Programmation des architectures hiérarchiques et hétérogènes

Khaled Hamidouche

UNIVERSITÉ PARIS-SUD XI

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PARIS-SUD XI

PRÉPARÉE AU LABORATOIRE DE RECHERCHE EN INFORMATIQUE
DANS LE CADRE DE *l'Ecole Doctorale d'Informatique de Paris-Sud"*

PRÉSENTÉE ET SOUTENUE PUBLIQUEMENT

PAR

KHALED HAMIDOUCHE

LE 10 NOVEMBRE 2011

PROGRAMMATION DES MACHINES HIÉRARCHIQUES ET HÉTÉROGÈNES

Directeur de thèse :
Pr. Daniel Etiemble

Dr. Joel Falcou

JURY

Pr. Philippe CLAUSS	Rapporteur
Pr. Frédéric LOULERGUE	Rapporteur
Pr. Gaétan HAINS	Examinateur (Président)
Pr. Alain MÉRIGOT	Examinateur
Pr. Daniel ETIEMBLE	Directeur de thèse
Dr. Joel FALCOU	Co-directeur de thèse

Remerciements

Au terme de ce doctorat, je tiens tout d'abord à remercier mon directeur de thèse le Professeur Daniel Etiemble pour la confiance qu'il m'a accordée pendant le déroulement de ces travaux. Sa disponibilité et sa gentillesse m'ont permis de mener de manière relativement libre cette thèse.

Mes plus vifs remerciements s'adressent à mon Co-directeur de thèse le Docteur Joel Falcou sans qui cette thèse ne serait pas ce qu'elle est. Je le remercie pour avoir encouragé les échanges avec différents chercheurs et laboratoires, pour les nombreuses collaborations qu'il a apportées, pour son dynamisme et son infatigable énergie, pour la compagnie au cours des longues journées de synchrotron.

Je remercie aussi Messieurs les Professeurs Philippe Clauss et Frédéric Loulergue pour m'avoir fait l'honneur de relire ce manuscrit et d'avoir participer au Jury. Ils ont également contribué par leurs remarques et suggestions à améliorer la qualité de ce mémoire, et je leur en suis très reconnaissant.

Mes remerciements vont également aux Professeurs Gaétan Hains et Alain Mérigot qui m'ont fait l'honneur de participer à mon Jury de thèse.

Je tiens à remercier tous les chercheurs et amis avec qui j'ai collaboré en particulier : Claude, Alba, Sylvain et Lionel. Je remercie aussi Sébastien pour son aide au niveau de la programmation et de l'utilisation de Cell BE.

Je remercie particulièrement Fatiha pour sa présence, son aide et sa joie de vivre. Je remercie également Yasmina, Amal, Amina et Ala.

Je tiens à remercier mes amis et collègues de travail qui m'ont supporté durant toute cette période en particulier : Pierre, Sébastien, Roumain, Mathias, Mikolj, Riad et Florence.

Mes remerciements s'adressent à tous mes amis : Amir, Aziz, Faycel, Lina, Hana, Raouf... Plus généralement, il me faut remercier toutes les personnes que j'ai eu l'occasion de côtoyer au cours de ces années, que ce soit au département Informatique pendant mes enseignements ou au laboratoire.

Enfin, merci à ma famille, de m'avoir donné la chance de faire de si longues études.

TABLE DES MATIÈRES

INTRODUCTION 1

1.1 Une fuite en avant ?

Dans le numéro de Juillet 2010 du magazine Spectrum de l'IEEE, David Patterson, célèbre pour avoir avec John Hennessy popularisé les processeurs RISC au milieu des années 80, publiait un article au titre spectaculaire et quasi-provocateur : « Le problème avec les microprocesseurs multi-cœurs - Les fabricants de circuit intégrés s'activent à concevoir des microprocesseurs que la plupart des programmeurs seront incapables d'utiliser. » Dans cet article, il rappelle tous les problèmes de la programmation parallèle qui existent depuis près de 50 ans maintenant, puisque les premières machines parallèles datent du début des années 60. Si ces dernières, notamment depuis la période où elles ont supplanté les machines vectorielles, ont été la solution de choix pour le calcul haute performance, elles sont restées confinées à ce domaine stratégiquement très important, mais qui restait un marché réduit par rapport à l'ensemble du marché du matériel informatique. La technologie CMOS [1], avec une progression exponentielle des fréquences d'horloge et les progrès architecturaux (superscalaires, caches, prédicteurs de branchement, exécution non ordonnée, etc.) ont permis pendant de nombreuses années un gain de performance des microprocesseurs de l'ordre de 50% par an. Cette progression spectaculaire associée à la simplicité de la programmation séquentielle et aux performances des compilateurs ont réservé l'approche parallèle aux serveurs et super-ordinateurs haut de gamme. La barrière de la chaleur est devenue au milieu des années 2000 la question clé qui interdit de continuer d'augmenter la fréquence d'horloge des processeurs. C'est cet obstacle qui a conduit au virage vers les processeurs multi-cœurs, l'augmentation du nombre de cœurs (processeurs élémentaires) devenant la seule solution permettant de continuer d'augmenter les performances. Mais ce qui était le problème des machines parallèles devient de fait le problème de tous les ordinateurs puisque tout ordinateur, du PC portable ou de bureau aux super-ordinateurs pour le calcul haut de gamme, devient une machine parallèle. En même temps, ces machines parallèles peuvent être très diverses, comme nous le préciserons dans le chapitre

1. Complementary metal oxide semi-conductor

2 : elles peuvent être hiérarchisées (clusters de multiprocesseurs), elles peuvent être hétérogènes (accélérateurs matériels associés aux cœurs), etc.

1.2 Un problème fondamental

Comme le précise David Patterson, si les fabricants de circuits intégrés ont fait un tel choix (construire des processeurs à grand nombre de cœurs sans être sûr qu'ils soient programmables par le plus grand nombre), c'est fondamentalement parce qu'ils n'avaient aucun autre choix possible. Le problème auquel sont confrontés les informaticiens du parallélisme est donc de fournir des outils permettant à la plupart des programmeurs d'être capable d'utiliser ces processeurs multi-cœurs. Un certain nombre de modèles de programmation parallèle et de bibliothèques les implémentant existent, comme nous le verrons au chapitre 3. Le problème est double. D'une part, il faut être capable d'utiliser les ressources de l'architecture parallèle, c'est-à-dire obtenir des performances les plus proches possibles des performances crête de la machine compte tenu des caractéristiques de l'application exécutée. Dit d'une autre manière, il faut obtenir les performances proches de celles qu'obtiendrait un programmeur expert spécialiste de l'architecture parallèle utilisée. D'autre part, la programmation doit rester simple, la plus proche possible de la programmation séquentielle classique. Dans cette thèse, nous proposons une plate-forme de développement et de programmation des machines parallèles hiérarchiques visant à répondre simultanément aux problèmes de performance et de programmabilité. Les outils proposés visent une haute abstraction et une facilité de programmation tout en assurant une grande efficacité.

1.3 Organisation du manuscrit

La présentation des travaux s'organise de la manière suivante :

- **Chapitre 2 : Architectures pour le calcul haute performance.** Nous commençons dans ce chapitre par présenter un état de l'art des architectures hybrides pour le calcul haute performance suivi par les outils, langages et bibliothèques de bas niveaux permettant leur programmation.

- **Chapitre 3 : Modèles et outils de calcul haut niveau.** Ce chapitre présente les classifications des modèles de programmation parallèle de haut niveau ainsi que quelques bibliothèques les implémentant.

- **Chapitre 4 : La bibliothèque BSP++.** Ce chapitre présente une description de notre proposition : le modèle BSP++ et la bibliothèque associée.

Et nous détaillerons son implémentation pour les différentes architectures cibles ainsi que son support pour les architectures hybrides.

- **Chapitre 5 : L'outil BSPGen.** Ce chapitre présente le framework BSPGen qui est un outil pour la génération de code parallèle automatiquement à partir d'une description de code parallèle BSP et une liste de fonctions séquentielles. Dans ce chapitre, nous exposons l'architecture globale du l'outil ainsi que la fonctionnalité des différents modules le constituant.

- **Chapitre 6 : Évaluation de performances sur applications.** Ce chapitre présente les performances obtenues pour différents types de machines parallèles sur des applications correspondant à différents domaines : model checking, bioinformatique (Algorithme de Smith Watermann), traitement d'images bas niveau (algorithme de Harris), etc.

- **Conclusions & perspectives :** Ce chapitre dresse une conclusion sur les résultats obtenus et donne des perspectives de travaux futurs.

ARCHITECTURES POUR LE CALCUL HAUTE PERFORMANCE 2

La réalisation de machines parallèles et massivement parallèles implique une hiérarchie de niveaux qu'illustre bien la structure du super-ordinateur Blue Gene d'IBM présenté en figure 2.1. La machine est constituée de 64 armoires. Chaque armoire comprend 32 cartes de calcul. Chaque carte comprend 2 nœuds et chaque nœud comprend 2 processeurs avec 4 Mo de mémoire DRAM. Au niveau de la machine globale, les armoires sont interconnectées via une topologie torique à 3 dimensions [1]. A partir de chaque armoire, on a un graphe de type arbre, dont le sommet est l'armoire et les feuilles sont les processeurs de calcul avec leurs caches de premier niveau. Il est évident que les communications entre les processeurs et les données en mémoire vont dépendre des chemins à parcourir dans l'ensemble "tore + arbres" pour établir la liaison entre le processeur et la donnée à laquelle il accède. Plus précisément, l'accès aux données va se décomposer en deux types.

– Les données auxquelles le processeur demandeur peut accéder via la hié-

rarchie mémoire proche (différents niveaux de cache) avec gestion matérielle simple de la cohérence des caches : c'est le cas typique des multiprocesseurs SMP.

– Les autres données. Il existe alors deux types de solutions. La première solution consiste à assurer la cohérence des caches via des répertoires avec des mémoires locales et des mémoires lointaines. C'est le cas des machines cc-NUMA [1], c'est-à-dire à accès mémoire non uniformes, mais avec cohérence des caches. Les machines SGI Origin 2000 (1995) et Origin 3000 (2000) sont des exemples de ce type. L'autre solution consiste à interconnecter via un réseau des multiprocesseurs de type SMP. Les transferts entre mémoires des SMP se font alors par passage de messages.

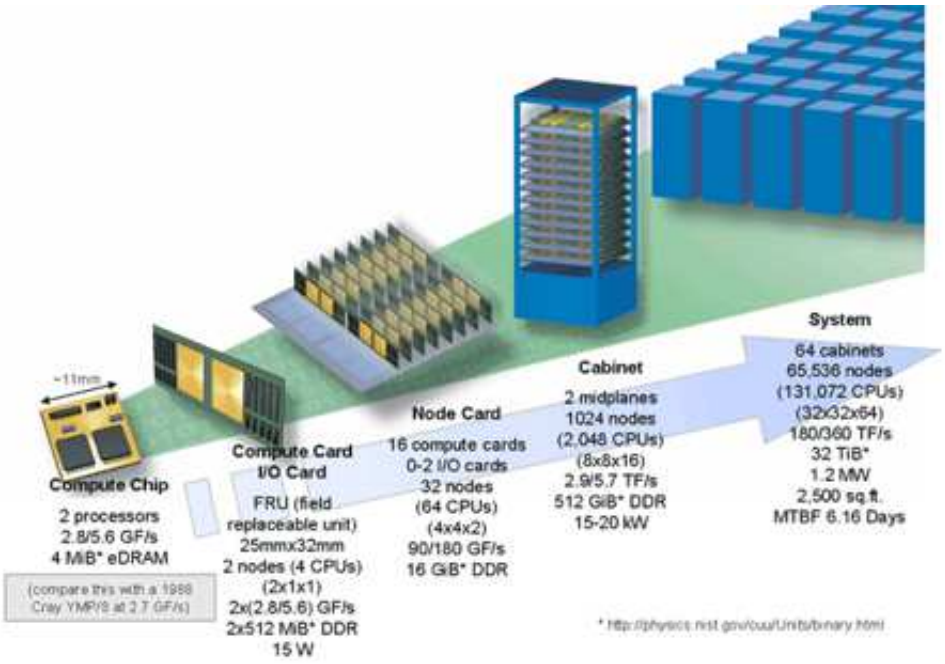


FIGURE 2.1 – L'organisation hiérarchique de l'IBM Blue Gene

Depuis de nombreuses années les grappes de multiprocesseurs se sont imposés comme une solution permettant de concilier performance et utilisation de composants standards. Le super-ordinateur apparaît comme un ensemble de stations de travail haut de gamme (multiprocesseurs) avec un réseau d'interconnexion performant. Le nœud de calcul (la station de travail) n'est pas fondamentalement différent des stations de travail commercialisées par le constructeur. Nous présentons maintenant l'évolution de ces grappes, avec l'impact sur les modèles de programmation utilisables.

1. Cache Coherent Non Uniform Memory Access

2.1 Les grappes de processeurs

Nous nous contenterons ici de présenter un seul exemple, représentatif de cette approche : le projet NOW [2] de Berkeley. Celui-ci cherche à démontrer qu'il est viable de construire de grands systèmes de calcul parallèle qui sont rapides, peu coûteux et hautement disponible, en raccordant un nombre de stations de travail par un réseau de communication.

La configuration matérielle du système NOW de Berkeley se compose de cent cinq postes de travail Sun Ultra 170, connectés par un réseau Myricom [3] à grand débit. Chaque poste de travail comporte un microprocesseur 167 MHz Ultra1 avec 512 Ko de cache de niveau-2, 128 Mo de mémoire. La figure 2.2 illustre l'architecture globale du système NOW.

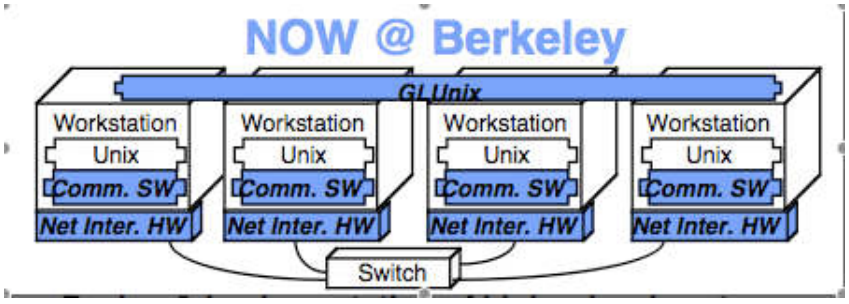


FIGURE 2.2 – Architecture globale du système NOW.

NOW étant constitué de stations de travail indépendantes reliées par un réseau d'interconnexion, la mémoire est distribuée et le modèle de programmation associé est le « passage de messages ».

2.2 Les grappes de multiprocesseurs

L'IBM SP/2, disponible au début des années 2000, est un exemple typique de cluster de multiprocesseurs. La version avec nœuds WinterHawk II avait des nœuds constitués de 4 processeurs Power 3 interconnectés par le réseau IBM SHSwitch. Un cluster de multiprocesseurs combine deux modèles de mémoire. Chaque multiprocesseur a une mémoire commune, avec le modèle de programmation associé. Le réseau d'interconnexion implique la mémoire distribuée au niveau de la machine globale, avec le modèle de programmation associé "passage de messages". La programmation des grappes de multiprocesseurs pose le problème du choix du modèle de programmation : soit utilisation d'un seul modèle, qui dans ce cas ne peut être que le modèle "passage de messages", soit utilisation de deux modèles, c'est-à-dire "mémoire partagée" à l'intérieur d'un

nœud et "passage de messages" entre les nœuds. C'est cette alternative qui est étudiée par F. Cappello, D. Etiemble *et al* dans plusieurs articles concernant les grappes de PC et les grappes SP-2 d'IBM [4] et que j'ai reprise lors de mon stage de DEA pour des grappes de multiprocesseurs multicœurs [5].

2.3 Les grappes de multiprocesseurs multicœurs

Les processeurs multicœurs sont relativement anciens. Les processeurs d'IBM Power 5 (2004) et Power 6 (2007) comprenaient deux cœurs sur la même puce. Les multicœurs introduits par Intel (Xeon) et AMD (Opteron) avec le jeu d'instructions x86-64 Intel64 et AMD64) ont conduit à la généralisation de l'utilisation des processeurs multi-cœurs, du PC aux machines parallèles et massivement parallèles. Le cluster de multiprocesseurs multicœurs interconnecte plusieurs nœuds via un réseau d'interconnexion. Chaque nœud comprend plusieurs puces «processeur », chaque puce comprenant plusieurs cœurs. Chaque nœud a une mémoire partagée avec le modèle de programmation associé. Mais il y a maintenant un niveau supplémentaire de hiérarchie, puisque les accès mémoire dans la hiérarchie mémoire sont différents selon qu'ils se font dans la même puce ou dans des puces différentes d'un même nœud (voir figure 2.3). Les nœuds sont interconnectés par un réseau, ce qui conduit à une mémoire distribuée pour l'ensemble du cluster. La problématique du modèle de programmation à utiliser, unique ou mixte, est la même que pour les grappes de multiprocesseurs.

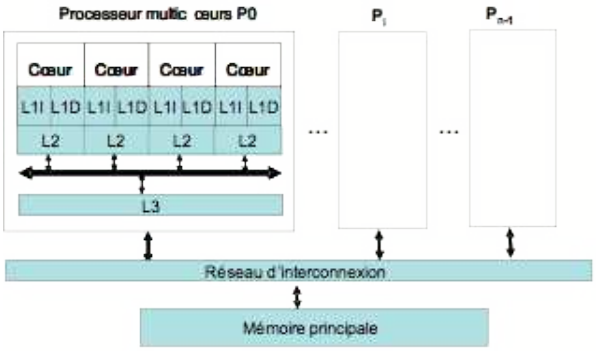


FIGURE 2.3 – Schéma de principe d'un cluster de multiprocesseurs multicœurs

La plupart des supercalculateurs figurant dans la dernière liste top500 sont des grappes de multicœurs, éventuellement avec des nœuds hybrides que nous allons présenter maintenant.

2.4 grappes de nœuds hybrides

Les grappes de multiprocesseurs multicœurs présentés précédemment ont des nœuds constitués de cœurs identiques. Avec des cœurs homogènes, le modèle de programmation est unique au niveau des nœuds. On peut cependant ajouter aux circuits multicœurs des accélérateurs spécialisés. Certains accélérateurs de calcul spécialisés ont été développés, comme les coprocesseurs ClearSpeed [6].

2.4.1 Nœuds avec GPU

Les processeurs graphiques (GPU) étaient à l'origine des coprocesseurs destinés à l'implémentation du pipeline graphique. Leur évolution spectaculaire a fait qu'un GPU est fondamentalement une puce multicœurs comprenant des centaines de cœurs simples (on parle alors de *manycore*), appelés *Streaming Processors* (SP), avec un contrôle logique pour les différents niveaux d'encapsulation dans le modèle de programmation CUDA (voir section 2.7.2). Chaque SP exécute les instructions séquentiellement, dispose d'un pipeline, 2 unités arithmétique-logique et une unité à virgule flottante. Les nouvelles générations des GPU ont en plus une hiérarchie de cache [7, 8].

Plusieurs SP sont encapsulés dans un multiprocesseur Streaming (SM). Plusieurs SM constituent le Texture Processor Cluster (TPC) (voir figure 2.4). A l'intérieur d'un SM, le SP exécute dans un mode SIMT (Single Instruction Multi Threads : ressemble à SIMD), tandis que différents SM peuvent exécuter différentes parties du flux d'instructions dans un mode SPMD. Chaque SM gère également des centaines de threads actifs dans un pipeline cyclique afin de masquer la latence mémoire. Dans la pratique, 32 threads sont regroupés dans un $Warp$ et exécutent les mêmes instructions.

2.4.1.1 Exemple de cluster de nœuds avec GPU : Le Tianhe-1A

L'utilisation d'une architecture hybride CPU-GPU pour la construction des super calculateur a permis d'atteindre des performances énormes. En effet, en 2010, le cluster le plus rapide de la planète utilise une architecture hybride CPU-GPU.

Le **Tianhe-1A** [8] est une machine installée à Tianjin et conçue par l'Université Nationale des Technologies de Défense en Chine. Tianhe-1A utilise une architecture mixant CPU et GPU, ce qui lui permet d'atteindre une puissance de traitement sans égale, tout en conservant une certaine flexibilité. Les 112 armoires dédiées aux serveurs de calcul comprennent ainsi un total de 14 336 processeurs

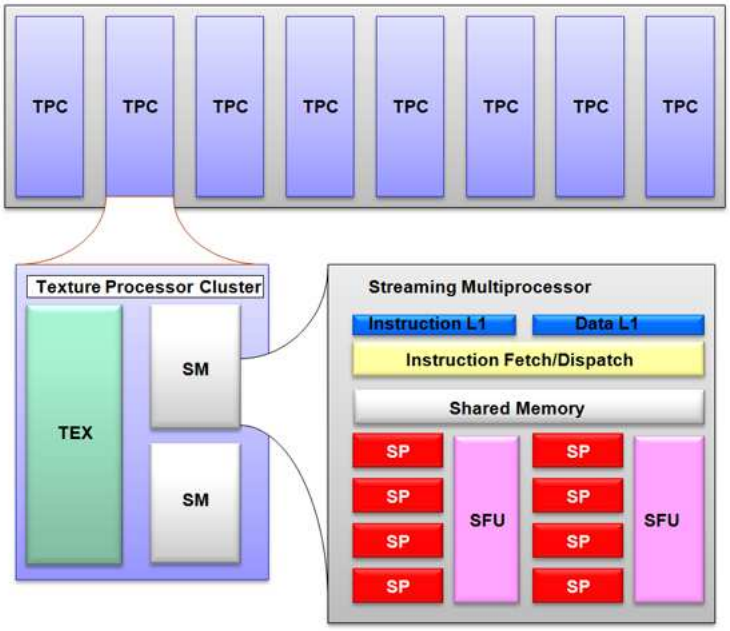


FIGURE 2.4 – Architecture d'un GPU NVIDIA

classiques et de 7168 cartes NVIDIA Tesla M2050 avec un total de 262 To de mémoire vive. La puissance brute atteint les 4,7 pétaflops et la performance est de 2,5 pétaflops pour le test Linpack.

Les GPU sont utilisés comme coprocesseurs accélérateurs de calcul. Le problème est l'existence de deux modèles de programmation différents d'une part, et l'alimentation en données des mémoires des GPU d'autre part.

