop on Petascal Data Analytics: Challenges and Opportunities (PDAC '11)*. Seattle, Washington, USA: ACM, 2011, pp. 23–28. ISBN: 978-1-4503-1130-4. DOI: 10.1145/2110205.2110210.

[151] F. Zheng, H. Abbasi, J. Cao, J. Dayal, K. Schwan, M. Wolf, S. Klasky, and N. Podhorszki. "In-situ I/O Processing: A Case for Location Flexibility". In: *Proceedings of the Sixth Workshop on Parallel Data Storage (PDSW '11)*. Seattle, Washington, USA: ACM, 2011, pp. 37–42. ISBN: 978-1-4503-1103-8. DOI: 10.1145/2159352.2159362.

[152] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi. "Using MPI in High-Performance Computing Services". In: *Proceedings of the European MPI Users' Group Meeting (EuroMPI '13)*. Sept. 2013, pp. 43–48.