2.4.2 Nœuds avec processeurs Cell BE

Le processeur Cell BE est utilisé comme accélérateur dans certains supercalculateurs. Il est lui-même un processeur multicœur hétérogène [9, 10, 11]. Composé d'un processeur 64-bit (*Power Processor Element* ou PPE), huit unités spécialisées appelées *Synergitic Processor Element* (SPE) et un bus à haut débit appelé *Element Interconnect Bus* (EIB) qui permet des communications entre ces différentes composantes. Le Cell BE est une puce hétérogène multi-cœurs contenant plusieurs niveaux de parallélisme qui peuvent être exploités pour atteindre des performances théoriques de 204,8 GFlops en simple précision. Le EIB est composé de quatre anneaux de 128 bits, chaque anneau peut gérer jusqu'à trois transferts concurrents. La bande passante maximale théorique du bus est 204,8 Go/s pour les transferts internes.

Le PPE est un processeur PowerPC classique 64 bits avec une extension vectorielle multimédia VMX (Altivec). Le PPE est considéré comme le processeur principal de Cell, il est chargé de la gestion du système d'exploitation et de la coordination des SPEs. Chaque SPE consiste en un SPU Synergitic Processor Unit et un controlleur de mémoire MFC (Memory Flow Controller).

Le SPE a une mémoire locale LS (Local Store) limitée à 256 Ko pour le stockage du code et des données qui sont gérées explicitement par le code utilisateur, et une unité vectorielle SWAR de 128 bits (très proche de Altivec) pour les instructions SIMD dédiées au calcul haute performance intensif. Le MFC contient un contrôleur DMA (Direct Memory Acces), qui est en charge du transfert de données de l'extérieur vers la mémoire locale (LS), ou pour écrire les résultats du calcul de la mémoire locale vers la mémoire principale (RAM).

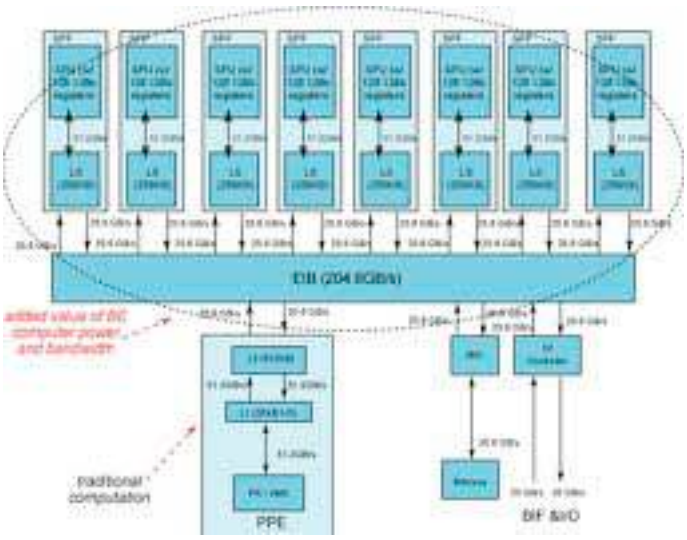


FIGURE 2.5 – Architecture du processeur Cell BE

Le processeur Cell utilise les mémoires locales des SPE comme un cache logiciel, les transferts entre ces caches logiciels et la mémoire principale du Cell étant réalisés par des DMA [10]. Cette spécificité introduit des contraintes de programmation que nous aborderons plus loin dans ce document.

2.4.2.1 Exemple de cluster de nœuds avec processeurs Cell BE : Le Roadrunner

L'un des plus célèbres grappes utilisant des accélérateurs hétérogènes est le Roadrunner [11] (qui a tenu la première place dans le Top 500 en Juin 2009). Roadrunner est un super-ordinateur construit par IBM au Los Alamos National Laboratory au Nouveau Mexique. Il s'agit d'une conception hybride avec 12 960 IBM PowerXCell8i et 6 480 processeurs AMD Opteron double-cœurs dans les serveurs lame spécialement conçus reliés par un réseau Infiniband.

Roadrunner diffère de nombreux super-ordinateurs contemporain en ce qu'il s'agit d'un système hybride, utilisant deux architectures différentes de processeur. La conception hybride comprend des processeurs Opteron bi-cœurs fabriqués par AMD en utilisant l'architecture AMD64 standard. Un processeur Cell est attaché à chaque cœur Opteron. Comme super-ordinateur, le Roadrunner est considéré comme un cluster Opteron avec des accélérateurs de type Cell.

D'un point de vue fonctionnel, les nœuds de gestion et d'optimisation de la distribution des applications (Lames LS22 à double processeur Opteron fig 2.6) sont utilisés uniquement pour la distribution des tâches, sur les nœuds d' accélération (Lame QS22 de Cell) et n'interviennent pas dans le calcul.

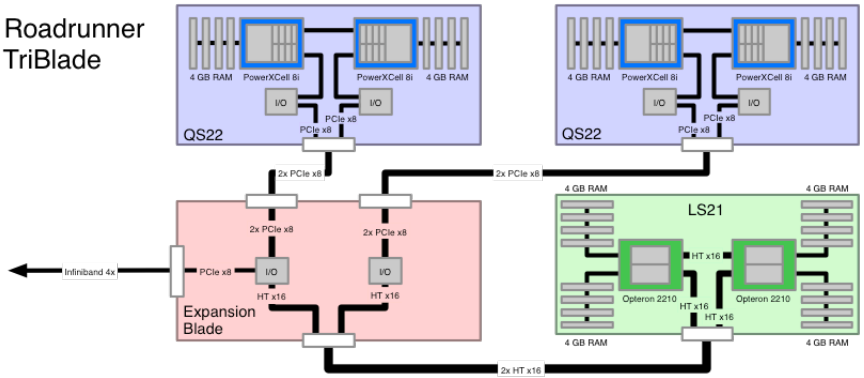


FIGURE 2.6 – Architecture et configuration du supercalculateur *Roadrunner*.

2.4.3 Nœuds avec FPGA

Des circuits programmables FPGA sont utilisés comme accélérateurs dans certains super-ordinateurs. Cependant, compte tenu de la différence de fréquence de fonctionnement entre multicœurs et FPGA, ce n'est que pour des applications particulières qui peuvent tirer parti du parallélisme d'opérations que permet le

FPGA que cette approche est rentable. Nous n'aborderons pas l'accélération via FPGA dans la suite du document.

2.5 Outils de programmation des grappes

Il existe de nombreux environnements de programmation pour les architectures à mémoires distribuées. Les plus connus sont MPI [2] [12] et PVM [3] [13]. Ces deux technologies reposent sur un moyen commun pour la communication : le passage de messages entre les nœuds. Néanmoins MPI est considéré comme le standard pour la programmation à passage de message grâce à ses meilleures performances.

2.5.1 MPI : Message Passing Interface

MPI offre plusieurs fonctionnalités et mécanismes nécessaires au développement d'applications parallèles :

- Mécanismes de communications point à point et collectives.
- Portabilité et support de plusieurs langage de programmation (C, C++, FORTRAN, ...).
- Support de l'hétérogénéité des systèmes et des topologies des réseaux de communication.

2.5.1.1 Principes

Un programme MPI est constitué d'un ensemble de tâches autonomes appelé "processus" exécutant leur propre code. MPI permet la programmation avec les deux modèles MIMD [14] et SPMD. Pour le modèle MIMD, chaque processus exécute un code différent alors que pour le modèle SPMD, tous les processus exécutent le même code. Chaque processus possède un identifiant unique et peut communiquer avec les autres processus en utilisant des primitives de passage de messages. Ces primitives constituent les opérations de bases pour MPI et peuvent être répertoriées en trois classes :

- **Communications point à point :**
  ce sont des primitives d'émission et réception de messages d'un processus à un autre. L'émission et la réception sont étiquetées par l'identifiant des processus communicants (identifiant de l'émetteur pour la réception et de récepteur pour l'émission). MPI fournit un large panel de primitives

2. Message Passing Interface
3. Parallel Virtual Machine

de communication point à point synchrone, asynchrone, bloquante et non-bloquante.

- **Communications collectives :**
  Ces primitives sont utilisées pour les communications entre tous les processus au sein d'un même groupe. Ces primitives sont :

  - `MPI_Barrier` : pour la synchronisation de tous les processus d'un même groupe.
  - `MPI_Broadcast` : diffusion des données vers tous les processus.
  - `MPI_Scatter` : dispersion des données sur un groupe de processus.
  - `MPI_Gather / MPI_AllGather` : concentration des données vers un processus unique (respectivement tous les processus).
  - `MPI_Reduce / MPI_AllReduce` : réduction sur un processus (respectivement tous les processus) des éléments provenant de tous les processus de même groupe.
  - `MPI_AllToAll` : c'est le schéma de communication le plus global : il transfère des données de tous les processus vers tous les processus dans le même groupe.

- **Gestion des groupes :**
  Comme chaque processus a un identifiant unique au sein d'un groupe et que les communications sont liées à un groupe, MPI permet la création de sous-groupes au sein du groupe mère appelé `MPI_COMM_WORLD`. Les primitives de partitionnement sont basées sur un objet nommé Communicateur pour la gestion des groupes.

2.5.1.2 Exemple

Listing 2.1 – Exemple de code MPI : Ping-pong

```
1  # include <mpi.h>
2  int main (int argc , char** argv)
3  { int rank, size, value;
4    MPI_Status st;
5    MPI_Init (& argc, &argv);
6    MPI_comm_rank (MPI_COMM_WORLD, & rank);
7    MPI_comm_size (MPI_COMM_WORLD, & size);
8    if (rank==0)
9    {
10    MPI_Send (& rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
11    MPI_Recv (&value, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, & st);
12   }
13   else if (rank==1)
14   {
15    MPI_Recv (&value,1, MPI_INT, 0, 0, MPI_COMM_WORLD, & st);
16    MPI_Send (&value,1, MPI_INT, 0, 1, MPI_COMM_WORLD);
17   }
18   MPI_Finalize();
19   return 0;
20 }
```

Le listing 2.1 présente l'exemple de base de communications avec MPI dans lequel les deux processus communiquent en se transmettant leur identifiant.

Le code parallèle débute par l'activation de l'environnement MPI via la primitive `MPI_Init` à la ligne 3. Puis chaque processus récupère son identifiant via la fonction `MPI_Comm_rank`. Vient ensuite une portion de code activée en fonction du l'identifiant de processus. Le processus d'identifiant 0 utilise la primitive d'émission `MPI_Send` pour envoyer la valeur de son identifiant et attend la réception d'une donnée du processus 1 avec la primitive de réception `MPI_Recv`. Parallèlement, le processus 1 reçoit la donnée du processus 0 et lui la retransmet. Enfin le programme MPI se termine par l'appel à la procédure `MPI_Finalize`.

2.6 Outils de programmation des nœuds SMP

De nombreuse bibliothèques existent pour la programmation des architectures à mémoire partagée. Parmi celles-ci, on trouve principalement : MPI, OpenMP et pThread.

2.6.1 MPI

MPI a été créé initialement pour la programmation des architectures à mémoire distribuée. Et comme MPI est basé essentiellement sur les notions de Processus et de passage de messages, plusieurs implémentations de MPI ont été optimisées pour la programmation des deux architectures à mémoire distribuée et partagée comme OpenMPI, MPICH, LAM-MPI et IBM-MPI.

2.6.2 OpenMP

OpenMP est un outil de programmation à base de directives pour l'implémentation des programmes parallèles sur des systèmes à mémoire partagée [15] en utilisant le concept de multithreading en FORTRAN, C ou C++ [16]. Il existe principalement deux styles de programmation avec OpenMP : le style OpenMP grain fin et le style OpenMP SPMD.

OpenMP grain fin est considéré comme l'approche la plus simple pour paralléliser un code séquentiel ou MPI. Cette approche consiste à paralléliser les boucles de la partie calcul du programme sur les différents *threads* en partitionnant le nombre d'itérations. Ce partitionnement est effectué en ajoutant simplement une directive OpenMP avant la boucle dans le code séquentiel.

La programmation OpenMP SPMD peut améliorer les performances comparées à OpenMP grain fin en réduisant la synchronisation, le partage des données entres les threads et la surcharge introduite par les directives OpenMP. Cette approche consiste à encapsuler le code des applications dans une seule section parallèle. Ainsi les threads agissent comme des processus MPI [17]. Comme avec MPI, le programmeur doit prendre en charge la distribution de données et des calculs entre les *threads* [18].

Le listing 2.2 présente un exemple de code OpenMP dans le style grain fin dans lequel la somme de deux vecteurs s'effectue en parallèle.

Listing 2.2 – Exemple de code OpenMP grain fin

```
1 # include <omp.h>
2 int main ()
3 {
4  int i;
5  float a[1000], b[1000], c[1000];
6  #pragma omp parallel for private(i)
7  for(i=0; i<1000; i++) c[i]=a[i]+ b[i];
8 }
```

Dans ce code, la directive *pragma omp parallel for private (i)* a deux effets :

- démarre l'environnement OpenMP et lance les threads via le *pragma omp parallel*.

- chaque thread exécute la boucle avec un nombre d'itérations en fonction du nombre de threads lancés. La clause $private$ indique que chaque thread utilise une copie locale de la variable i pour éviter l'accès partagé à la variable, ce qui réduit les performances en séquentialisant l'accès.

Listing 2.3 – Exemple de code OpenMP SPMD

```
1  # include <omp.h>
2  int main ()
3  {
4   #pragma omp parallel
5   {
6     int i, rank, size, taille;
7     rank= omp_get_thread_num();
8     size= omp_get_num_threads();
9     taille= 1000/size;
10    float a[taille], b[taille], c[taille];
11
12    for(i=0; i<taille; i++) c[i]=a[i]+ b[i];
13   }
14 }
```

Le code dans le listing 2.3 présente la version OpenMP SPMD de l'exemple précédant. Le code montre la distribution explicite des données et de calcul ef-

fectuée par l'utilisateur en déterminant les bornes des boucles en fonction de nombre des threads . Les fonctions `omp_get_num_threads()` et `omp_get_thread_num()` donnent respectivement le nombre des threads et le rank du thread dans l'environnement.

L'avantage d'OpenMP est que la majorité des compilateurs l'implémentent comme une cible native. Même si le compilateur n'implémente pas OpenMP, comme celui-ci est basé sur des directives $pragma$, le compilateur les considère comme des commentaires.

2.6.3 pThread

Listing 2.4 – Exemple de code pThreads

```
1  # include <pthread.h>
2  struct th_arg {float *pa, *pb, *pc};
3  void * func (void * in)
4   {
5    th_arg * p= (th_arg*) (in);
6    for( int i=0; i<250; i++)
7      p->pc[i]=p->pa[i]+p->pb[i];
8    return NULL;
9   }
10 int main ()
11 {
12    float a[1000], b[1000], c[1000];
13    pthread_t     t[4];
14    th_arg      arg[4];
15
16    for(i=0; i< 4; i++)
17     {  arg[i].pa = &a[i*250];
18        arg[i].pb = &b[i*250];
19        arg[i].pc = &c[i*250];
20     }
21    for(int i= 0; i< 4; i++)
22     pthread_create (&t[i], NULL, NULL, func, & arg[i]) ;
23
24   for(int i= 0; i< 4; i++)
25    pthread_join (t[i], NULL);
26 }
```

pThread est la bibliothèque qui implémente le standard IEEE POSIX 1003.1c pour le système UNIX. pThread est définie comme un ensemble de primitives et de procédures pour la programmation des architectures à mémoire partagée de type SMP en utilisant les threads. La bibliothèque fournit des fonctionnalités bas niveau pour la création, destruction, synchronisation et la manipulation des verrous. L'utilisation de pThread est implémentée par l'inclusion de l'entête $pthread.h$. Le code du listing 2.4 présente l'exemple précédent en utilisant pThread avec quatre threads. La fonction `pthread_create` crée un thread $t[i]$

et lui assigne la fonction $func$ avec les paramètres $arg[i]$

2.6.4 Hybride MPI+OpenMP

Le modèle hybride MPI+OpenMP est vu comme le modèle adéquat pour les grappes de SMP [4, 19, 20]. Le principe de base de ce modèle est l'utilisation de la parallélisation à mémoire distribuée via MPI pour la communication entre les nœuds du cluster et OpenMP pour la parallélisation à mémoire partagée dans un nœud. L'approche OpenMP grain fin est considérée comme la plus simple pour construire un code hybride à partir d'un code MPI. Elle consiste à détecter les boucles qui contribuent le plus au calcul dans un processus MPI et de distribuer leurs itérations entre les différents cœurs du nœud en utilisant la directive OpenMP.

2.7 Outils pour la programmation des accélérateurs

2.7.1 Outils pour le Cell BE

IBM a fourni un SDK [4] pour la programmation des accélérateurs multi-cœurs hétérogènes de type Cell BE. Cette bibliothèque est constituée d'un ensemble de fonctions et de procédures de bas niveau implémentées en langage C. En utilisant ces fonctions, le développeur doit prendre en considération tous les aspects architecturaux et fonctionnels de l'accélérateur.

Vu la difficulté et la complexité de la programmation avec les outils IBM SDK, plusieurs propositions sous forme de bibliothèques et de compilateurs ont tenté de diminuer cette complexité et de faciliter la programmation des processeurs Cell BE. Parmi ces nombreux outils, on peut citer OpenMP pour le Cell [21], la bibliothèque CellSS [22] et une implémentation d'un sous ensemble de routines MPI pour le Cell [23].

L'OpenMP pour Cell a été proposé par IBM sous forme d'un compilateur. Dans cette proposition, le Cell est vu comme un multi-cœurs à mémoire partagée. Le compilateur distribue les tâches entre les SPEs en utilisant des directives OpenMP. Bien que cette solution soit de haut niveau et facilite la programmation de cette architecture, elle n'a pas eu un grand succès à cause des performances dégradées et médiocres comparées à la performance crête du processeur.

La bibliothèque CellSS ressemble à OpenMP en terme de simplicité d'utilisation. Le modèle CellSS est basé sur l'annotation du programme source pour

4. *Software Development Kit*

distribuer les tâches entre les SPEs. (voir chapitre 3)

2.7.2 Outils pour le GPU

Pour la programmation des processeurs graphique GPU, il existe essentiellement deux langages : CUDA et OpenCL.

CUDA [7] pour Compute Unified Device Architecture est une architecture à usage général pour le calcul parallèle - avec un modèle de programmation parallèle et un nouveau jeu d'instructions qui tire parti du parallélisme dans les GPUs. CUDA est une extension du langage C proposée par NVIDIA au profit de la programmation des cartes graphique de classe Gforce8 ou supérieur. CUDA permet d'identifier certaines fonctions comme destinées à être traitées par le GPU au lieu du CPU. Ces fonctions (Kernels) sont alors compilées par un compilateur spécifique à CUDA ce qui leur permet d'être exécutées par les nombreuses unités de calcul des GPUs.

Le modèle de programmation CUDA est basé sur la notion des threads. En effet, chaque kernel (fonction exécutée sur le GPU) démarre des centaines de threads CUDA qui sont regroupés en une série de blocs. Tous les threads dans un bloc s'exécutent sur le même SM, et peuvent se synchroniser et s'échanger des données en utilisant la mémoire partagée. Cependant, la synchronisation entre les blocs de threads n'est pas possible. Un autre niveau d'encapsulation est possible par le regroupement des blocs sous forme de $Grids$ indépendant. (Voir la figure 2.7).

OpenCL [24] est un standard ouvert et gratuit pour la programmation parallèle dans un environnement informatique hétérogène. L'utilisation la plus commune de OpenCL est pour la programmation des GPUs. Comme CUDA, OpenCL étend le langage C pour permettre l'écriture des kernels. Contrairement à CUDA, OpenCL est utilisé pour différents types de carte (NVIDIA, AMD ...).

2.8 Conclusion

Plusieurs types d'architectures sont donc utilisés pour satisfaire la demande des utilisateurs en terme de calcul haute performance. On trouve d'une part des grappes de multiprocesseurs et avec l'apparition des multi-cœurs et leur démocratisation, même les stations de travail personnelles offrent des performances et des capacités de parallélisme époustouflantes. Dans cette optique, les grappes de multiprocesseurs multi-cœurs règnent depuis des années sur le domaine du calcul haute performance. D'autre part, on trouve les architectures hétérogènes qui

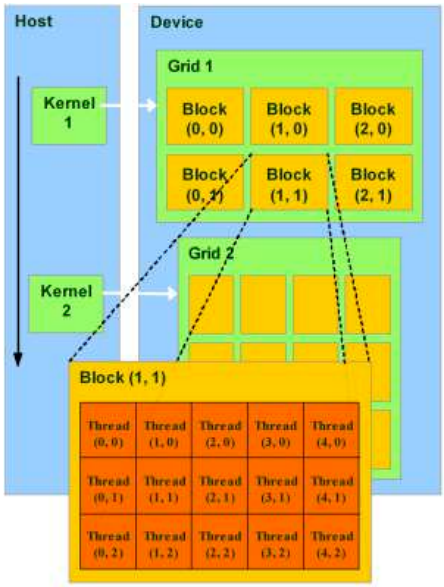


FIGURE 2.7 – Architecture du modèle de programmation CUDA

utilisent des accélérateurs de type Cell BE pour augmenter leurs performances crêtes. Le cluster *Roadrunner* pour citer un exemple atteint des performances de $1,7$ pétaflops.

Cependant, bien que les performances crêtes de ces architectures satisfassent les besoins des utilisateurs, leurs outils de programmation restent toujours complexes et difficiles à utiliser. En effet, on a vu dans ce chapitre différentes bibliothèques, outils et langages de bas niveau permettant de tirer profit des performances offertes par ces architectures. Cependant, l'aspect bas niveau de ces outils limite la maîtrise d'utilisation et de programmation aux experts des architectures parallèles.

Les utilisateurs des ces machines sont pour la plupart des chercheurs dans des sciences éloignées de l'informatique telles que la physique, la chimie ou la biologie. Pour ces utilisateurs, l'exploitation de ces machines parallèles est très importante. Il y a donc un besoin d'outils et de modèles de programmation parallèle offrant à la fois une haute abstraction de la complexité de la machine cible et des performances proches de celles des experts des outils de bas niveau précédemment mentionnés pour permettre à ces utilisateurs de bénéficier de ces performances avec un coût de développement réduit.

Le prochain chapitre présentera quelques outils et modèles de programmation de haut niveau existant dans la littérature.

MODÈLES ET OUTILS DE CALCUL HAUT NIVEAU 3

Le chapitre précédent a montré que les architectures de calcul haute performance sont hiérarchiques et offrent des capacités crêtes satisfaisantes. Aussi, on a vu qu'il est possible, avec des outils comme MPI, OpenMP et la SDK d'IBM de tirer parti efficacement du parallélisme offert par ces architectures. Néanmoins, la complexité de ces outils due au fait qu'ils sont de bas niveau nécessite une expertise et une prise en main non-triviale pour exploiter au maximum leur potentiel. Dans le cadre de MPI, la principale difficulté provient de l'élaboration

de schémas de communication et la distribution des données et des tâches entre les différentes unités de calcul. Cette phase de mise au point est critique car elle conditionne les performances qui dépendent essentiellement du ratio des temps calcul/communication. C'est aussi la première source d'erreurs comme l'interblocage qui ralentissent énormément le développement des applications. Concernant OpenMP, même s'il est plus simple que MPI, il souffre des mêmes lacunes que ce dernier en termes de partition des charges et des surcoûts introduit par la composition des différents directives. L'utilisation de la SDK IBM pour la programmation des processeurs Cell BE souffre de son interface de bas niveau et surtout du fait que c'est au programmeur de prendre en charge tous les aspects de programmation et les caractéristiques architecturales de la machine. Cela va de la répartition des données et des tâches entre les SPUs à l'alignement des données en mémoire en passant par la gestion des transferts.

Ainsi, et afin de faciliter et de démocratiser l'utilisation de ces architectures, il faut des outils de programmation de haut niveau offrant à la fois une **simplicité** d'utilisation et une **efficacité** en terme de performances : de tels outils doivent procurer une haute abstraction de la complexité de l'architecture du point de vue du programmeur et de garantir des performances équivalentes à celles offertes par un code bas niveau écrit par un expert de la programmation des ces machines. En plus de la simplicité et de l'efficacité, ces outils doivent offrir un moyen pour prédire les performance. Cette dernière caractéristique vise à alléger la difficulté de conception et d'analyse des algorithmes parallèles ainsi que le choix de l'architecture cible.

L'objectif de nos travaux et de fournir des modèles de programmation et des outils permettant l'exploitation des différents niveaux de parallélisme offerts par les architectures hybrides de type cluster de multi-cœurs et des grappes équipés des accélérateurs Cell BE. Ces outils ont pour objectif une haute abstraction permettant une simplicité et une facilité d'utilisation tout en mettant l'accent sur l'efficacité pour permettre de conserver un niveau de performances comparable à celui des programmes avec des outils de bas niveau.

Il existe plusieurs outils de haut niveau pour la programmation des machines parallèles. Ces outils peuvent être classés en deux catégories d'approches : approches structurées sur des modèles et approches non structurées. Les sections suivantes dressent un panorama des différentes solutions des deux catégories et présentent les avantages et les inconvénients de chacune d'elles.

3.1 Approches non structurées sur un modèle

Les solutions de ce type ne se basent sur aucun modèle de calcul haut niveau. En effet, la plupart de ces approches utilisent des langages de programmation qui visent à faciliter le développement des applications parallèles. Parmi ces outils, on trouve :

3.1.1 StarPU

Probablement StarPU est le plus proche de nos propositions qu'on va présenter par la suite. StarPU est un outil proposé par INRIA Bordeaux [25] pour faciliter l'utilisation des architectures hétérogènes. La portabilité des codes entre les multicœurs et les accélérateurs est définie par l'utilisation des *codelets*. Une codelet est une structure qui représente une fonction de calcul et cette structure peut contenir la même fonction implémentée différemment pour chaque architecture cible (CPU, Cell et GPU). Cependant, l'écriture de ces implémentations ainsi que la partitionnement des données restent à la charge d'utilisateur.

En plus de la génération de code pour les différentes architectures, StarPU implémente un ordonnanceur pour placer efficacement les différentes parties de calcul (codelets) sur les différentes ressources de la machine cible. Toutefois, les politiques de cet ordonnanceur se basent soit sur des mesures au runtime soit sur l'aide de l'utilisateur pour donner cette estimation.

3.1.2 Charm++

Charm++ [26] est un langage de programmation parallèle orienté objet basé sur le langage C++, développé par le laboratoire de programmation parallèle de l'Université de Illinois. Les programmes écrits en Charm++ sont décomposés en un certain nombre d'objets (classes) communiquant via un système de passage de messages appelé $CHARES$. Quand un programmeur appelle une méthode sur un objet, le système d'exécution de Charm++ envoie un message à l'objet invoqué, qui peut se situer sur le processeur local ou sur un processeur distant dans un calcul parallèle. Ce message déclenche l'exécution du code de cette méthode. Un code Charm++ peut être exécuté sur un cluster de SMPs. Une extension pour les accélérateurs de type Cell a été proposé par [27].

Basé sur la programmation orientée objet de C++, le style de développement de Charm++ permit une simplicité d'utilisation. En effet, le programmeur instancie des classes et fait des invocations des méthodes pour développer un code parallèle. Cependant l'utilisateur est en charge de synchroniser tous les appels et de spécifier les parties du code qui doivent être exécutées sur les accélérateurs .

3.1.3 Sequoia

Sequoia [28] est un langage de programmation axé sur l'intervention constante du programmeur dans la structuration des programmes parallèles. L'idée principale de ce langage est que le mouvement et l'emplacement des données à tous les niveaux de la hiérarchie mémoire de la machine, devraient être sous contrôle explicite de programmeur via un mécanisme du langage.

Les auteurs ont mis en place un système de programmation complet autour de cette idée, comportant un compilateur et un système d'exécution pour le processeur CellBE et les grappes à mémoire distribuée. La conception de Sequoia est centrée autour des idées suivantes :

– L'introduction de la notion de mémoire hiérarchique directement dans le modèle de programmation permet de gagner à la fois la portabilité et la performance. Les programmes Sequoia fonctionnent sur des machines qui sont sous forme d'arbres de modules de mémoires. Ils décrivent comment les données sont déplacées et où elles résident dans la hiérarchie mémoire d'une machine.

– L'utilisation des tâches comme des unités d'abstraction autonomes de calcul. Elles comprennent des descriptions et des informations clés telles que la communication et les sous tâches de calcul. Les tâches isolent chaque calcul dans son propre espace d'adressage local.

– Afin d'assurer la portabilité, il y a une stricte séparation entre l'expression générique des algorithmes et les optimisations spécifiques à la machine. Pour minimiser l'impact de cette séparation sur les performances, les détails de contrôle spécifiques à la machine sont exposés aux programmeurs.

Ce langage prend en considération les deux types d' architectures visées par nos recherches, à savoir les grappes de multi-cœurs et les grappes avec des Cell BE. Malheureusement, le niveau de programmation reste un peu compliqué car l'utilisateur doit réaliser les tâches de décomposition et de placement de son application sur les différents niveaux de la hiérarchie mémoire de la machine.

3.1.4 Bibliothèques CellSs et SMPs

Dans le paradigme intermédiaire entre la programmation séquentielle et parallèle, il y a le *framework* Cell Superscalaire (CellSs) [22], qui est basé sur deux techniques : une compilation source à source et une bibliothèque d'exécution (runtime). Le modèle de programmation pris en charge permet aux programmeurs d'écrire des applications séquentielles et le framework est en mesure d'ex-

ploiter la concurrence existante entres les tâches de l'application et d'utiliser les différents composants du processeur Cell BE (PPE et SPE) au moyen d'une parallélisation automatique au moment de l'exécution. CellSs exige que le programmeur écrive des annotations (en quelque sorte similaire à ceux OpenMP) avant la déclaration de certaines des fonctions utilisées dans l'application.

Pour être en mesure d'exploiter le parallélisme, le runtime CellSs construit un graphe de dépendance de données où chaque nœud représente une instance d'une fonction annotée et les arcs entre les nœuds désignent les dépendances de données. De ce graphe, la bibliothèque d'exécution est en mesure d'ordonnancer les fonctions indépendantes sur différents SPEs en même temps. Basée sur un compilateur et une bibliothèque d'exécution, cette approche utilise des techniques importées de domaines de l'architecture des ordinateurs et des compilateurs, comme l'analyse de dépendance de données, le renommage des données et l'exploitation de la localité des données.

Pour les architectures SMP multi-cœurs, les auteurs proposent le framework SMPSs [29] (SMP Superscalaire) qui suit la même stratégie que CellSs pour ordonnancer des tâches indépendantes sur les différents cœurs d'un nœud SMP. Ainsi le même code écrit pour les accélérateurs Cell BE peut être exécuté sur un nœud SMP.

CellSS et SMPs offrent une simplicité d'utilisation grâce à la parallélisation semi-automatique. Cependant, la génération d'un graphe de tâches et la détection des dépendances à l'exécution limite son efficacité aux problèmes triviaux.

3.1.4.1 XscalableMP

XcalableMP [30] est une extension du paradigme de directives pour permettre le développement de programmes parallèles pour les systèmes à mémoire distribuée. XcalableMP Application Program Interface (API XcalableMP) fournit une collection de directives de compilation, qui sont des routines de la bibliothèque d'exécution utilisées pour spécifier ce type de programmation. Les directives comme dans OpenMP étendent les langages C et Fortran de base pour décrire les programmes parallèles à mémoire distribuée. Cette spécification définit un modèle de programmation parallèle pour les grappes.

XcalableMP utilise la parallélisation de données. Il permet la parallélisation du code original séquentiel en utilisant un minimum de modifications via l'usage des directives. Le principe de conception de XcalableMP est "la performance" : toutes les opérations de communication et de synchronisation sont effectuées par des directives, qui ne sont pas incluses dans la parallélisation automatique

du compilateur. La connaissance du modèle d'exécution sur ces architectures, par l'utilisateur, est exigée pour faciliter le réglage des performances.

Comme CellSS, XcalableMP offre un moyen simple pour le développement des applications. Toutefois, cette bibliothèque est limitée aux grappes de multi-cœurs et ne prend pas en considération les architectures avec accélérateurs.

Il existe d'autres outils basés sur des compilateurs comme Par4all [31] et PGI [32] pour la programmation de différents types d'architectures. Cependant, ils sont concentrés sur la génération de code pour les GPUs à partir d'un code OpenMP.

Toutes ces solutions (approche non structurée) ont en commun l'inconvénient de ne permettre de prédire les performances. En effet, comme elles sont basées sur des modèles ad hoc (utilisation de primitives, de classes ou de fonctions) spécifiques pour l'expression de parallélisme, elles n'offrent aucun moyen pour la prédiction des performances à partir du code source.

3.2 Approches structurées sur un modèle

Dans [33], Skillicorn et Talia passent en revue les modèles et langages de programmation parallèle. Six critères ont été utilisés pour évaluer leur aptitude à la programmation des machines parallèles. Quatre de ces critères sont liés aux besoins des développeurs de logiciels, et les deux autres répondent aux besoins de l'exécution effective des modèles sur des machines parallèles réelles. Les auteurs ont conclu que le modèle idéal devrait avoir ces caractéristiques :

- **Facilité de programmation :** Quel que soit le domaine, la programmation d'applications est une tâche complexe. Aussi, un modèle de programmation doit cacher autant que possible les détails bas niveau liés à la mise en œuvre effective du parallélisme sur l'architecture cible.

- **Facilité de compréhension :** Un modèle de programmation doit être facile à comprendre sinon son utilisation ne peut être large et reste limitée à un public restreint de spécialistes. Si un modèle est capable de masquer la complexité de programmation (illustrée au point 1) et de fournir une interface de développement simple et conviviale, alors celui ci a de fortes chances d'attirer bon nombre de programmeurs désirant accroître les performances de leurs applications.

- **Indépendance de l'architecture :** Un modèle se doit d'être décorrelé de toute architecture cible sur laquelle le programme doit s'exécuter. En effet,

l'expérience montre que les applications possèdent une espérance de vie largement supérieure à celle des machines. A chaque changement même minime de l'architecture, il est très coûteux d'effectuer une phase de redéveloppement plus ou moins profonde de l'application. Aussi, un modèle indépendant du matériel offre des opportunités de portabilité des applications non négligeables. De plus, l'abstraction vis à vis de la plate-forme cible facilite la programmation. Il n'est plus nécessaire de maîtriser les caractéristiques de l'architecture pour effectuer du développement d'applications parallèles. Ce facteur contribue fortement à un élargissement du champ des utilisateurs potentiels.

- **Mesures prévisibles :** L'implantation d'une application sur une architecture parallèle a généralement pour premier objectif une réduction du temps d'exécution. Or, dans la majorité des cas, la validation ou non de ce but n'est possible que lors de l'exécution effective du programme. Un modèle de programmation parallèle doit donc fournir un ensemble d'outils de mesures prédictives autorisant l'évaluation précise des performances avant toute implantation réelle. Cette mesure prédictive facilite de même le développement de nouveaux algorithmes.

- **Outils de développement :** Les quatre premiers points discutés impliquent qu'il existera un large fossé entre la spécification de l'application fournie par l'utilisateur et les informations nécessaires à l'exécution du programme sur l'architecture. Ces informations manquantes concernent, par exemple, la décomposition en tâches, le placement-ordonnancement, les communications au niveau application et le nombre de processeurs, le réseau d'interconnexion au niveau matériel. Il est donc nécessaire d'associer à tout modèle de programmation un ensemble d'outils intégrés effectuant la transformation de la spécification minimale utilisateur en une représentation détaillée et implémentable des programmes.

- **Efficacité des applications :** Enfin, un modèle se doit d'être efficacement implémentable sur une ou plusieurs architectures parallèles. Le but n'est pas forcément d'obtenir la meilleure implantation possible pour la machine cible mais d'établir un compromis idéal entre performance et facilité de développement. En effet, il est indéniable qu'un programmeur expérimenté obtiendra toujours de bien meilleures performances que celles que pourrait donner le plus sophistiqué des outils de parallélisation, mais au détriment d'un coût de développement prohibitif.

Ces critères reflètent notre conviction que l'évolution du parallélisme doit être conduit par une industrie du logiciel parallèle basée sur la portabilité, l'efficacité, la facilité d'utilisation et la caractère prévisible des performances. Dans cette section, nous allons présenter un ensemble de modèles de calcul et de programma-

tion de haut niveau.

3.2.1 Squelettes algorithmiques

Les langages à patrons comme les squelettes algorithmiques [34, 35, 36, 37] sont une alternative qui consiste à utilisé un ensemble réduit d'opérateurs prédéfinis – les squelettes – qui se comportent comme des fonctions d'ordre supérieures dont la sémantique parallèle est implicite. Ces langages en général, et les Squelettes algorithmiques en particulier, sont souvent issus d'une collection de structures orientées métiers qui encapsulent des stratégies récurrentes. Il existe une large sélection de bibliothèques et de langages basés sur ce modèle, parmi lesquels on peut citer :

3.2.1.1 P3L

P3L (Pisa Parallel Programming Language) [38, 39] développé par l'université de Pise et Hewlett-Packard est un système parallèle de programmation basé sur les squelettes de parallélisation. P3L est un langage de *haut niveau structuré* permettant l'expression $explicite$ du parallélisme [40] :

- Premièrement, P3L est un langage $haut\ niveau$ car la programmation des applications n'utilise pas des tâches et instructions bas niveau relatives au développement de programmes parallèles : placement et ordonnancement des processus, communications, synchronisations, etc.

- Deuxièmement, P3L est un langage *structuré* car l'opportunité est offerte au programmeur d'exprimer son application sous la forme d'une composition hiérarchique et éventuellement imbriquée de squelettes.

- Troisièmement, P3L est un langage $explicite$ puisque l'expression du parallélisme est obligatoirement et entièrement effectuée par le programmeur. Celui-ci a donc pour tâche de décrire et de déclarer quels schémas de parallélisation doivent être utilisés.

P3L est basé uniquement sur un langage impératif séquentiel (langage C). Les squelettes sont exprimés au sein de ce langage et y apparaissent comme des constructeurs sous la forme de mots-clés spécifiques.

3.2.1.2 eSkel

eSkel (edinburgh Skeleton library) est une bibliothèque à base des squelettes algorithmiques construite en utilisant C et MPI. Elle a été développée par le groupe Structure et Parallélisme à l'Université d'Edimbourg [35]. Le code source

de la version actuelle de la bibliothèque ainsi que ses manuels et les publications qui la référencent sont librement accessibles en ligne.

Le modèle de mise en œuvre de eSkel est basé sur le paradigme de programmation SPMD. La bibliothèque est utilisée avec un programme C, (ou un programme capable d'effectuer des appels de fonction C). Hérité de MPI, les opérations de eSkel doivent être appelées à partir d'un programme qui a déjà initialisé un environnement MPI. En effet, il s'agit d'une couche supérieure composante MPI, qui étend les fonctionnalités de MPI. eSkel a été testée avec plusieurs versions de MPI.

Pour plus de flexibilité de programmation, eSkel étend les objets de communication MPI en introduisant la notion des eSkel Data Model (eDM). Cette notion s'appuie sur l'approche de MPI qui considère les objets communiquant comme des triplés (pointeur, longueur, type) en utilisant deux structures : Atome eDM et Collection eDM : Un atome eDM est en réalité un triple MPI, complété par une étiquette, indiquant si les données fournies par chaque processus participant est un objet local, ou un objet formé par la concaténation des données fournies par tous les processus dans le groupe. Une collection eDM est simplement une séquence d'Atomes eDM pour permettre au programmeur de spécifier les données qu'un squelette traite.

3.2.1.3 Sketo

Le projet sketo [41] est une bibliothèque C++ utilisant MPI. Sketo est différente des autres bibliothèques basées sur les squelettes algorithmiques car au lieu de fournir des modèles de parallélisme emboîtables, sketo fournit des squelettes parallèles pour les structures de données parallèles, tels que : liste, arbre [42] et matrice [43]. Les structures de données sont génériques grâce à l'utilisation des Templates C++, et plusieurs opérations parallèles peuvent être invoquées sur eux. Par exemple, la structure de liste fournit des opérations parallèles telles que : map, reduce, scan, zip, shift, ...

Des recherches supplémentaires autour de sketo ont également mis l'accent sur les stratégies d'optimisation par transformation, et plus récemment des optimisations orientées domaines. Dans [44] , par exemple, sketo prévoit une transformation de fusion de l'invocation des fonctions successives en un seul, réduisant ainsi les frais généraux d'appel de fonction en évitant la création de structures de données intermédiaires passées entre les fonctions.

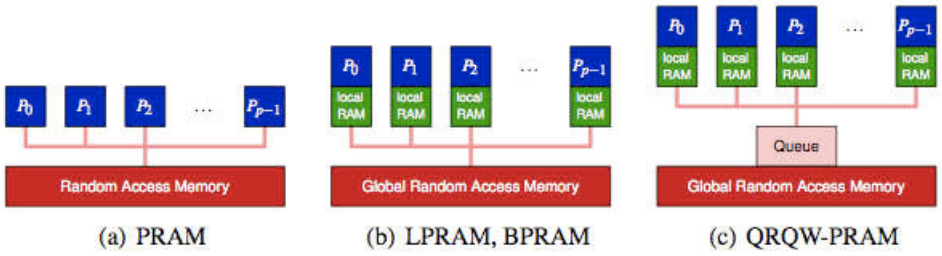
Les avantages des squelettes algorithmiques sont bien sûr le déterminisme et l'absence du blocage. Cependant, le concepteur de bibliothèques de patrons doit

faire face à deux problèmes majeurs qui sont opposés. D'une part, il doit fournir un ensemble de patrons le plus complet et le plus expressif possible. Trop restreint, celui-ci serait inutilisable en pratique. D'autre part, il faut que cet ensemble soit le plus petit possible car il faut implanter efficacement chaque patron sur chaque type ciblé de machine parallèle. De plus, s'il y a pléthore de patrons, le programmeur qui en dispose aurait les plus grandes difficultés à choisir celui qui convient le mieux à son problème. Une autre difficulté des squelettes est que certains patrons n'offrent pas un modèle pour la prédiction des performances. Et leur combinaison rend la prédiction encore plus complexe. Par exemple, la combinaison d'un $Farm$ et d'un $Pipeline$ rend difficile la prévision des performances car $Farm$ est un patron asynchrone.

Pour remédier à ce manque, les squelettes algorithmiques sont souvent combinés avec d'autre modèles de calcul haut niveau qui possèdent un modèle de prédiction des performances. Ce qui suit présente ces modèles.

3.2.2 Famille PRAM

PRAM est un modèle de calcul parallèle qui suppose la présence d'un nombre de processeurs fonctionnant de manière synchrone en parallèle et d'avoir accès à une seule mémoire globale partagée [45]. Ce modèle peut être considéré comme une extension naturelle du modèle séquentiel classique. Il est composé d'une unité centrale de traitement CPU et d'une mémoire à accès aléatoire qui s'y rattache. Le CPU peut lire et écrire dans n'importe cellule de la mémoire. Le modèle PRAM a été largement utilisé, notamment par la communauté d'informatique théorique pour la conception et l'analyse des algorithmes parallèles combinatoires et de théorie des graphes.



(a) PRAM (b) LPRAM, BPRAM (c) QRQW-PRAM

FIGURE 3.1 – Les machines PRAM

Dans le modèle PRAM, les P processeurs peuvent lire et écrire dans des cellules arbitraires de la mémoire partagée. Les processeurs ont une horloge centrale et exécutent chaque instruction d'un programme donné de façon synchrone. Toutes les communications entre les processeurs se font par l'écriture et la lecture

de la mémoire globale (Figure 3.1). Il existe plusieurs variantes de PRAM, selon la façon dont l'accès simultané à la même cellule de la mémoire partagée est manipulé. Les principales variantes sont les suivantes.

- Lectures exclusives et écritures exclusives (EREW PRAM) : la version la plus restrictive, en sens qu'elle ne permet pas l'accès simultané à un seul emplacement mémoire par des processeurs différents. Notez cependant que l'accès simultané à différents emplacements de la mémoire est une caractéristique fondamentale de toutes les variantes du modèle PRAM.

- Lectures concurrentes et écritures exclusives (CREW PRAM) : une version permettant un accès simultané pour une opération en lecture seule. Les opérations d'écritures simultanées sur la même cellule ne sont pas autorisées.

- Lectures concurrentes et écritures concurrentes (CRCW PRAM) : une version qui permet la lecture et l'écriture à un seul emplacement mémoire. Les opérations d'écritures simultanées à un seul emplacement mémoire peuvent être résolues de différentes manières, y compris le choix arbitraire du processeur écrivant (arbitraire CRCW), ou l'utilisation de la priorité ; le processeur qui a la plus haute priorité réussit à écrire (priorité CRCW), ou encore, en insistant pour que tous les processeurs écrivent la même valeur dans la même zone mémoire (commun CRCW).

Plusieurs extensions de PRAM ont été proposées pour prendre en considération la localité des données et l'accès par bloc.

3.2.2.1 Le modèle LPRAM : Local-memory PRAM

Le modèle PRAM à mémoire locale a été introduit par Alok Aggarwal, Ashok K. Chandra et Marc Snir [46]. Le principe de base de ce modèle est l'utilisation d'une hiérarchie de mémoires. En plus de la mémoire globale, chaque processeur a sa propre mémoire privée locale. A chaque étape, un processeur peut lire ou écrire un mot dans la mémoire locale ou globale, ou effectuer une étape de calcul local. L'accès à la mémoire globale est modélisé par la politique CREW (lectures concurrentes, écriture exclusives).

3.2.2.2 Le modèle BPRAM : Block-PRAM

Le modèle BPRAM [47] (PRAM à blocs) est une extension du modèle LPRAM. Dans ce modèle, l'accès à la mémoire globale partagée est défini en terme de bloc. Cela signifie qu'un processeur peut transférer un bloc de cellules consécutives de mémoire à partir de la mémoire globale vers la mémoire locale. La même chose

est possible dans la direction opposée. Le coût d'une copie d'un bloc de données est $b + l$, où b est la taille du bloc et l est la latence (un paramètre de la machine). Dans le modèle BPRAM, différents accès ne sont pas autorisés à se chevaucher, autrement dit, la mémoire partagée est modélisé par EREW (lecture exclusive, écriture exclusive). Dans ce modèle, le temps minimal de fonctionnement et le travail exécuté par les transformateurs sont des mesures importantes de la performance de l'algorithme.

3.2.2.3 Le modèle PRAM Asynchrone : APRAM

Ce modèle de Phillip B. Gibbons [48] est aussi un modèle PRAM qui intègre également à la fois une mémoire locale et une globale. Cependant, contrairement aux modèles PRAM précédents, où tous les processus utilisent la même horloge, chaque processeur du modèle APRAM a sa propre horloge locale et exécute les instructions de son propre programme indépendamment des autres processeurs. Ce modèle introduit une nouvelle instruction pour synchroniser arbitrairement un groupe de processeurs : tous les processeurs dans ce groupe attendent jusqu'à ce que tout le monde ait atteint la barrière de synchronisation.

3.2.3 Le modèle CGM : Coarse Grained multicomputer

Le modèle Coarse Grained Multicomputer CGM a été proposé par Dehne, Fabri, et Rau-Chaplin [49, 50] en 1993. Ce modèle comporte P processeurs, $O(N/P)$ mémoires locales sur chaque processeur (N la taille globale des données) et un réseau de communication arbitraire. La taille de la mémoire locale est typiquement considérée plus grande que $O(1)$ et en pratique, le modèle assume que N/P est supérieur à P^e pour e > 0. Cette caractéristique du modèle lui donne le nom de "$Coarse\ Grained$".

Les algorithmes écrits dans ce modèle sont composés d'une succession de deux phases :

- Une phase où chaque processeur effectue un calcul local sur ces données locales.
- Une phase où les processeurs s'échangent les données.

Pendant cette deuxième phase, chaque processeur peut envoyer $O(N/P)$ données et recevoir $O(N/P)$ données. Ainsi dans un modèle CGM à P processeurs, chaque processeur a une mémoire de $O(N/P)$. La communication est globale entre tous les processeurs et peut utiliser n'importe quel réseau de connexion.

3.2.4 Le modèle LogP

Dans [51], Culler et al ont développé un modèle de calcul pour les architectures multiprocesseurs à mémoire distribuée, dans lequel les processeurs communiquent par messages point à point. Le modèle spécifie les caractéristiques de performance du réseau d'interconnexion, mais ne décrit pas la structure du réseau. Les principaux paramètres du modèle sont les suivants :

- L : Une limite supérieure sur la latence, ou le retard, engagés dans la communication d'un message contenant un mot de processeur source au processeur cible.
- o : Les frais de surcharge (overhead), définis comme la durée pendant laquelle un processeur est engagé dans la transmission ou la réception de chaque message. Pendant ce temps, le processeur ne peut pas effectuer d'autres opérations.
- g : L'écart, défini comme l'intervalle de temps minimal entre la transmission ou la réception des messages consécutifs. L'inverse de g correspond la bande passante.
- P : Le nombre de processeurs/mémoire.

En outre, le modèle suppose que le réseau a une capacité limitée, de sorte qu'au maximum L/g messages peuvent être en transit dans le réseau à tout moment. Si un processeur tente de transmettre un message qui dépasse cette limite. il est mis en attente jusqu'à ce que ce message puisse être envoyé sans dépasser cette limite.

3.2.5 Le modèle LogGP

Le modèle LogGP a été proposé par Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, et Chris Scheiman [52] comme une extension du modèle LogP pour prendre en considération la transmission des gros messages. En effet, comme en témoignent les données expérimentales recueillies par [53] , le modèle LogP peut prédire avec précision les performances de communication uniquement pour les messages de petite taille . En plus des paramètres du modèle LogP, le modèle LogGP ajoute un autre paramètre : G : l'écart pour les messages longs, défini comme étant le temps par octet pour un message de grande taille. La réciproque de G caractérise la bande passante du réseau.

3.2.6 Le modèle BSP

Le modèle Bulk Synchronous Parallel (BSP) a été introduit par Leslie G. Vaillant [54] comme un pont entre le matériel et le logiciel pour simplifier le développement d'algorithmes parallèles. Du point de vue du développeur, le modèle fournit une abstraction sur l'architecture de la machine et les caractéristiques du réseau. D'un autre point de vue, le modèle simplifie la tâche des concepteurs des machines parallèles en faisant une abstraction sur les applications à exécuter. Classiquement, le modèle BSP est défini par trois éléments :

- **Modèle de machine** : il décrit la machine parallèle comme un ensemble de processeurs reliés par un moyen de communication supportant des communications point-à-point et des synchronisations. Ces machines sont alors décrites par un ensemble de paramètres expérimentaux [55] :
  - P : le nombre de processeurs.
  - r : la vitesse du processeur en flops.
  - g : la vitesse du moyen de communication : le débit de réseau ou la vitesse d'accès à la mémoire pour les architecture à mémoire partagée.
  - L : la durée de synchronisation.
- **Modèle de programmation** : il décrit comment un programme parallèle est structuré (Figure 3.2). Un programme BSP se compose d'une séquence de super-étapes dans lesquelles chaque processus effectue les calculs locaux suivis des communications. Lorsque tous les processus atteignent la barrière de synchronisation, la prochaine super-étape commence.

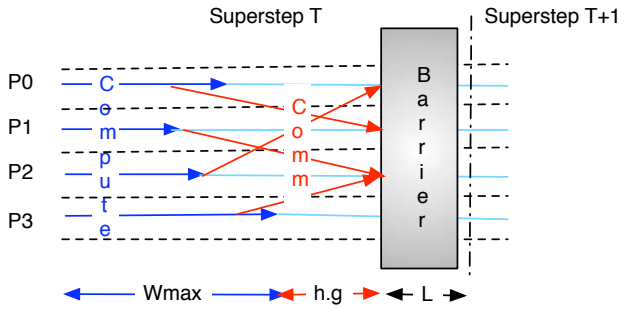


FIGURE 3.2 – Une vue du modèle de programmation BSP

- **Modèle de coût** : Le temps nécessaire à l'exécution d'une super-étape s est la somme :
  - Du maximum des temps de calculs locaux ;
  - Du temps de la réalisation des échanges entre les processeurs ;
  - Du temps de la réalisation d'une barrière de synchronisation globale.

On l'exprime par la formule suivante :

$$\delta_s = \max_{0<i<=P} W_i + \max_{0<i<=P} h_i * g + L$$

où W_i est le temps de calcul local sur le processeur i pendant cette super-étape et $h_i = max(h_{i+}, h_{i-})$ où h_{i+} (resp. h_{i-}) est le nombre de mots envoyés (resp. reçus) par le processeur i durant la super-étape.

Le temps d'exécution total d'un programme BSP composé de S super-étapes est donc la somme de temps de chaque étape :

$$T_{total} = \sum_s \delta_s$$

3.2.7 Discussion

A la lumière des critères d'un modèle parfait et nos objectif présentés précédemment, le modèle BSP satisfait le plus nos exigences en terme de simplicité, efficacité et prédictibilité des performances. Dans ces paragraphes, les raisons de notre choix sont exposés sous forme de comparaisons entre le modèle BSP et les trois autres modèles à savoir PRAM, LogP et CGM.

3.2.7.1 BSP vs PRAM

Le modèle BSP peut être vu comme une généralisation du modèle PRAM. En effet pour des architectures avec une valeur de g très petite (g=1), le modèle BSP est équivalent au modèle PRAM. Dans ce cas, en utilisant des tables de hachage, le modèle BSP peut obtenir automatiquement une gestion efficace de la mémoire. En plus, la valeur de l détermine le degré de relâchement parallèle nécessaire pour atteindre une efficacité optimale. En effet, la configuration (l = g = 1) correspond à la machine PRAM idéalisée où aucun relâchement n'est nécessaire.

En plus, comme mentionné précédemment, le modèle PRAM et ses extensions sont des modèles pour des machines à mémoire partagée. Par conséquence, ils ne peuvent pas être utilisés pour des architectures de type $Cluster$.

3.2.7.2 BSP vs CGM

Comparé au modèle BSP, la phase de calcul dans le modèle CGM est équivalent à une super-étape dans le modèle BSP avec $L = g(N/p)$. Et la phase de communication se compose d'une h-relation unique avec $h < N/p$. La principale différence entre les modèles BSP et CGM est que ce dernier est limité à un seul type d'opération de communication, la h-relation, et par conséquent, compte

simplement le nombre de $h - relation$ comme son principal critère d'évaluation des coûts de communication. Notez que BSP est plus général que CGM : tout algorithme CGM est également un algorithme BSP mais le contraire n'est pas vrai.

3.2.7.3 BSP vs LogP

Chaque modèle peut simuler efficacement l'autre. Cependant, BSP diffère de LogP de trois points :

- LogP utilise une forme de passage de messages basée sur la synchronisation par paires alors que BSP utilise une synchronisation globale. Ceci offre plus d'abstraction pour la conception d'outils.

- LogP ajoute un paramètre supplémentaire représentant les frais généraux liés à l'envoi d'un message qui s'applique à toute communication. Ce paramètre est difficile à quantifier vu qu'il dépend du runtime de la machine.

- BSP a une vue globale de g alors que LogP a une vision locale de g.

Concernant la prédiction des performances du modèle LogP , il est souvent nécessaire (ou pratique) d'utiliser des barrières. Ce qui donne :

$$LogP + barriers - overhead = BSP$$

L'avantage de BSP par rapport à logP est bien évidemment l'existence d'un modèle de programmation structuré sur la notion d'étapes qui facile énormément la conception et le développement de codes parallèles. En effet, comme le modèle de programmation de BSP est basé sur un style SPMD et les calculs asynchrones de chaque étape sont des fonctions séquentielles, la conception d'algorithmes parallèles BSP revient simplement à l'élaboration des schémas et des patterns de communication.

3.3 Extensions de BSP

De nombreuses extensions du modèle de base BSP ont été proposées depuis 1990. Certaines d'entre elles essayent d'améliorer la précision du modèle de coût en utilisant plusieurs paramètres pour décrire la machine BSP. D'autres visent à réduire le coût de la synchronisation globale L qui est vu comme l'objection majeur à l'utilisation de BSP pour les machines avec un grand nombre de processeurs. Le coût d'une synchronisation globale L peut être assez dominant (bien que dans la plupart des systèmes, il est de l'ordre ($log\ p$) pour p processeurs).

La figure 3.3 donne un aperçu de quelques-uns de ces modèles et les relations entre eux.

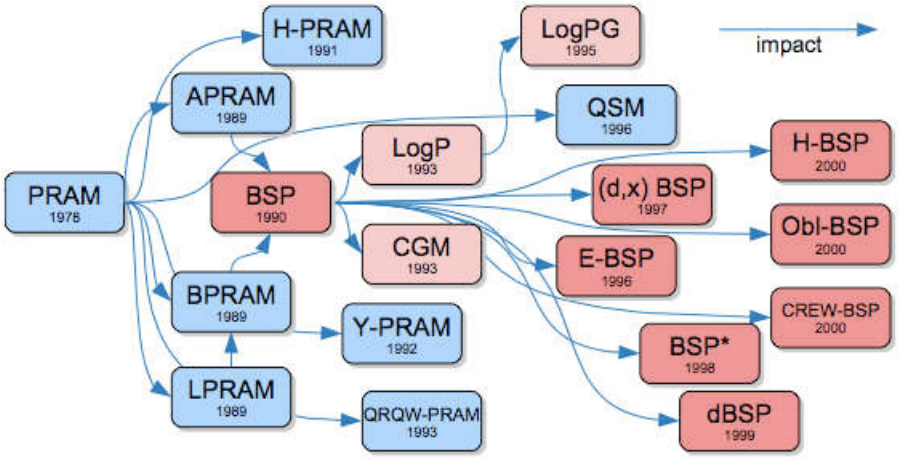


FIGURE 3.3 – Une carte de certains modèles pour le calcul parallèle

3.3.1 Le modèle Oblivious BSP

Dans le modèle BSP standard, tous les processeurs sont dans la même super-étape, garantis par la barrière de synchronisation entre les super-étapes. Mais pour une exécution correcte, il est seulement nécessaire que les messages soient reçus au bon moment. Autrement dit, deux processeurs qui ne communiquent pas les uns avec les autres n'ont pas à se synchroniser. Dans le modèle Oblivious BSP [56], les processus peuvent être dans des super-étapes différentes, la seule limitation étant que le processeur n'est pas autorisé à entrer dans une nouvelle super-étape jusqu'à la réception de tous les messages qui lui sont destinés. Comme il n'est pas possible pour le système de savoir à l'avance le nombre de messages qu'un processus va recevoir, l'utilisateur doit spécifier ce nombre pour chaque processeur. Dans ce modèle, un processus attend jusqu'à la réception de tous les messages pour commencer la nouvelle super-étape.

3.3.2 Le modèle D-BSP

Le modèle `D-BSP` (décomposable BSP) de Martin Beran [55] est une extension de BSP pour inclure la localité. Une machine `D-BSP` peut se décomposer dynamiquement en plusieurs petites partitions où chaque partition se comporte comme une machine BSP avec un nombre de processeurs (p) petit. Dans ce modèle, la communication entre deux processeurs de deux partitions différentes est impossible. Les paramètres g et L du modèle original BSP sont exprimés en fonction de p : $g(p)$ et $L(p)$. Évidemment, les valeurs de $g(p)$ et $L(p)$ augmentent avec l'augmentation de p. Cependant la majorité des implémentations utilisent le passage de messages et un arbre comme structure de communication et $L(p)$ est généralement de l'ordre $O(logp)$.

3.3.3 Le modèle (d,x)BSP

Guy E. Blelloch et al [57] ont proposé une extension du modèle BSP pour les machines parallèles hautes performances avec une mémoire globale partagée : le modèle (d,x)BSP. Les paramètres supplémentaires d (retard) et x (expansion) sont reliés à la mémoire globale. d décrit le temps pour un accès à la mémoire (typiquement de l'ordre de 10 cycles d'horloge pour les machines de 1996) et x correspond au rapport entre le nombre de processeurs et le nombre de bancs de mémoire accessibles en parallèle. La fonction d'estimation de coût est fonction du nombre maximal d'accès à un banc mémoire, avec le retard d.

Voici des exemples typiques de ces paramètres pour des machines de la période de proposition de ce modèle :

$$CrayC90(p = 16, x = 64, d = 6)$$

$$CrayJ90(p = 16, x = 64, d = 14)$$

3.3.4 Le modèle E-BSP

L'extension proposée par [58], en 1996 nommée $Extended$ BSP (E-BSP), prend en considération la localité et la proximité des nœuds de calcul par rapport au réseau de communication. Les auteurs remplacent le terme h-relation dans le modèle de coût de BSP standard par le terme (M,K1,K2)-relation, où M est le nombre total de messages, K1, K2 sont le nombre maximal de messages qu'un seul processeur envoie ou reçoit respectivement.

La localité d'une super-étape est définie comme la différence maximale des identifiants d'une paire émetteur/récepteur. La mise en œuvre devrait utiliser un système de numérotation des processeurs, tel que deux processeurs avec une petite différence dans leur mode d'identification ont une courte distance dans le réseau physique.

3.3.5 Le modèle CREW-BSP

Les architectures actuelles supportent des schémas de communication plus généraux que l'envoi et la réception du modèle standard BSP. La diffusion (broadcast) et la diffusion globale (alltoall) sont facilement réalisables sur ces architectures.

Michael T. Goodrich [59] a proposé le modèle CREW-BSP (lectures concurrentes écritures exclusives), qui est une extension du modèle original en utilisant les opérations de multi-diffusion arbitraires, c'est-à-dire, la destination d'un

message est un ensemble de processeurs. Dans le même papier, l'auteur a défini également les machines CRCW-BSP, qui sont capables de combiner les données reçues de plusieurs processeurs en utilisant une relation arbitraire d'envois et de réceptions.

On note bien que toutes ces extensions à l'exception de Oblivious BSP utilisent les mêmes modèles d'architecture et de programmation que ceux de la version originale. Oblivious BSP n'utilise pas le modèle de programmation car il n'y a pas une synchronisation globale et que deux processus peuvent être dans différentes étapes.

3.4 Implémentations de BSP

Basées sur le modèle BSP et ses extensions, il existe de nombreuse implémentations qui décrivent le programme parallèle comme une séquence d'opérations sur des données distribuées. Ces implémentations offrent une facilité d'utilisation et le non blocage est garanti. Les sections suivantes décrivent ces implémentions.

3.4.1 BSPLib

BSPlib [60] est une alternative à MPI et PVM qui est basée sur le modèle de calcul parallèle BSP. BSPlib peut être utilisée avec les langages C, C + + et Fortran. Elle utilise le style de programmation SPMD et repose sur l'efficacité des communications à sens unique. La bibliothèque de base (hors communications collectives) se compose de 20 primitives. Les primitives de communication collectives ainsi que d'autres schémas et opérations parallèles sont implémentés dans un niveau supérieur de BSPlib. BSPlib supporte l'accès distant direct à la mémoire et le passage de messages.

BSBlib est parfois appelée Oxford BSPlib qui est la boîte à outils incluant des outils de profilage et des implémentations de BSPlib pour de nombreuses machines différentes. Il existe de nombreuses publications qui concernent cette boîte à outils. Certaines décrivent son utilisation [60], d'autres répondent aux questions générales sur le modèle BSP et la mise en œuvre de l'ensemble des outils [61].

3.4.2 BSPonMPI

BSPonMPI (cf. page web du projet [62]) est une implémentation du standard BSPlib de Wijnand J. Suijlen sous la direction de Rob H. Bisseling, l'auteur du

livre sur BSP et Edupack benchmark [63] et le mainteneur de site web $Worldwide\ BSP$. BSPonMPI est semblable à la boite d'outils Oxford BSP (à une exception près) et fonctionne sur toutes les machines qui ont MPI. Cette dernière propriété est la principale caractéristique de cette bibliothèque. En effet, avec cette fonctionnalité, elle se distingue des autres bibliothèques comme la boite d'outils Oxford BSP et PUB (voir section 3.4.5) du fait que ces dernières sont optimales pour des architectures spécifiques alors que BSPonMPI est optimale pour toutes les architectures utilisant MPI.

3.4.3 BSPk

BSPk [64] est une bibliothèque BSP pour les architectures à mémoire distribuée utilisant le passage de messages. Les auteurs ont mis en œuvre cette bibliothèque en utilisant le modèle $Oblivious\ BSP$. Ils ont implémenté le concept de synchronisation $paresseuse$ (non globale), c'est à dire que les super-étapes ne sont pas séparées par des barrières globales : il est juste garanti que les messages envoyés dans la super-étape i seront reçus dans la super-étape $i + 1$. Si un processeur n'intervient pas dans une phase de communication, il n'a pas à attendre les autres processeurs à la fin d'une super-étape. Cependant, pour utiliser cette optimisation, la bibliothèque exige que l'utilisateur informe chaque processeur du nombre de messages entrant à chaque super-étape.

3.4.4 Green BSP

La bibliothèque BSP Green [65] a été conçue pour être aussi simple et portable que possible. Elle ne peut transmettre que des paquets de taille fixe (par exemple, 16 octets, configuré à la compilation). L'utilisateur doit diviser les messages en plusieurs paquets et de les recombiner à la destination. La seule fonction de réception est $bspGetPkt$ qui retourne un pointeur vers un paquet. Autrement dit, pour recevoir un message de grande taille, on doit itérer l'appel de cette fonction. Tous les paquets doivent être copiés ensemble en utilisant un grand nombre d'opérations de copie de mémoire, chacun de petite taille. Ce qui n'est ni confortable pour l'utilisateur, ni très efficace pour les messages de tailles importantes.

3.4.5 PUB

La bibliothèque Paderborn University BSP (PUB) a été proposée par [66] pour une implémentation haute performance des algorithmes BSP. Elle a été initialement développée pour les architectures parallèles monolithiques, et ensuite étendue pour les grappes de stations de travail. Bien que développée en utilisant le langage C, PUB est orientée objet, c'est-à-dire qu'elle utilise les notions d'objets (par exemple, les messages), et les méthodes pour leur utilisation. PUB est constituée de plusieurs modules pour garantir la portabilité sur de nombreuses archi-

tectures. Comme la bibliothèque BSPlib, PUB-BSP supporte les deux modes de communication : par passage de messages et l'accès direct distant à la mémoire.

3.4.6 BSML

La bibliothèque BSMLlib [67, 68] (on utilise aussi l'acronyme BSML) a été développée pour le langage Objective Caml [69] (OCaml). Elle permet la programmation data-parallèle basée sur une structure de données parallèle qui est polymorphe. Les programmes sont des fonctions (séquentielles) que l'on peut programmer en OCaml mais qui manipulent cette structure de données parallèle à l'aide d'opérations dédiées. Elle suit le modèle d'exécution BSP. La prévisibilité des performances y a été vérifiée [70] et, étant basée sur un calcul confluent, elle est déterministe. Enfin, contrairement à l'approche des langages à patrons, une sémantique parallèle explicite a été proposée [71].

L'implémentation parallèle de BSML (il existe aussi une implantation séquentielle qui donne les mêmes résultats que celle parallèle) est écrite sous la forme d'un programme SPMD (Single Program Multiple Data). BSML implémente une API utilisateur de haut niveau qui simplifie la tâche de programmeurs.

La table 3.1 expose les similitudes et les différences entre tous ces implémentations :

TABLE 3.1 – Comparaison entre diffirentes bibliothèques basées sur le modèle BSP

	Modèle	Abstraction	Langage	Architecture
$BSPlib$	BSP	Basse	C, C++, Fortran	Cluster
$BSPonMPI$	BSP	Basse	C, C++	Cluster
$BSPK$	Oblivious BSP	Basse	C,C++	Cluster
$BSPGreen$	BSP	Basse	C, C++	Cluster
BUP	BSP	Moyenne	C, C++	Cluster et Embarquée
$BSML$	BSP	Trés élévé	OCaml	Cluster

3.5 Conclusion

Dans ce chapitre, nous avons présenté diverses solutions qui visent à faciliter la programmation des architectures de calcul haute performance. Ces solutions peuvent être réparties en deux classes : structurées sur un modèle ou non.

D'une part, on a les solutions comme Charm++, sequoia et CellSs par exemple

qui offrent une facilité d'utilisation et une expressivité car elles mettent en œuvre des méthodologies de programmation basées sur les annotations de code source ou l'utilisation de classes et de méthodes spécifiques pour la programmation. Mais si on considère les critères mis en avant au début de ce chapitre, ces solutions ne permettent pas la prédiction des performance. En plus, la création de nouveaux langages et leur intégration dans le cycle de développement d'applications compromet énormément la productivité. En outre, de tels langages ont souvent une durée de vie relativement courte du faite du manque de support comme le débogueur par exemple.

D'autre part, on a les solutions qui se basent sur un modèle de calcul de haut niveau. Parmi tous ces modèles à savoir : squelettes algorithmiques, PRAM, LogP, CGM et BSP, on a vu que le modèle Bulk Syncronous Parallel (BSP) répond bien à nos critères en terme de facilité, efficacité et prédictibilité de performance. Puis nous avons présenté différentes extensions et variantes du modèle de base qui essaient d'exploiter les avantages des architectures hybrides et la localité des unité de calcul. Le modèle `D-BSP` désire diminuer le coût de la synchronisation globale du modèle de base en permettant la décomposition de la machine BSP en plusieurs sous-machines, chacune avec un petit groupe de processeurs. Le modèle E-BSP prend en considération la localité des processeurs. Le terme h-relation est exprimé en fonction de la localité des processeurs communiquant. Le support des schémas de communication collective offert par les architectures contemporaines est pris en charge par le modèle CREW-BSP.

Nous pensons qu'au sein de la communauté scientifique et utilisatrice des architectures hautes performances, l'utilisation de nouveaux langages ou compilateurs n'est pas une solution judicieuse. Une vaste sélection de code existant est disponible dans les langages comme C et C++. Ces portions de code sont réutilisables et il est inconcevable de les ignorer. En plus, les aspects orienté objets et la méta-programmation de C++ offrent une expressivité non négligeable. Ils permettent en effet de répondre aux besoins d'abstraction exigés par le développement de bibliothèques et d'outils de programmation.

Nous proposons la conception et le développement de deux outils pour faciliter la programmation des architectures CHP de type cluster de multi-cœurs et cluster avec accélérateur comme le Cell BE. `BSP++` : une bibliothèque générique basée sur la simplicité de l'interface de BSML et sur un modèle hiérarchique semblable à `D-BSP` et qui tire parti des avantages de localité des cœurs du nœud avec la mémoire partagée et l'utilisation des schémas de communication collectifs. `BSPGen` : un framework pour la génération de code adéquat pour ces architectures. Basé sur le modèle de coût de `BSP++`, il estime et génère le code correspond à l'architecture adéquate en utilisant la bibliothèque `BSP++`.

La description de ces outils fera l'objet des deux chapitres suivants.

4 BSP++ : BIBLIOTHÈQUE HAUT NIVEAU POUR LA PROGRAMMATION DES MACHINES HYBRIDES

La problématique majeure dans la conception des outils et bibliothèques haut niveau pour la programmation des architectures parallèles hiérarchiques et hétérogènes se situe dans le compromis entre $l'abstraction$ et les $performances$, ce qui explique le manque de tels outils. Afin de remédier à cette carence, nous proposons de concevoir et de développer une bibliothèque méta-programmée permettant de profiter efficacement des différents niveaux de parallélisme fournis par ces architectures tout en préservant le haut niveau d'abstraction et de simplicité : `BSP++` [72]. `BSP++` prend nativement différentes architectures de calcul haute performance comme les nœuds SMP, les grappes de multi-cœurs et les grappes de Cell BE.

Dans ce chapitre, nous évoquerons successivement le modèle de programmation de `BSP++` et son interface utilisateur. Puis nous introduirons son support pour la programmation hybride. Ensuite, nous étudierons en détail la structure de notre bibliothèque et son implémentation pour chaque architecture cible. Enfin, nous évaluerons les performances et le passage à l'échelle de `BSP++` à travers une série de tests et de benchmarks de calcul scientifique.

4.1 Le modèle de programmation

Afin d'exploiter les avantages des machines hiérarchiques et de réduire le temps des synchronisations globales des machines avec un grand nombre de processeurs, `BSP++` est basé sur un modèle hiérarchique. Comme le modèle `D-BSP`, `BSP++` peut se décomposer en différentes sous-machines hiérarchiques. Cependant, l'avantage de `BSP++` comparé à `D-BSP` est l'exploitation de la localité des cœurs à mémoire partagée en utilisant différentes valeurs des paramètres (L et g), à chaque niveau de la hiérarchie des sous-machines BSP. En effet, on a vu dans le chapitre précédent que le modèle `D-BSP` vise à réduire le temps de synchronisation globale en décomposant la machine BSP en plusieurs sous-machines, chacune avec un petit nombre de processeurs. Toutefois, toutes les sous-machines dans ce modèle ont le même paramétrage. Notre modèle, `BSP++`, comme nous allons voir dans la suite de ce document, tire avantage des caractéristiques des grappes en utilisant différentes valeurs des paramètres à chaque niveau de la hiérarchie.

En outre, comme le modèle `CREW-BSP`, `BSP++` tire profit des facilités offertes par les modules de communication de ces architectures, à savoir les communications collectives comme le `all_to_all` et le `all_Gather`.

Pour garantir une haute abstraction et une simplicité d'utilisation, nous avons choisi l'interface de BSML [67, 68] comme base pour notre bibliothèque. L'utili-

sation d'un aspect fonctionnel comme dans OCamel permet de bénéficier des caractéristiques suivantes :

4.1.1 Transparence référentielle

Un programme impératif est un processus évolutif qui, à partir d'un état initial, exécute un ensemble d'instructions jusqu'à parvenir à un état final qui "$contient$" le résultat voulu. Toutes ces opérations modifient physiquement le contenu des adresses mémoire représentant l'état courant du système. De fait, la valeur d'une expression à un instant donné peut dépendre non seulement de sa définition syntaxique mais aussi de l'état courant du programme. Ces **effets de bord** n'ont pas lieu d'être dans un programme fonctionnel. Un programme fonctionnel est par définition une fonction mathématique sans état interne qui, étant appliquée à un ensemble d'arguments, produit un ensemble de résultats.

Cette propriété dite de **transparence référentielle** est un atout considérable puisqu'elle garantit une plus grande lisibilité et fiabilité des programmes parallèles. Dans le but d'émuler un comportement fonctionnel et de bénéficier de cette propriété, l'API de `BSP++` utilise les notions des objets fonction, foncteur et lambda-fonction.

4.1.2 Fonctions d'ordre supérieur

Les **fonctions d'ordre supérieur (fos)** sont des fonctions dont les arguments ou les résultats sont eux-mêmes des fonctions. En règle générale, ces fonctions sont polymorphes. Pour les fos, la difficulté est de retrouver le type de retour d'une expression combinant plusieurs types de fonctions différents. Nous allons le voir en détail dans les sections suivantes, le type de retour des primitives de notre API sont des fonctions (fos) et leur type est complexe. Pour contourner cette complexité, la déduction de ce type est à la charge du compilateur en utilisant des notions avancées de C++ : le protocole $result\_of$.

4.1.3 Généricité

La généricité dans notre bibliothèque est assurée par l'utilisation des templates C++. Les $Templates$ sont une caractéristique du langage de programmation C++ qui permet d'implémenter des fonctions et des classes avec des types génériques. Autrement dit, cela permet à une fonction ou une classe de travailler sur plusieurs types de données différentes sans être réécrite pour chacun d'eux. Pour plus de détails sur les $Template\ C++$, voir l'annexe A.

4.2 Interface utilisateur

L'interface utilisateur de BSP++ fournit un ensemble de primitives et de concepts haut niveau implémentés sous forme générique en utilisant la notion des Template C++. Ce paragraphe va exposer les différents concepts et primitives de l'interface BSP++ : la notion de vecteur parallèle, les primitives de communication et les fonctions de contrôle.

4.2.1 Initialisation et démarrage de l'environnement BSP++

Pour plus d'abstraction, le démarrage de l'environnement BSP++ est transparent à l'utilisateur. Le démarrage est effectué automatiquement par la bibliothèque. BSP++ contient la fonction principale $main$ qui se charge de lancer l'environnement. Après le démarrage de ce dernier, BSP++ appelle la fonction bsp_main qui correspond à la fonction principale dans le code utilisateur. Autrement dit , le code utilisateur doit obligatoirement :

- avoir une et une seule fonction bsp_main qui joue le rôle de la fonction principale.
- ne pas contenir une fonction $main$ puisque celle-ci est définie directement par la bibliothèque.

4.2.2 Fonctions de contrôle

BSP++ fournit les fonctions de contrôle nécessaires pour le développement des applications parallèles :

- **`size()`** : cette fonction retourne le nombre de processeurs (processus/-threads) utilisés par l'environnement démarré dans la machine BSP.

- **`pid()`** : la valeur retournée par cette fonction correspond au `rank` du processeur dans l'environnement. Cette valeur est comprise entre 0 et `size()-1`.

- **`synchronize()`** : Cette fonction implémente la barrière du modèle BSP. Elle synchronise tous les processus au sein de l'environnement. Cette fonction est utilisée explicitement pour terminer une super étape qui ne possède pas de phase de communication.

4.2.3 Vecteur parallèle : par

Pour localiser la source potentielle de parallélisme, BSP++ fournit une interface structurée sur le modèle BSP basée sur la notion de **vecteur de données**

parallèle. Dans ce modèle, l'utilisateur stocke des données distribuées dans une classe générique spécialisée appelée **par** qui gère les styles de communication du modèle BSP.

La classe par<T> encapsule le concept de vecteur parallèle. Cette classe peut être instanciée à partir d'un grand nombre de constructions allant du tableau de style C aux conteneurs standards C++ en passant par les fonctions ou les lambda-fonctions C++. par<T> a une sémantique distribuée, ce qui veut dire qu'elle construit un élément de type T sur chaque processus de la machine BSP.

Listing 4.1 – Déclaration de par

```
1 bsp::par< double >          r; // r est un vecteur parallèle;
2                             sur chaque processus, on a
3                             une variable r de type double.
4
5 bsp::par< vector<double> > v; // v est une vecteur parallèle
6                              de type tableau de double.
```

L'accès à la valeur locale d'un vecteur parallèle est fait par l'opérateur de déréférencement (*). Chaque processus a l'accès en lecture/écriture uniquement à sa valeur locale. La lecture des valeurs distantes nécessite une communication explicite via les primitives de communication.

Listing 4.2 – Accès local au par

```
1 bsp::par< double >          r;
2 *r= 2.2;  //Modifier la valeur locale.
```

4.2.4 Primitives

Visant un haut niveau d'abstraction, de généricité et la facilité d'utilisation, notre bibliothèque implémente deux primitives de communication pour l'échange de données entres les processus. Les deux primitives **proj** et **put** sont basées sur des schémas de communication collectifs. Cette caractéristique implique que tous les processus de l'environnement sont requis lors d'appel à ces primitives.

4.2.4.1 proj

Étant donné un vecteur parallèle par en entrée, proj retourne un objet fonction qui mappe le pid d'un processus à la valeur du vecteur parallèle locale à ce processus et termine la super-étape courante en appelant la fonction synchronize. Cette primitive est utilisée pour projeter les valeurs locales de chaque processus sur tous les processus. Après l'appel à proj, tous les processus d'une machine BSP ont une copie locale des valeurs distribuées du vecteur parallèle. La figure 4.1 illustre le schéma de communication de proj.

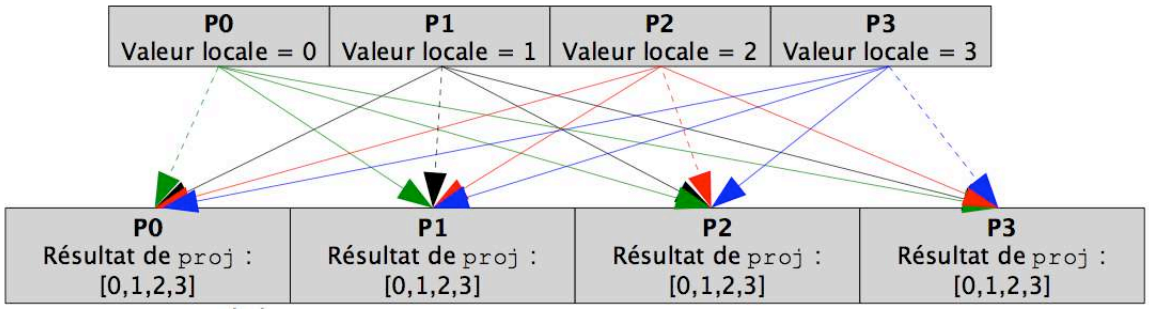


FIGURE 4.1 – Le schéma de communication de proj

Le type de retour de proj est un objet fonction polymorphe. La déduction de ce type est à la charge du compilateur en utilisant le protocole `result_of` (voir section 4.6.3.1).

Listing 4.3 – Exemple de proj

```
1 par<int> a;
2 result_of::proj<int>::type RP;
3 RP= proj(a);
4 std::cout<<"la valeur de rank 0 est : "<<RP(0)<<std::endl;
```

Dans l'exemple ci-dessus, on a projeté la valeur locale du vecteur parallèle de type `int` en appelant la primitive proj à la ligne 3. Le résultat de cette appel est un objet fonction de même type que le vecteur parallèle d'entrée. La définition de ce type ainsi que la déclaration de la variable sont illustrées à la ligne 2 de ce code.

4.2.4.2 put

Contrairement à `proj` qui permet uniquement un seul schéma de communication (la projection), `put` accepte tous les schémas de communication possibles. En d'autres termes, `put` permet à toute valeur locale à un processeur d'être transférée à tout autre processeur de la machine BSP. Comme `proj`, `put` appelle explicitement la fonction `synchronize()` pour terminer la super-étape courante.

Fondamentalement, `put` construit une matrice distribuée $\mathcal{P}$ de sorte que $\mathcal{P}_{ij}$ contient les valeurs que le processeur i envoie au processeur j. La réception consiste simplement à la lecture de la transposée de $\mathcal{P}$. Voir figure 4.8.

Le paramètre d'entrée de `put` est un vecteur parallèle `par` de type objet de fonction. Chaque processeur est titulaire d'un objet de fonction de type `T (int)`. Cet objet retourne les données de type T à envoyer au processeur i lorsqu'il est appliqué à i. Cette fonction correspond à la fonction utilisateur décrivant d'une

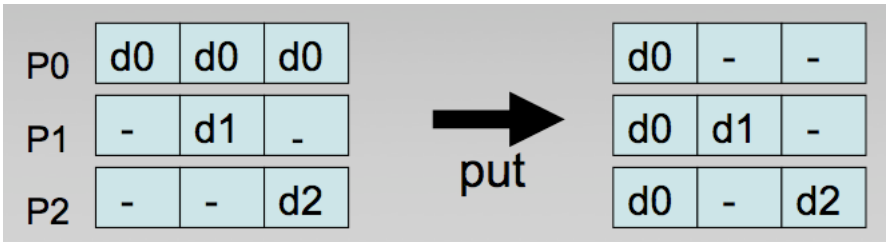


FIGURE 4.2 – Le schéma de communication de put

façon simple et abstraite son schéma de communication. Par exemple, la fonction utilisateur pour décrire le schéma de communication où le processus de rank 0 envoie sa valeur locale (tous les types fondamentaux) à tous les autres processus est illustrée dans le code 4.4.

Listing 4.4 – Accès local au par

```
1 template <class T>
2 T inline func_name (int id, par<T> const& v, const int& rank)
3 {
4  if(rank==0) return *v;
5  else return T();
6 }
```

La fonction utilisateur **`func_name`** est une fonction template qui prend les paramètres suivants :

- **id** : correspond à l'identifiant de processus récepteur. id itère sur toutes les valeurs de l'ensemble 0 à size()-1 à l'intérieur d'une boucle implémentée par la primitive **put**. (réfère à la section 4.6.3.2).

- **v** : est un vecteur parallèle au sens `BSP++` (`par`) de type template T. Ce paramètre est passé par référence (`&`). Il est constant car on ne modifie pas une valeur locale lors de sa transmission.

- rank : correspond à l'identifiant du processus. On a besoin de ce paramètre dans cet exemple, car son schéma de communication exige qu'il n' y ait que le processus de rank 0 qui envoie à tous les autres.

Cette fonction teste si le rank de processus est égale à 0. Si c'est le cas, elle retourne la valeur locale du vecteur parallèle v en utilisant l'opérateur *, dans le cas contraire, elle retourne la valeur vide d'un objet de même type que celui du retour de la fonction en utilisant le constructeur par défaut de sa classe.

En utilisant ce prototype générique, l'utilisateur peut définir tous les schémas

de communication.

Le code 4.5 montre un exemple d'utilisation de la primitive put. La ligne 2 représente la déclaration du paramètre d'entrée de la primitive `put` S, qui est un vecteur parallèle (`par`) d'objets fonction de type `<T (int)>`. Ce type indique que cette fonction prend au moins un paramètre de type int en entrée (correspond au $placeholder$ dans la fonction utilisateur) et retourne un objet de type $Template$ (dans cet exemple, le retour est de type int). Cet objet S mappe sur chaque valeur locale l'appel à la fonction utilisateur décrivant le schéma de communication avec les paramètres adéquats.

Listing 4.5 – Exemple de put

```
1 par<int> a;
2 par<boost::function<int (int)> > S= bind(func_name,_1, a, pid() );
3 result_of::put<boost::function <int (int)> >::type recv;
4 recv= put(S);
5 std::cout<< la valeur envoyée par le proc 0 est: <<(*recv)(0)<<std::endl;
```

La ligne 3 déclare le type de retour de la primitive `put`. Comme pour `proj`, `put` utilise le protocole `result_of` pour retrouver le type automatiquement à la compilation. Contrairement à `proj`, le type de retour de `put` est un `par` d'objet fonction. Donc l'accès à la valeur locale nécessite l'utilisation de l'opérateur * (ligne 5).

On peut remarquer que toute la sémantique de BSP est entièrement encapsulée au sein de ce petit nombre de composants et de fonctions, limitant ainsi l'impact de la bibliothèque sur le code utilisateur. L'interface avec les composants standard C++ et les fonctions de style C permet une intégration facile des codes existants dans les sections BSP.

En outre, l'API `BSP++` ne comprend aucune référence spécifique aux éléments de MPI, OpenMP ou CellBE. Le choix du support d'architecture se fait via un symbole de prétraitement passé au compilateur : `-DBSP_OMP_TARGET` pour OpenMP, `-DBSP_MPI_TARGET` pour MPI et `-DBSP_CELL_TARGET` pour CellBE.

En plus de ces composants, `BSP++` offre une fonction $time()$ qui retourne le temps de la machine en implémentant $get\_time\_of\_day$ pour faciliter le benchmarking des applications `BSP++`.

4.3 Support pour la STL et la programmation fonctionnelle

Les constructeurs de `par` ont été conçus avec des interfaces permettant l'interopérabilité et l'intégration de la STL et les tableaux du style C dans les codes `BSP++`. Pour avoir une interface fonctionnelle comme celle de BSML, le type de retour de `proj` et `put` a été élaboré sous forme d'objet fonction. En plus de cet aspect fonctionnel, pour plus d'interopérabilité avec des codes existants, nous avons surchargé les opérateurs d'itérations afin que ces objets aient une sémantique de tableaux pour le C-style ou $range$ pour le C++. Nous avons surchargé les fonctions membres $begin()$ et $end()$ pour renvoyer l'itérateur de début et de fin respectivement de l'objet retourné par les primitives `put` et `proj`.

Listing 4.6 – Support pour la STL

```
1 par<int> a;
2 result_of::proj<int>::type RP;
3 RP= proj(a);
4 int acc= std::accumulate(RP.begin(),RP.end(),0);
```

Dans le code 4.19, on constate la facilité d'intégration des composants du standard C++ avec `BSP++`. Effectivement, à la ligne 4, on a un appel à la fonction standard $accumulate()$ qui prend l'objet résultat de `proj` comme un range (tableau).

Le schéma de communication peut être défini par un pointeur de fonction et l'appel à `put` via l'utilisation de lambda fonctions comme illustré dans les codes 4.5 et 4.4. Aussi, il peut être présenté sous forme de foncteurs. Les foncteurs sont des fonctions à état, en général en C++, qui sont implémentées sous forme de classe avec un ou plusieurs membres privés pour stocker l'état et une surcharge de l'opérateur d'appel `: :operator()()` pour exécuter la fonction. L'annexe A fournit plus d'explications et de détails sur le fonctionnement et l'utilisation des foncteurs.

Listing 4.7 – Exemple d'un Foncteur pour put

```
1  struct sender
2  {
3   sender(par<int> const& v, int rank): val(v), rank_(rank) {}
4   int operator () (int i)
5   {
6    if(rank_==0) return *val;
7    esle return int();
8   }
9
10  private:
11  par<int> val;
12  int rank_;
13 }
```

L'exemple ci-dessus décrit le schéma de communication de l'exemple 4.4 sous forme de foncteurs. Les lignes de 4 à 8 constituent la surcharge de l'opérateur (). Le code exécuté par cet opérateur est exactement le même que celui de la fonction du code 4.4 avec une instance de type entier (int) pour le template.

L'appel à ce foncteur dans la déclaration de la primitive **put** est comme suit :

Listing 4.8 – Appel du foncteur par la déclaration du par de put

```
1 par<boost::function<int (int)> > S= Sender(a, pid());
2 result_of::put<boost::function<int (int)> >::type recv;
3 recv=put(S);
```

En plus des fonctions C-style et des objets fonctions classiques de C++, `BSP++` prend également en charge les deux principales implémentations de la fonction lambda en C++ : la bibliothèque BOOST.PHOENIX[73] et les lambda fonctions de style C++0x [74].

Ce support simplifie la définition des fonctions locales de remplissage des `par` par exemple ou pour spécifier la définition des fonctions de communication dans la déclaration de `put` pour les utilisateurs avec plus d'expertise dans le langage C++ et la bibliothèque Boost.

Par exemple, la construction d'un vecteur parallèle `par` d'entiers avec des valeurs aléatoires appartenant à l'intervalle $[-pid, pid)$ peut être faite avec une simple construction de BOOST.PHOENIX :

Listing 4.9 – Exemple d'utilisation de lambda fonction

```
1 par<int> rnd = (2.*rand()/MAX_RAND)*_1 - _1;
```

Cette notation construit un objet de fonction temporaire et effectue l'opération spécifiée. Lorsque cet objet est passé comme paramètre au constructeur de `par`, il est comparé avec le prototype de fonction approprié (ici `int (int)` et il sera appelé de manière asynchrone sur chacun des processeurs de la machine BSP, en remplaçant `_1` avec le $Rank$ réel du processus dans l'environnement.

Le même procédé peut être utilisé pour spécifier une communication avec `put`. Considérons un algorithme où chaque processeur doit envoyer le i^{me} élément de sa valeur locale d'un tableau au i^{me} processeur de la machine BSP :

Listing 4.10 – Exemple d'utilisation de lambda fonction

```
1 result_of::put< vector<double> > r = put( make_par( at(*v,_1) ) )
```

Dans ce fragment de code, `make_par` est une fonction d'assistance polymorphe qui construit une instance de `par` à partir de son argument et la fonction

`at` est une lambda-fonction qui effectue un accès aléatoire à un vecteur passé en argument. Dans cet exemple, `at` prend en entrée le tableau local (la valeur locale au vecteur parallèle v) et le $placeholder(\_1)$ qui parcourt tous les identifiant des processus de la machine BSP.

Pour tous ces exemples, l'ensemble des codes est *inliné* lors de l'appel, limitant ainsi l'surcoût introduit par cette abstraction à l'équivalent de l'utilisation d'un pointeur.

4.4 Support pour la programmation hybride

`BSP++` tire parti de l'architecture hybride des grappes en décomposant la machine BSP globale en deux niveaux hiérarchiques. Dans le niveau supérieur, une machine BSP est définie entre les nœuds du cluster avec le passage de messages comme mode de communication. Dans le niveau inférieur, une machine BSP est définie entre les cœurs d'un nœud avec la mémoire partagée, ou entre les unités de calcul de l'accélérateur. Par exemple, pour un cluster multi-cœurs, `BSP++` génère du code avec MPI au niveau supérieur et OpenMP au niveau inférieur. De même, dans un cluster d'accélérateurs de type CellBE, `BSP++` utilise MPI au niveau supérieur et le kit de développement de CellBE au niveau inférieur.

`BSP++` offre un moyen simple pour le support de code hybride. Pour développer un code hybride, l'utilisateur imbrique des super-étapes à l'intérieur d'une autre. Autrement dit, il suffit de remplacer la fonction de calcul d'une super-étape de la sous-machine BSP du niveau supérieur par une fonction contenant la super-étape compilée en mode d'un niveau inférieur. La figure 4.3 (a) montre la disposition d'une super-étape MPI. Pour permettre le calcul hybride MPI+OpenMP (Figure 4.3 (b)), la phase de calcul de la super-étape MPI est remplacée par un appel à une fonction `BSP++` OpenMP. Cette fonction contient presque le même code utilisateur que la super-étape MPI de la figure (a) compilée en mode OpenMP.

Avec cette optique d'hybridation des super-étapes, le modèle de coût de `BSP++` peut être exprimé sous la forme suivante :

$$\delta = W_{max} + \sum_{i}^{niv}(h_i * g_i(P_i) + L_i(P_i))$$

Dans cette formule, h_i, g_i et L_i représentent la taille de données transmises ou reçues en octet, le coût de la communication et le coût de la synchronisation respectivement pour chaque niveau i de la hiérarchie (MPI ou OpenMP ou CellBE). Comme les coûts de communication et de synchronisation dépendent du nombre d'éléments utilisés, ils sont exprimés en fonction de ce nombre (P_i) pour chaque niveau.

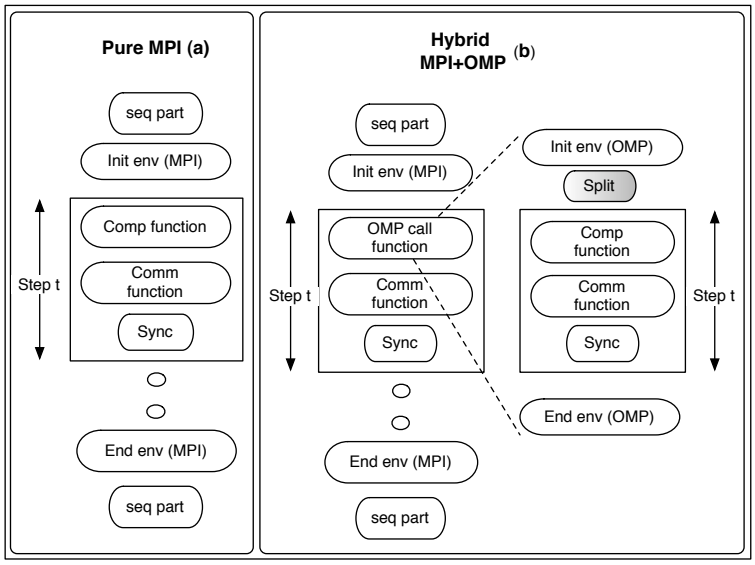


FIGURE 4.3 – Parallélisation avec le modèle hybride. *La phase de calcul de MPI est remplacée par un appel à une fonction BSP compilée avec le mode OpenMP*

Le mécanisme d'hybridation de `BSP++` est conçu pour être abstrait, simple et surtout **extensible**. En effet, cette stratégie offre une hybridation à plusieurs niveaux de hiérarchie. Par exemple, sur le cluster $Roadrunner$, on peut imaginer une hybridation à trois niveaux : 1) Communication à passage de message entre les nœuds du cluster en mode MPI. 2) L'utilisation de la mémoire partagée entre les cœurs des processeurs AMD en mode OpenMP. 3) Chaque thread OpenMP lance une exécution sur un accélérateur Cell BE où la communication entre les SPE se fait en mode Cell.

4.4.1 Hybride avec OpenMP

Pour profiter de l'efficacité de la communication entre les cœurs d'un SMP, `BSP++` utilise OpenMP pour le mode mémoire partagée. La table 4.1 illustre la comparaison des valeurs des coûts de synchronisation et de la communication pour OpenMP et de MPI sur une machine à mémoire partagée. On remarque que la synchronisation avec OpenMP est 8 fois plus rapide que MPI sachant que cette installation de MPI est configurée pour une architecture à mémoire partagée.

Puisque le modèle BSP est basé sur le style SPMD, la version OpenMP de `BSP++` utilise le paradigme OpenMP SPMD. L'utilisation de style OpenMP SPMD améliore les performances [17, 18] comparées à MPI et OpenMP grain fin, et offre aussi les facilités et les caractéristiques suivantes :

TABLE 4.1 – Variation de L (en ms) et g (en $sec\,par\,MB$) sur une machine à mémoire partagée 4x4 cœurs

	MPI					
P	4	8	16	4	8	16
g	0.087	0.22	1.69	0.025	0.069	0.68
L	4.46	20.8	108.0	2.94	8.13	13.1

– Facilite la prédiction des performances, puisque le temps d'exécution total peut être décomposé facilement en temps de calcul et en temps de communication. En plus de cette facilité, le style SPMD introduit moins d'surcoût car on utilise uniquement un ensemble limité de clauses et de primitives OpenMP.

– Diminue le faux partage ($false\ sharing$) : puisque chaque thread utilise des tableaux et des variables locaux pour le calcul et les communications (recommandé pour les architectures NUMA).

– Assure l'équilibrage des charges : chaque thread exécute le même code.

Pour les grappes de SMP à multi-cœurs, la programmation hybride MPI+OpenMP est vue comme le modèle adéquat. Pour générer un code hybride MPI+OpenMP avec `BSP++`, l'utilisateur suit la stratégie décrite par la figure 4.3. En remplaçant la partie de calcul de la super-étape MPI par une fonction qui a exactement le même code de base compilé en mode OpenMP. La différence entre les deux codes est l'existence de la fonction $split$ dans le code OpenMP.

La fonction $split$ partage la valeur locale d'un processus MPI entre les threads OpenMP de la sous-machine. Pour les valeurs scalaires, $split$ fait une copie locale de cette valeur sur chaque thread. Par contre, pour les variables de type tableau, comme le temps des copies augmente en fonction de la taille des tableaux, nous avons opté pour l'utilisation des ranges pour décomposer la variable MPI. Dans ce cas, la fonction $split$ retourne une paire de pointeurs (range) qui pointent sur les début et la fin de la portion de tableau global (MPI) attribué à ce thread. Cependant, cette stratégie peut introduire du faux partage dû aux accès à une variable partagée par tous les threads.

La portion de code 4.11 illustre un exemple d'utilisation de la fonction $split$ qui décompose un vecteur d'entiers. La ligne 1 définit le type de retour de la fonction $split$ en utilisant le protocole $result\_of$. La ligne 2 illustre l'appel à la fonction $split$. La fonction prend deux paramètres :

Listing 4.11 – Déclaration de split

```
1 typedef result_of::split<vector<int>, linear >::type slicer_type;
2 Slicer_type Omp_vect = split(MPI_vect, linear());
```

- La variable globale MPI : correspond à la variable qu'on veut partager entre les threads de deuxième niveau de la hiérarchie.

- Splitteur : le foncteur de décomposition, qui est une fonction utilisateur pour décrire le schéma de décomposition souhaité. Pour faciliter la tâche des utilisateurs, nous avons implémenté des $splitteurs$ par défaut (les plus utilisés dans le calcul scientifique).

La fonction retourne pour chaque thread un range local de même type que la variable MPI d'entrée.

4.4.2 Hybride avec Cell BE

Le modèle d'architecture de BSP considère les unités de calcul comme des paires processeur-mémoire. Or, un SPE contient une mémoire locale rapide et un processeur RISC. En ce qui concerne le réseau de communication, nous avons vu que chaque SPE est relié par l'EIB. Le système de bus présente l'avantage d'être multi-canaux (2 dans le sens horaire et 2 dans le sens anti-horaire). La phase de communication en masse transitera donc par ces anneaux. Le modèle réalise des hypothèses sur la phase de communication et en particulier on considère que les processeurs envoient et reçoivent en parallèle (tous les processeurs envoient au même moment). Cette hypothèse sur l'envoi global des données dans le réseau peut être validée sur le processeur Cell par la présence des quatre anneaux de communication entre les SPEs.

Les SPEs sont alors vus comme des paires processeur-mémoire, l'EIB est vu comme le réseau de communication. Dans la version `BSP++` sur le Cell, le PPE est considéré uniquement comme le chef d'orchestre. Il démarrera les exécutions sur les SPEs par exemple mais il n'intervient pas dans le calcul. **Uniquement les SPEs constitueront la machine BSP sur le processeur Cell** (8 SPEs sur les blades d'IBM et 6 sur les PlayStations 3).

La même stratégie d'hybridation est à suivre pour le développement de programmes pour les architecture multi-Cell BE. Une sous-machine au niveau supérieur en mode MPI pour les nœuds de cluster (entre les PPEs) et une sous-machine BSP au niveau de chaque accélérateur Cell (entre les SPEs). Cependant pour la fonction $split$, la version pour le Cell BE est différente car elle dépend de

l'architecture. En effet, pour satisfaire les restrictions architecturales à la fois matérielles et logicielles du processeur Cell BE, un mécanisme de *Split* est nécessaire même dans un code mono-Cell (pas hybride).

Les contraintes logicielles liées au SDK IBM exigent que le développement de programmes pour le Cell implique le développement de deux codes séparés :

- Un code parallèle exécuté par les SPE (*spe.cpp*). Nommé le *kernel*, il correspond au code `BSP++` avec la version Cell BE. Ce code est compilé avec un compilateur spécifique au SPE fourni par la SDK avec une version 4.1 de gcc (le `spe-gcc`), et génère un fichier objet en sortie qui ensuite sera utilisé pour construire l'exécutable final.

- Un code séquentiel exécuté par le PPE (*ppe.cpp*) contenant la fonction principale. Son rôle est la préparation des données, le démarrage des threads sur les SPEs et l'appel au code exécuté par les SPEs. Ce code est compilé avec une version spécifique au PPE le (`ppe-gcc`) et fait l'édition de liens avec une version enfouie du code objet généré par le compilateur SPE. La version enfouie est générée par le compilateur PPE en utilisant la commande `ppu32-embdedspu`. Pour plus de détails sur la compilation et la génération du code sur le processeur Cell, le lecteur pourra se référer à l'annexe B.

la figure 4.4 montre l'organisation globale d'une compilation sur le processeur CellBE.

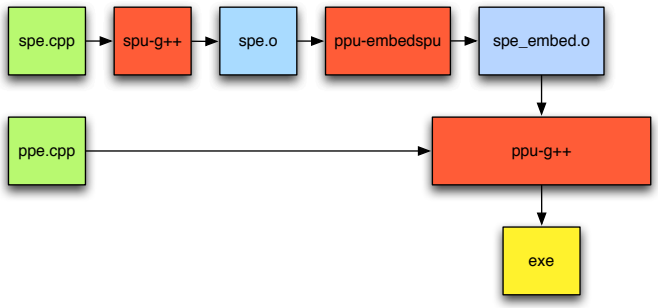


FIGURE 4.4 – organisation globale de la compilation pour le Cell BE

En outre, comme on a vu dans la section 2.4.2, les SPEs du processeur Cell BE souffrent des restrictions matérielles liées à la capacité de la mémoire locale (Local Store) qui est uniquement de 256 Ko pour charger le code et les données. Donc les tableaux de données de taille supérieure à cette limite ne peuvent pas être chargés dans cette mémoire. Pour contourner cette limitation, les données sont stockées dans la mémoire principale du PPE (RAM) et décomposées en plu-

sieurs blocs de petite taille et les SPEs itèrent sur ces blocs. A chaque itération, chaque SPE charge un bloc de données et effectue un calcul.

De ce fait, la fonction *split* dans sa version Cell BE est simplement une boucle qui englobe le code de la super-étape et itère sur les blocs des données. Cette boucle est la seule différence entre un code BSP++ pour MPI ou OpenMP et un code BSP++ pour le Cell et uniquement pour les données de grande taille. Pour simplifier cette tâche de décomposition et d'itération, BSP++ offre deux classes génériques qui encapsulent ces deux tâches. $block\_iterator < T >$ et $block\_iterator\_2D < T >$. Comme leur nom l'indique, ce sont des itérateurs pour des vecteurs de donnée à une et deux dimensions. Chacune des deux classes existe en deux versions : en entrée ($block\_input\_iterator$) effectue le transfert des données de PPE vers SPE) et inversement en sortie ($block\_output\_iterator$). La version 2D des itérateurs utilise des transferts DMA [1] en 2 dimensions qui sont plus performants qu'un accès DMA à une seule dimension de la même taille [75].

Listing 4.12 – Déclaration et utilisation des itérateurs

```
1 remote_block_input_iterator<double> it =
2 remote_block_input_iterator<double>(2,size,vector_slicer(size));
3
4 it= vect.begin(); // vect est une addresse sur le PPE.
5 printf("la valeur est %f la derniere est %f \n",(*it)[0],(*it)[size-1]);
6 it++;
```

L'exemple 4.12 met en avant la simplicité d'utilisation de ces itérateurs. La ligne 1 représente la déclaration d'un itérateur d'entrée d'un tableau à une dimension. Le schéma de décomposition est décrit par la fonction utilisateur $vector\_slicer$ passée comme argument au constructeur de l'itérateur. BSP++ implémente plusieurs fonctions de décomposition et offre la possibilité à l'utilisateur de définir son propre schéma juste en fournissant un foncteur. L'exemple 4.13 illustre l'opérateur() du foncteur $vector\_slicer$.

Listing 4.13 – Opérateur() du foncteur vector-slicer

```
1 int32_t operator() (uint32_t  iter)
2 {
3   return iter* iterSize + RankOffset;
4 }
```

Pour une exploitation maximale des performances offertes par l'architecture de Cell BE, les itérateurs implémentent la notion de $Multi - Buffering$ pour recouvrir le temps de communication par le calcul. Dans notre exemple (code 4.12), les itérateurs utilisent un double buffer (la valeur du premier paramètre dans le constructeur - ligne 2). Dans cette optique, nous avons surchargé les opérateurs $*$ et $++$. L'opérateur $*$ indique le buffer actif pour le calcul à cette itération et l'opérateur $++$ demande le chargement asynchrone du prochain bloc. Le scénario d'utilisation classique est le suivant. A la ligne 4, l'opérateur $=$ initialise

1. Direct Memory Access

l'itérateur et charge les deux premiers blocs dans les deux buffers. Pendant la première itération, l'opérateur $*$ pointe sur le premier buffer. L'appel de l'opérateur $++$ à la ligne 6 engendre deux actions :

- Demande de chargement asynchrone du prochain bloc dans l'emplacement du buffer 1. Dans ce cas, on écrase les valeurs anciennes de ce buffer. A cause de cet écrasement, nous avons implicitement différencié entre les itérateurs de lecture uniquement (entrées) et ceux d'écriture (sorties).

- Commuter l'opérateur $*$ qui pointe maintenant sur le deuxième buffer. Ce changement est périodique. À chaque appel, on change de buffer actif.

Dans un but d'efficacité et de simplicité, nous avons aussi adopté une séparation explicite entre les vecteurs locaux à la machine BSP Cell et les vecteurs distants sur le PPE. Cette séparation est effectuée par deux namespaces $local$ et $remote$.

Le transfert des données entre le PPE et SPE et entre les SPEs doit utiliser des adresses alignées sur 16 octets (128 bits) car le DMA utilise un bus de 128 bits et les données doivent être de taille multiple de 16 octets. Pour prendre en considération cette caractéristique, nous avons modifié l'allocateur mémoire de la classe $local :: vector$ pour allouer des espaces mémoire alignés.

Toutes ces propositions sont faites dans la version Cell de `BSP++` pour répondre aux exigences matérielles et logicielles du processeur. Et comme souligné précédemment, le programme sur le Cell contient deux parties séparées même pour un code Mono-Cell. La version hybride MPI+ Cell (Multi-Cell) avec `BSP++` est donc aussi simple que la version MPI+OpenMP. Il suffit d'écrire le code `BSP++` MPI sur les nœuds (PPEs) et de remplacer la fonction de calcul par l'appel au kernel qui contient la fonction $split$ (la boucle et les itérateurs).

4.5 Exemple

Dans cette section, nous allons montrer la simplicité de la programmation de `BSP++` en utilisant un benchmark simple (le produit interne ou *inner product*). Les codes suivants mettent en avant la facilité d'utilisation de `BSP++` et mettent l'accent sur la pertinence de son interface utilisateur, l'intégration avec la STL et la génération des codes hybrides. Le programme produit scalaire est considéré comme le programme de base dans le calcul scientifique. Grâce à sa simplicité, il est souvent utilisé pour décrire les outils et les bibliothèques de calcul parallèles.

L'algorithme BSP utilisé dans cette exemple fait l'hypothèse qu'à la fin de l'al-

gorithme, tous les processeurs ont besoin de la valeur du produit et que chaque processeur génère le tableau aléatoirement. L'algorithme consiste en deux super-étapes :

- **étape 1 :** Chaque processeur effectue localement un calcul de produit scalaire en utilisant ses variables locales et projette la valeur du résultat partiel sur tous les processeurs.

- **étape 2 :** Après la projection, chaque processeur calcule une accumulation de tous les résultats partiels pour avoir le résultat total de produit.

4.5.1 Mono architecture (Un seul niveau)

Le code 4.14 représente la version BSP++ de base du programme produit scalaire. Ce morceau de code démontre la facilité d'intégration avec le standard C++. En effet, on remarque l'utilisation directe et intuitive des fonctions standards avec les composants de BSP++. A la ligne 6, on a un appel à la fonction standard pour calculer localement le produit scalaire de la partie locale du vecteur sur chaque processeur. L'accès aux valeurs locales du tableau est exprimé par l'opérateur $*$, mais dans le but d'avoir une interface intuitive et compatible avec la programmation C++, nous avons surchargé aussi l'opérateur -> pour encapsuler le comportement de la conjonction de l'opérateur de déréférencement et l'opérateur d'appel de méthode (* suivi .). Le résultat partiel est stocké dans la valeur locale de chaque processeur (illustré par l'utilisation de l'opérateur *).

Listing 4.14 – le code de base BSP++ de l'exemple *inprod* avec la version pure MPI ou OpenMP ou Cell

```
1  int bsp_main(argc, argv)
2  {
3    par<vector<double> > v;
4    par<double>                  r;
5
6    *r=std::inner_product(v->begin(),v->end(), v->begin(), 0.0);
7
8    result_of:: proj<double>::type fw;
9    fw=proj(r);
10   // step 2
11   *r=std::accumulate(fw.begin(), fw.end(),0.0);
12 }
```

Notez bien que le code en lui même ne contient aucune instruction, méthode ou fonction spécifique à MPI, OpenMP ou Cell BE. Comme annoncé précédemment, c'est le même code exécuté sur ces trois architectures. Le choix de l'architecture est donné par le passage d'un symbole de préprocesseur au compilateur. Cependant ce code avec la version Cell est limité aux petites tailles de tableaux à

cause des différentes contraintes précédemment indiquées.

Afin de permettre l'exécution de la version Cell avec des grandes tailles, il suffit d'introduire une boucle et les itérateurs. Le code 4.15 correspond à la version Cell pour des données de grande taille. Les lignes 5-9 représentent la déclaration des itérateurs sur le tableau d'entrée et celui de sortie. On utilise un tableau pour retourner la valeur de sortie car les transferts des donnée entre les SPEs et le PPE exigent des tailles multiples de 16 octets. A l'exception des itérateurs et de la boucle d'itération à la ligne 14, c'est le même code que précédent.

Listing 4.15 – Le code BSP++ de l'exemple *inprod* pour la version Cell avec des tableaux de grande taille

```
1  int kernel (remote::vector<double>& in, remote::vector<double>& out)
2  {
3  int size = 16; // multiple de 16 en nombre d'octets
4  int size_res= 16/sizeof(double); //  multiple de 16 en nombre d'octets
5  remote_block_input_iterator<double> in_it =
6  remote_block_input_iterator<double>(2,size,vector_slicer(size));
7
8  remote_block_output_iterator<double> out_it =
9  remote_block_output_iterator<double>(1,size_res,vector_slicer(size_res));
10
11   par<double>                 r;
12
13   // la boucle correspondante à la fonction split
14   for(in_it=in.begin(); in_it!=in.end(); in_it++)
15     *r=std::inner_product((*in_it), ((*in_it)+size), (*in_it), 0.0);
16
17   result_of:: proj<double>::type fw;
18   fw=proj(r);
19   // step 2
20   out_it= out.begin();
21   (*out_it)[0]=std::accumulate(fw.begin(), fw.end(),0.0);
22   out_it++;
23 }
```

4.5.2 Hybride MPI+OpenMP

BSP++ offre deux moyens pour décrire une super-étape en mode hybride : le premier utilise deux compilations différentes comme décrit dans la section 4.4. On écrit une fonction OpenMP contenant la super-étape, on la compile en mode OpenMP et on fait l'édition de lien avec le code MPI. La deuxième solution consiste en une seule compilation avec le mode hybride direct en utilisant le symbole de prétraitement $-DBSP\_HYB\_TARGET$. Dans ce cas, on a un seul fichier qui contient la totalité du code. Pour spécifier la fonction (la super-étape) qui s'exécute en mode hybride (OpenMP), celle-ci est délimitée par deux jetons indiquant le début et la fin de l'hybridation : $BSP\_HYB\_START()$ et

$BSP\_END()$.

Dans le code suivant, on montre l'utilisation de la deuxième proposition.

Listing 4.16 – La version BSP++ hybride MPI+OpenMP de inprod

```
1  double inner_hyb(int argc, char** argv, vector<double>const& v)
2  {
3    double ret_value;
4    BSP_HYB_START(argc, argv)
5    {
6    typedef result_of::split<vector<double>, linear>::type slice_type;
7    slice_type v_omp=split(v,linear());
8
9    par<double>  r_omp;
10   *r_omp=  std::inner_product(v_omp->begin(),v_omp->end(),
11               v_omp->begin(), 0.0);
12    result_of:: proj<double>::type fw_omp;
13    fw_omp=proj(r_omp);
14    *r_omp=std::accumulate(fw_omp.begin(), fw_omp.end(),0.0);
15     if(pid()==0)ret_value=*r_omp ;
16   }
17   BSP_HYB_END()
18   return ret_value;
19 }
20
21 int bsp_main(argc, argv)
22 {
23   par<vector<double> > v;
24   par<double>                  r;
25
26   *r=inner_hyb(argc,argv, *v);
27
28   result_of:: proj<double>::type fw;
29   fw=proj(r);
30   // step 2
31   *r=std::accumulate(fw.begin(), fw.end(),0.0);
32 }
```

4.5.3 Hybride MPI+Cell

Pour le processeur CellBE, en raison de ses contraintes logicielles, le code exécutable final est obtenu suite à deux compilations de deux fichiers séparés. Pour cette raison, l'hybridation à partir d'une seule compilation en utilisant le symbole de pré-traitement $-DBSP\_HYB\_TARGET$ n'est pas adéquate. Les codes 4.15 et 4.17 illustrent la partie `BSP++` Cell et la partie `BSP++` MPI respectivement pour la version hybride MPI+Cell de l'algorithme produit scalaire.

Listing 4.17 – La version BSP++ hybride MPI+Cell de inprod : Partie MPI

```
1  #include "kernel.hpp"
2 int bsp_main(int argc, char** argv)
3 { par<local::vector<double> > v;
4   local::vector<double> out= local::vector<double>(16/sizeof(double));
5   par<double>                r;
6
7   PPE_Init();
8   kernel(*v,out);
9   PPE_Finalize();
10  *r= out[0]; // affecter les résultats partiaux aux variables locales
11
12  result_of:: proj<double>::type fw;
13  fw=proj(r);
14  *r=std::accumulate(fw.begin(), fw.end(),0.0);
15 }
```

4.6 Détails d'implémentation

La structure générale de la bibliothèque BSP++, le détail d'implémentation de quelque principes et fonctions ainsi que les problèmes rencontrés et leurs solutions seront exposés dans les paragraphes suivants.

4.6.1 Architecture et extensibilité de BSP++

Pour faciliter l'extension de BSP++ à d'autres architectures et cibles, la structure générale de la bibliothèque est conçue d'une manière modulaire et hiérarchique. En effet, on a un module pour chaque concept et fonction de la bibliothèque. Le module contient un fichier racine qui encapsule l'idée de base et l'âme de ce concept et un fichier spécifique à chaque architecture.

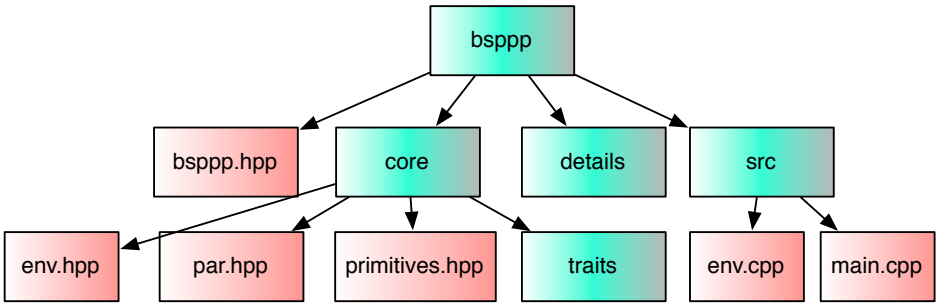


FIGURE 4.5 – Arborescence de la bibliothèque

La bibliothèque est de type « header only », ce qui signifie du point de vue de

l'utilisateur qu'il n'aura qu'à inclure des headers C++ (par exemple : $file.hpp$). Son arborescence est la suivante :

- Le dossier **bsppp** contient le header principal $bsppp.hpp$, un dossier **core** et un dossier **détails**.
- Le fichier d'entête $bsppp.hpp$ regroupe les headers nécessaires pour l'environnement BSP et les primitives.
- Dans le dossier **core**, on retrouve les fichiers définissant les primitives et l'environnement.
- Le dossier **détails** contient tous les fichiers d'implémentation en fonction des architectures.

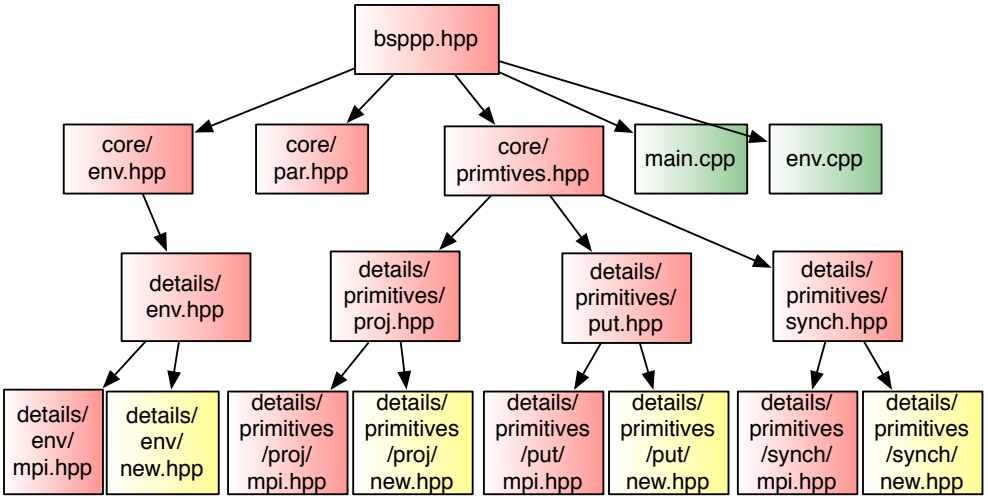


FIGURE 4.6 – Insertion du nouveaux fichiers pour une nouvelle architecture

Sur le schéma 4.6 est représenté le parcours que réalise le compilateur pour inclure les fichiers comportant les appels aux fonctions bas niveau pour une compilation MPI (utilisant le symbole de pré-traitement $-DBSP\_MPI\_TARGET$). Ces fichiers sont dans le dossier détails de l'arborescence. Grâce à l'option de compilation, le préprocesseur va directement savoir quels fichiers d'implémentation il doit inclure. Par exemple, tous les fichiers pour une implémentation MPI sont dans un **namespace MPI**. Le fichier $main.cpp$ commence par tester quelle version générer en utilisant la directive **# ifdef BSP_MPI_TARGET** afin d'utiliser le $namespace$ adéquat. Notez bien que le concept de `par` n'est pas instancié pour chaque architecture dans le dossier **détails**, car il n'est pas basé sur des

implémentations faisant appel à des notions bas niveau spécifiques à ces architectures.

Les fichiers en jaune dans la figure représentent les nouveaux fichiers à ajouter pour une nouvelle architecture cible. L'extensibilité de `BSP++` est donc très bien conçue : seulement quatre fichiers supplémentaires sont insérés dans l'arborescence, et un nouveau symbole de prétraitement à définir pour permettre le support d'une nouvelle architecture.

4.6.2 Démarrage de l'environnement et les fonctions de contrôle

Pour les grappes de multi-cœurs, comme on a vu, c'est la bibliothèque qui se charge du lancement de l'environnement via la fonction $main$. Pour MPI, celà consiste simplement à démarrer l'environnement MPI et à initialiser les valeurs du $rank$ et $size$ retournées par les fonctions de contrôle. Ces variables sont déclarées comme étant $static$ d'où l'utilisation d'un fichier $environment.cpp$ pour leur initialisation. La fonction `synchronize()` est juste un appel à la fonction `MPI_Barrier()`.

Grâce à l'utilisation du style SPMD, dans le cas d'OpenMP, le lancement de l'environnement consiste à la création d'une seule *section parallèle*. Cependant comme la variable $rank$ ne peut pas être déclarée $static$, car tous les threads auraient le même identifiant. Donc l'appel à `pid()` fait appel directement à la fonction OpenMP $omp\_get\_num\_thread()$. Malheureusement l'appel à cette fonction dégrade énormément les performances. Pour limiter cette dégradation, l'utilisation du $rank$ par la bibliothèque dans la déclaration des par, primitives ainsi que l'accès aux variables locales n'utilisent pas directement l'appel à `pid()` mais des référence à sa valeur. Comme pour MPI, `synchronize()` est juste un appel à la directive *pragma omp_barrier*.

Pour ce qui concerne la version hybride mono-compilation, l'environnement MPI est lancé classiquement comme dans la version pure MPI et l'environnement OpenMP est lancé par l'appel à la macro BSP_HYB_START. Cette dernière démarre la section parallèle OpenMP et assigne l'utilisation du **namespace** OMP pour toutes les primitives et les fonctions `BSP++` comme présenté ci-dessus. La terminaison de la section OpenMP est définie par la macro BSP_HYB_END

Listing 4.18 – macro BSP_HYB_START

```
1 #define BSP_HYB_START(argc,argv)      \
2 OMP::bsp::details::environment BOOST_PP_CAT(env_,_COUNTER_)(argc, argv);\
3 if (bool stop_ = false){} else      \
4 for(; !stop_; stop_=true)           \
5 BSP_STARTUP()
```

Les lignes 2-4 du code 4.18 correspondent à une macro de prétraitement de BOOST pour générer différentes versions de la macro $BSP\_HYB\_START()$, chacune avec un nombre différent d'arguments. La macro $BSP\_STARTUP()$ s'occupe du lancement de l'environnement et l'attribution du namespace :

Listing 4.19 – macro BSP_HYB_START

```
1 #define BSP_STARTUP()   BSP_PRAGMA ( omp parallel )     \
2 {                    \
3 using OMP::bsp::pid;        \
4 using OMP::bsp:: put;      \
5 /*
6  toutes les fonctions et primitives ...
7 */
```

La version Cell de BSP++ se base sur une implémentation de MPI pour le Cell. Cette version (**CBE_MPI**), développée en interne au sein du LRI avec Sébastien SCHAETZ, implémente les fonctions de base de MPI pour la communication entre les SPEs en utilisant les primitives bas niveau du SDK IBM et en fournissant une API semblable à celle de MPI : *MPI_send* et *MPI_recv* par exemple.

Le démarrage de l'environnement BSP++ pour le Cell est effectué manuellement avant l'appel au $kernel$ dans le code PPE. En effet, les SPEs ne possèdent pas de système d'exploitation, le démarrage des SPEs est effectué par le chef d'orchestre (PPE). Le PPE crée un thread sur chaque SPE et transmet le code du kernel (le code BSP++) vers la LS de chaque SPE. De plus, l'utilisateur doit indiquer le nom du fichier objet des SPE dans l'appel du kernel.

Pour encapsuler toutes ces étapes, BSP++ implémente deux macros : `PPE_Init()` et `PPE_Finalize()`. Ainsi l'appel au $Kernel$ est emboîté par ces deux macros comme illustré par le code 4.20. La seule contrainte exigée par BSP++ est que les deux codes SPE (Kernel) et PPE incluent un fichier d'entête contenant le prototype du kernel. Cette contrainte est due à l'utilisation d'un code enfoui pour les SPEs.

Listing 4.20 – Lancement du kernel

```
1 # include "kernel.hpp"
2
3 PPE_Init()
4 Kernel (argc, argv, ...);
5 PPE_Finalise();
```

En outre, deux stratégies d'appel au kernel sont implémentées : l'appel *synchrone* où le PPE est bloqué jusqu'à la fin du kernel pour continuer son exécution

et un appel *asynchrone* où le PPE lance le kernel et reprend la main juste après. Cette différence est exprimée par l'ajout du mot clé $Asyn\_$ lors de l'appel au kernel (ligne 4 code 4.21).

Listing 4.21 – Appel Asynchrone du Kernel

```
1 # include "kernel.hpp"
2
3 PPE_Init()
4 Async_Kernel (argc, argv, ...);
5 // faire aute chose en Asynchrone avec les SPEs
6 PPE_Sync(); // attendre la fin du Kernel
7 PPE_Finalise();
```

4.6.3 Primitives

4.6.3.1 Proj

La classe principale de `proj` est une classe générique (Template), constructible et copiable : elle implémente un constructeur et un opérateur d'affectation (opérateur =) qui appellent la fonction $proj\_impl$ qui est spécifique à chaque architecture cible. (code 4.22).

Listing 4.22 – Classe principale de proj

```
1  template <class T>
2  struct proj
3  {
4  template<class P> proj( P const& p): data_(size())
5   {
6    proj_impl(data_,pid(),*p.get());
7   }
8  template<class P> proj& operator=(P const& p)
9   {
10   proj_impl(data_,pid(),*p.get());
11   return *this;
12  }
13
14  private:
15  buffer<T> data_;
16 };
```

Nous avons précédemment introduit le type de retour de `proj` comme étant un objet fonction méta-programmé. En effet pour avoir ce type de retour, nous avons créé un **namespace result_of** qui hérite du protocole de base **result_of** de Boost comme suit :

Listing 4.23 – Protocole result-of

```
1 namespace result_of
2 {
3 template<class T> struct proj { typedef proj<T> type;}
4 }
5 template <class T> struct result_of_proj : result_of::proj<T>{};
```

Le protocole $result\_of$ permet de déterminer le type d'une expression d'appel. Étant donnée une valeur f de type F et n valeurs $t1, t2, ..., tn$ de types $T1, T2, ..., TN$, respectivement, le type $result\_of < F(T1, T2, ..., TN) >:: type$ définit le type de résultat de l'expression $f(t1, t2, ..., tn)$. Cette mise en œuvre permet la généricité de type F. Celui-ci peut être de type pointeur de fonction, référence de fonction, pointeur de fonction membre ou un type de classe comme vu précédemment. Pour plus de détails sur le protocole $result\_of$, voir l'annexe A.

Afin d'émuler un comportement fonctionnel de `BSP++`, nous avons surchargé l'opérateur d'appel (`operator()`) ainsi que les itérateurs.

Listing 4.24 – Surcharge du l'opérateur () et des itérateurs

```
1 typedef T const& result_type;
2 result_type operator()(int i) const {return data_[i];}
3
4 iterator begin() {return data_.begin();}
5 iterator end()    {return data_.end();}
```

Ces opérateurs utilisent la variable $data\_$ qui est le buffer utilisé par la fonction `proj_impl`. L'allocation de l'espace mémoire du $data\_$ ainsi que l'implémentation de la fonction `proj_impl` dépendent de l'architecture cible.

L'allocation de l'espace mémoire du buffer dans sa version MPI est réalisée par le constructeur de `proj` qui alloue `size` (nombre de processeurs dans la machine BSP) éléments de type T sur chaque processeur. Ensuite la fonction `proj_impl` remplit le buffer dans chaque processeur avec la valeur locale du vecteur parallèle et effectue un échange global via la fonction `all_gather`. (code 4.25).

Listing 4.25 – Version MPI de l'implémentation de proj

```
1 namespace MPI
2 {
3  template<class T>
4  inline void proj_impl(buffer<T>& data, int i, T const & v)
5   {
6    std::fill(&data[0], &data[0]+size(), v);
7    boost::mpi::all_gather(runtime::world, & data[pid()], 1, &data[0]);
8   }
9 }
```

On remarque que le même buffer est utilisé pour l'émission et la réception afin d'éviter de faire des copies. Celà est possible car l'implémentation MPI de `all_gather`

se charge de l'utilisation des temporaires en interne.

La version Cell est similaire à cette dernière. En effet, une fonction semblable à `all_gather` a été implémentée en utilisant les fonctions `Isend` et `Irecv` de CBE-MPI qui elles-mêmes implémentent des transferts DMA en utilisant le SDK IBM. La seule différence est que dans cette version, nous avons utilisé une allocation des buffers alignée pour satisfaire les contraintes des transferts DMA entre les SPEs (voir section 4.6.4.1).

Concernant la version OpenMP, afin de profiter de l'avantage de la mémoire partagée, notre stratégie consiste en une **seule** allocation de $size$ éléments effectuée par un seul thread et les autres récupèrent le pointeur vers cet espace. Le code suivant montre cette allocation.

Avec cette stratégie, la fonction `proj_impl` est simplement une copie asynchrone de la valeur locale de chaque processeur dans la case adéquate du buffer.

Listing 4.26 – Allocation du buffer en mode OpenMP

```
1  namespace OMP
2  {
3   template<class T>
4   inline void alloc (T& data, int size)
5    {
6    static T* temp;
7    #pragma omp single
8    temp = new T[size];
9
10   data=temp;
11   }
12 }
```

4.6.3.2 Put

Comme déjà mentionné auparavant, `put` implémente une interface fonctionnelle grâce à la surcharge des opérateurs et à l'utilisation du protocole `result_of` dans la classe mère `put`. Cependant le schéma de communication de `put` est défini par la fonction utilisateur de type $T(int)$. Cette fonction est utilisée pour remplir un buffer. Dans le code de la fonction **`comm_fun`** (listing 4.4), le premier paramètre est un $placeholder$ (une variable utilisée par le cœur de `put` qui peut être optionnelle pour l'utilisateur). En effet cette variable représente le rank du processeur destinataire. Pour celà, cette variable parcourt tous les $ranks$ de la machine BSP. A chaque itération, le résultat de retour de la fonction utilisateur est stocké dans une case du buffer.

Le listing 4.27 présente le code de la fonction `put_impl` avec sa version OpenMP. Dans cette version, le buffer $data$ est une matrice de taille ($size \times size$) stockée dans la mémoire partagée. Nous avons utilisé la même stratégie d'allocation que celle de `proj`. L'utilisation des variables pid et $size$ a pour but d'éviter l'appel aux fonctions $pid()$ et $size()$ qui détériorent les performance comme expliqué auparavant.

Listing 4.27 – La version OpenMP de l'implémentation de put

```
1  namespace OMP
2  {
3   template<class T>
4   static inline void put_impl (buffer<T>& data, boost::function<T(int)>
5                const& f, int const& pid, int const& size)
6    {
7     int offset=pid*size;
8     for(int i=0;i<size;i++) data[i+offset]=f(i);
9     synchronize();
10   }
11 }
```

Dans sa version MPI, le buffer $data$ est un tableau à une dimension existant sur chaque processeur. Après le remplissage de ce buffer, on appelle la fonction MPI `all_to` pour faire l'échange de données.

Basée sur une implémentation d'une fonction `all_to_all` pour le Cell, la version Cell de `put_impl` est semblable à celle de MPI. Cependant, à cause des contraintes mémoire et de performances, la version utilisée diffère sur deux points :

- Elle utilise deux buffers : un pour l'émission et l'autre pour la réception car les buffers d'émission et de réception n'ont pas la même taille sur chaque processeur.
- A cause de la même contrainte que le point précédent, `put_impl` utilise deux appels à `all_to_all` : le premier échange la taille des données. Après l'allocation d'espace pour la réception (deuxième buffer), en utilisant ces tailles, le deuxième `all_to_all` est utilisé pour la transmission des données.

Le listing 4.28 montre le code de `put_impl` dans son implémentation pour le Cell BE pour les vecteurs.

Notez bien que l'utilisation d'un schéma de communication basé sur la collectivité d'un `all_to_all` ne signifie en aucun cas que tous les processeurs transmettent la même valeur à tout le monde. En effet, lorsque le schéma de communication de l'utilisateur indique que le processeur i ne transmet rien au processeur

j, cela est exprimé par le retour d'une valeur par défaut de la fonction de communication. Cette valeur est affectée à une case dans le buffer. Dans la version OpenMP, cela correspond simplement à une écriture d'une valeur dans la mémoire principale. Avec les versions à passage de messages, le processeur source envoie un message contenant une seule valeur pour indiquer au processeur destination qu'il n'y a rien à transmettre. Avec ce système, `put` simule efficacement le comportement des schémas point à point aussi bien que les schémas collectifs.

Listing 4.28 – La version Cell de l'implémentation de put

```
1  namespace CELL
2  {
3   template<class T>
4   static inline void put_impl (buffer<local::vector<T> >& data,
5                 boost::function<local::vector<T> (int)> const& f,
6                 int const& pid, int const& size)
7    {
8     int size_send[size];
9     int size_recv[size];
10
11    buffer<local::vector<T> > data_send(size);
12
13    for(int i=0;i<size;i++)
14     {
15      send_data[i]=f(i);
16      size_send[i]= (send_data[i]).size();
17     }
18     // 1er appel - échange des tailles.
19     cbe_mpi::all_to_all(*runtime..world, size_send, 1, size_recv);
20
21     // allocation d'espace memoire
22        for(int i=0;i<size;i++)
23           data[i]=local::vector<T>(size_recv[i]);
24
25    // 2 ème appel - echange des données.
26    cbe_mpi::all_to_all(*runtime..world, send_data, 1, data);
27    data[pid]=f(pid);
28    synchronize();
29   }
30 }
```

4.6.4 Spécificité de la version Cell BE

Comme précédemment indiqué, pour prendre en considération les spécificités matérielles et logicielles du processeur Cell BE, nous avons proposé une solution pour chacune d'elles. Cette section détaille l'implémentation de ces dernières.

4.6.4.1 Alignement des données

Le transfert des données entre les différents éléments du processeur Cell (PPE et SPEs) via l'utilisation des DMAs exige en terme de performance que celles-ci soient alignées [10]. Pour cela, dans les namespaces $local$ et $remote$, nous avons surchargé l'allocateur de la classe $vector$ ainsi que dans les buffers utilisés par les primitives de communication. Dans toutes ces classes, l'allocation est effectuée par l'instruction `_malloc_align` et la libération avec `_free_align`.

Cette surcharge nous a obligé à changer le code des collectives `all_to_all` et `all_gather` du Cell. La collective `all_to_all`, par exemple, prend deux pointeurs dans son implémentation MPI. Elle itère sur ce pointeur pour accéder aux données du buffer, comme sur les schémas ci-dessous.

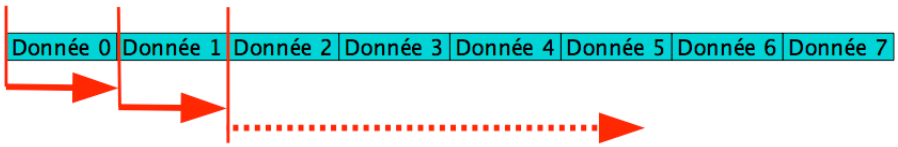


FIGURE 4.7 – Transfert des données non alignées dans la mémoire

Lorsque l'on itère sur un pointeur, les données sont supposées contigües. Mais lorsque l'on aligne les données, elles se retrouvent dans la configuration représentée sur le schéma ci-dessous. Avec des adresses alignées, les données ne sont plus contigües.

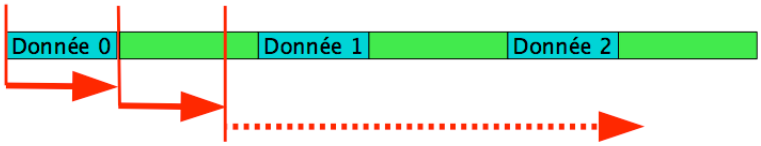


FIGURE 4.8 – Transfert des données alignées dans la mémoire

Pour résoudre ce problème, nous avons proposé une nouvelle interface pour les collectives. Cette dernière prendra des itérateurs et non des pointeurs. Il faut donc construire un itérateur sur notre classe $buffer$ qui saura comment itérer sur les données. L'implantation de la classe $buffer$ de données alignées est un objet $template$. La surcharge des opérateurs nous permet un accès rapide aux données alignées de la façon suivante : $buffer[i]$

Listing 4.29 – Surcharge d'opérateur d'accès dans la classe buffer : version Cell

```
1 reference   operator [](size_t i)    {return *aligned_at(i);}
2 const_reference operator [](size_t i) const {return *aligned_at(i);}
```

Listing 4.30 – Nouvelle interface de all_to_all dans la version Cell de BSP++

```
1 template<typename RandomAccessIterator>
2 void all_to_all (communicator& comm, const RandomAccessIterator in_values,
3                          int n, RandomAccessIterator out_values );
```

Grâce à cette nouvelle interface, nous pouvons envoyer et recevoir des données qui sont alignées et de plus, l'interface des collectives devient plus générique. En effet, $RandomAccessIterator$ est un itérateur qui fournit à la fois l'incrémentation ou la décrémentation, et qui fournit également des méthodes en temps constant pour aller de l'avant et en arrière dans des étapes arbitraires. $RandomAccessIterator$ fournit la quasi-totalité des opérations de l'arithmétique des pointeurs ordinaires du langage C

4.6.4.2 Itérateurs et multi-buffering

L'architecture de Cell BE offre un moyen très efficace pour masquer le temps de transfert des données entre les différentes mémoires (RAM et LSs). Ce moyen se base sur l'**Asynchronisme** de ces derniers. En effet, sur chaque SPE, on a un contrôleur de mémoire qui a le rôle d'un coprocesseur pour le SPU (l'unité de calcul). Le contrôleur de mémoire se charge des requêtes des transferts en asynchrone avec le SPU. Pour une exploitation maximale des performances du Cell, nous avons utilisé ce mécanisme pour recouvrir le temps de calcul effectué par les SPU et le temps de communication géré par les contrôleurs MFC. Nous avons implémenté la notion de multibuffering déjà introduite dans la section 4.4.2. Les listings 4.31 et 4.32 représentent le code de la surcharge des opérateurs * et ++ respectivement.

Listing 4.31 – Surcharge d'opérateur * pour les itérateurs : version Cell

```
1 inline T* operator * () const
2 {
3 dma_synchronize_c(tags[current]);
4 return buffers[current].get();
5 }
```

Le code 4.31 illustre bien l'objectif de cet opérateur : pointer sur le buffer courant. Pour cela, on commence par synchroniser le chargement DMA asynchrone effectué par l'opérateur++ (voir le code 4.32) en utilisant le tag correspondant au buffers courant.

Listing 4.32 – Surcharge d'opérateur ++ pour les itérateurs -version Cell-

```
1 inline void operator ++ ()
2 {
3  int32_t addr_offset= addr_calc(n);
4  spe_ppe_get_asyn_c(buffers[n%depth].get(), base_addr+
5            (addr_offset*sizeof(T)), size, tags[n%depth]);
6   n++;
7   current=(current+1)%depth;
8 }
```

Dans le code 4.32, la fonction $addr\_calc(n)$ est un pointeur de fonction qui pointe sur la fonction utilisateur ($slicer$) déclarée dans le constructeur de l'itérateur (voir le code 4.12). Cette fonction retourne l'adresse du début de chargement. La variable $depth$ indique le nombre de buffers utilisés (correspond au 1er paramètre dans la déclaration d'un itérateur). Les variables n et $current$ sont un compteur séquentiel et une valeur indiquant le buffer courant respectivement. Le code commence par calculer l'adresse de chargement, ensuite il lance la requête DMA en utilisant la fonction SDK d'IBM à la ligne 4 et se termine en positionnant la valeur du buffer courant pour l'opérateur *.

4.7 Absence de support direct pour les GPU

Concernant les GPU, nous avons pensé a étendre le support de `BSP++` pour les GPUs en définissant une machine BSP entre les blocs et chacune démarre un nombre fixe de threads. A première vue, le modèle BSP est intuitivement adaptable aux cartes graphiques. En effet, l'aspect hiérarchique de CUDA sous la notion de bloc et de threads par bloc et l'architecture many-core de la carte rendent cette intuition encore plus pertinente. Cependant, cette intuition s'est rapidement avéré incompatible avec le modèle `BSP++` à cause de plusieurs incohérences à la fois de programmation et de conception. Parmi ces causes, on cite :

- L'API CUDA ainsi que le matériel des GPUs n'offrent actuellement aucun moyen de faire une synchronisation globale entre tous les threads d'un $kernel$. Autrement dit, il n'y a pas un moyen à part la terminaison d'un $kernel$ de faire une synchronisation entre les blocs de ce dernier, alors que la synchronisation est l'un des piliers du modèle BSP.

- Si on effectue la synchronisation globale par la terminaison du $kernel$, cela implique que chaque super-étape correspond à un seul kernel. Et dans ce cas, il n'y aura pas un intérêt à faire la communication entre les blocs à la fin du $kernel$ car la prochaine super-étape lance un nouveau $kernel$. Autrement dit, il n'aura pas une sémantique pour l'utilisation des primitives de communication `put` et `proj`.

– L'idée de base et l'âme de BSP++ est l'utilisation d'un même code utilisateur pour générer différentes versions en fonction de l'architecture cible. Le code utilisateur peut être constitué de fonction séquentielles compilées (des boîtes noires), alors que le modèle de programmation sur les GPUs exige un développement spécifique du code différent de celui des CPUs.

Néanmoins, BSP++ offre un moyen pour faciliter et augmenter l'accessibilité de la programmation **multi-GPU**, en se basant sur sa caractéristique soulignée dans ce troisième point. A savoir, l'acceptation des fonctions boîtes noires dans la génération de code parallèle. En effet, il suffit de remplacer la fonction du calcul d'un code BSP++ (MPI, OpenMP ou hybride) par l'appel à un kernel considéré comme une boîte noire compilée pour les GPU. De cette façon on génère des codes hybride (MPI+OpenMP) accélérés par les GPUs.

4.8 Evaluation

Pour évaluer les performances et le passage à l'échelle de notre bibliothèque, nous avons élaboré une série de tests et d'expériences.

4.8.1 Plateformes et protocoles

Nos expériences ont été effectués sur trois plateformes différentes :

– **AMD-Machine** : est une machine quadri-processeurs quad-cœurs AMD Opteron 8354 à mémoire partagée de 16 Go de RAM et 3 Mo de cache L3. Le noyau linux 2.6.26 et OpenMP2.0 avec la version 4.4 du compilateur *gcc* sont utilisés. Nous avons utilisé la bibliothèque OpenMPI sous la version 1.4.2.

– **Cluster-Machine** : c'est un cluster de 32 nœuds du site de Bordeaux de la platforme grid5000 [76] (Une grille de calcul nationale distribuée sur 9 sites sur le territoire français). Chaque nœud est un bi-processeur bi-cœurs de 2.6 Ghz avec 4 Go de mémoire et 2 Mo de cache L2. gcc-4.4 et une version 1.4.3 de OpenMPI sont utilisés.

– **CellBE-Machine** : est un petit cluster de deux Blades IBM center : un QS20 (avec un seul PowerXCell 8i de 3.2 Ghz disponible dans sa version 5.1) et un QS22 (avec deux PowerXCell 8i de 3.2 dans leur version 4.8). Chaque nœud contient 2 Go de RAM et utilise le noyaux Linux 2.6.18, le compilateur gcc-4.1 et OpenMPI 1.2.5. Les deux blades sont interconnectés par un réseau Ethernet à 100 Mb/s.

4.8.2 Benchmarks

Différents benchmarks ont été utilisés dans nos tests :

- Nous avons comparé les performances de BSP++ avec celles d'EDUPACK [63]. EDUPACK est une suite de benchmarks et tests écrits en C et BSPLib [60]. Cette comparaison nous permet d'évaluer l'surcoût de C++. Les tests utilisés sont le produit scalaire interne $inprod$, la transformée de Fourier FFT et la décomposition LU.

- Nous avons comparé les performances et le passage à l'échelle de BSP++ fonctionnant sur une machine SMP à mémoire partagée utilisant MPI et OpenMP pour évaluer le passage à l'échelle de nos implémentations et évaluer le gain offert par l'utilisation de OpenMP sur les architectures à mémoire partagée. Les algorithmes choisis sont : le $inprod$, la multiplication matrice-vecteur GMV et la multiplication matrice-matrice GMM.

- Nous avons comparé le passage à l'échelle et l'efficacité des versions MPI et hybrides sur les grappes. Les mêmes tests que dans le cas précédent ont été utilisés.

4.8.3 BSP++ et EDUPACK

Les benchmarks utilisés sont le produit scalaire de $6,4 \times 10^7$ élément, une FFT de 8388608 éléments une décomposition LU de 4096×4096 éléments. Globalement, les versions C d' EDUPACK et celles de BSP++ passent à l'échelle d'une façon très similaire. Pour le produit scalaire, BSP++ est toujours mieux qu' EDUPACK, ce qui signifie qu'il n' y a aucune surcharge à l'utilisation de C++. Pour la FFT et LU, BSP++ introduit un petit surcoût de 7 % à cause de l'utilisation des variables supplémentaires pour adapter la structure de l'algorithme original dans lequel les données ont été transférées sur place dans leurs blocs de mémoire respectifs. Pour LU, BSP++ a de meilleures performances. En effet, sur 16 cœurs, BSP++ a un gain de 7 %. Il est important de noter que la mise en œuvre d'un modèle générique, basé sur les *Template* C++ ne dégrade pas les performances car le code généré est en général entièrement *inliné*, y compris les appels aux lambda fonctions .

4.8.4 MPI vs OpenMP

Dans le but de mettre en avant l'avantage d'utilisation de la mémoire partagée comme moyen de communication entre les cœurs d'un nœud sous la paradigme OpenMP, nous avons comparé les performances des versions MPI et OpenMP de quatre benchmarks différents. Notez bien que la version de MPI utilisée est la

TABLE 4.2 – Temps d'exécution des versions BSP++ et BSPlib pour les benchmarks EDUPACK.

	BSP++ avec OpenMP			BSPlib avec mémoire partagée		
	P=4	P=8	P=16	P=4	P=8	P=16
InProduct	0.098	0.059	0.042	0.11	0.063	0.046
FFT	2.07	1.20	1.10	2.10	1.11	0.87
LU	48.1	31,2	22.1	44.67	30.12	23.82

version mémoire partagée d' $OpenMPI$.

Le benchmark $inprod$ (figure 4.9) a presque les mêmes performances pour les deux versions MPI et OpenMP en terme de temps de calcul et de communication car le pattern de communication de celui-ci est un simple `proj` d'un seul élément. Cependant sur 16 cœurs, OpenMP a un petit avantage dû au temps de synchronisation. Comme $inprod$, le benchmark GMV expose les mêmes caractéristiques. A partir de la figure 4.10, on peut voir que ces deux benchmarks passent très bien à l'échelle avec les deux versions MPI et OpenMP.

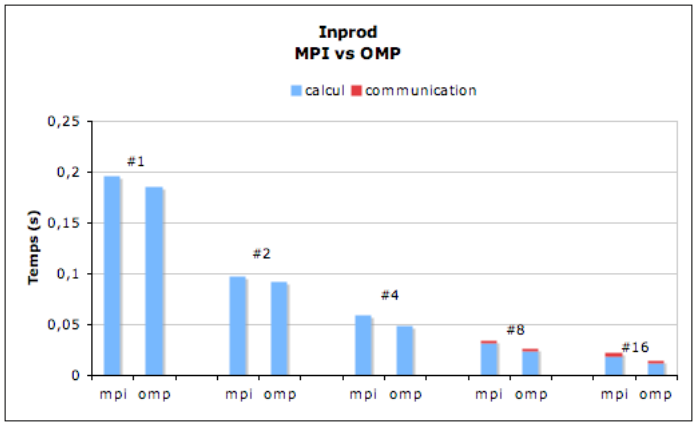


FIGURE 4.9 – Temps d'exécution (MPI et OpenMP) du benchmark Inprod sur la AMD-Machine.

Le benchmark GMM exige que le nombre de processeurs utilisés soit un carré d'entier dû à l'utilisation d'un algorithme basé sur une grille de processeurs en ligne et en colonne. Les deux versions passent très bien à l'échelle (voir figure 4.11). Toutefois, OpenMP exhibe de meilleures performances en terme de communication. En effet, sur 16 cœurs, le temps de communication représente 47% du temps total d'exécution pour MPI et le gain de OpenMP en temps de communication est entre 8% et 40% sur 1 et 16 cœurs respectivement.

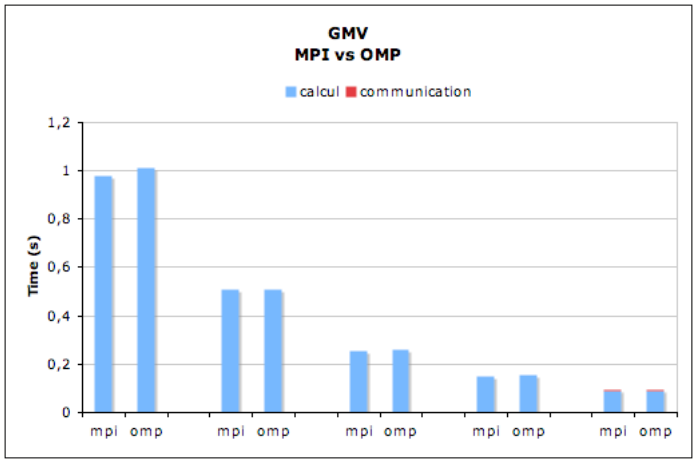


FIGURE 4.10 – Temps d'exécution (MPI et OpenMP) du benchmark GMV sur la AMD-Machine.

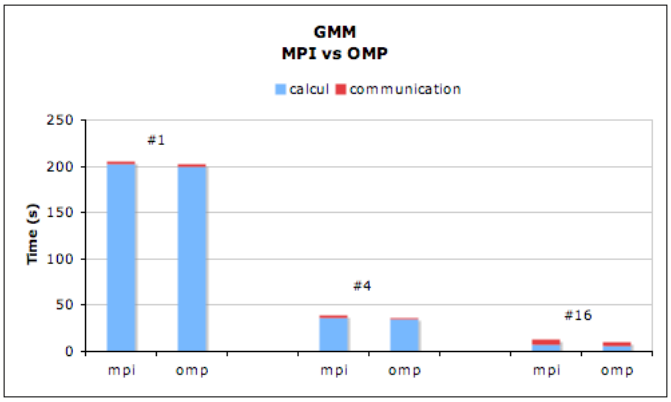


FIGURE 4.11 – Temps d'exécution (MPI et OpenMP) du benchmark GMM sur la AMD-Machine.

4.8.5 MPI vs Hybride / MPI+Cell

4.8.5.1 Hybride MPI+OpenMP

En utilisant le **Cluster-Machine**, nous avons testé l'efficacité et le passage à l'échelle des versions hybrides de ces benchmarks. Les résultats sont présentés sous forme d'une comparaison entre les temps d'exécution des deux versions pure MPI et hybride MPI+OpenMP. Comme les nœuds de ce cluster contiennent 4 cœurs communiquant par une mémoire partagée, nos versions hybrides utilisent 4 threads OpenMP par processus MPI lancé sur chaque nœud.

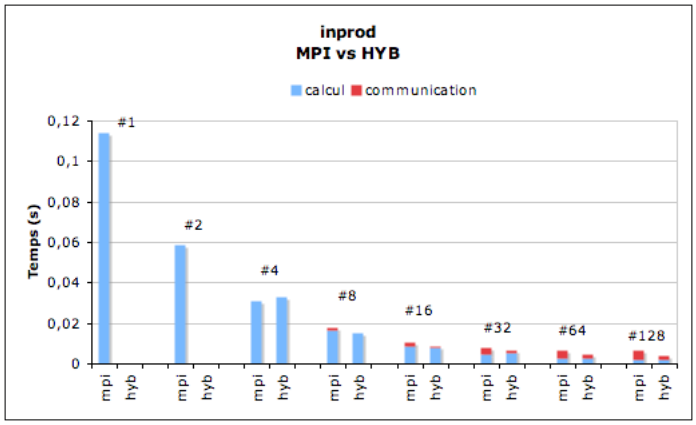


FIGURE 4.12 – Temps d'exécution (MPI et hybride) du benchmark Inprod sur le Cluster-Machine.

Comme pour la comparaison entre MPI et OpenMP, la version hybride du benchmark inprod a le même comportement que la version pure MPI jusqu'à 16 cœurs. Ceci est une conséquence de la simplicité du pattern de communication. Au delà de 32 cœurs, la version hybride devient plus performante en terme de communication due à la synchronisation qui pénalise la version MPI. Cette avantage est de plus de 50% sur 128 cœurs.

Les deux versions du benchmark GMV ont une accélération linéaire jusqu'à 32 cœurs avec un avantage en performances pour la version hybride. Pour 64 et 128 cœurs, la version hybride a un gain en temps de communication de 27% et 20% respectivement.

La version hybride du benchmark GMM (figure 4.14) a une accélération linéaire jusqu'a 100 cœurs, avec des gains allant de 24% à 53% de temps de communication de 4 à 100 cœurs.

4.8.5.2 Hybride MPI+Cell

Pour les versions hybrides sur le cluster de Cell BE, la table 4.3 ci-dessus résume les résultats des différents tests. Notez bien que le code séquentiel de chaque benchmark est exactement le même utilisé dans les tests sur les multicœurs. Autrement dit, ces versions ne contiennent aucune optimisation pour le Cell BE tels que l'utilisation des instructions vectorielles. De ce fait, on constate que les performances séquentielles sur un seul Cell sont médiocres comparées à celles des multicœurs. Ceci dit, concernant l'hybridation sur le cluster, pour les trois benchmarks, on remarque une accélération linéaire confirmant de la sorte

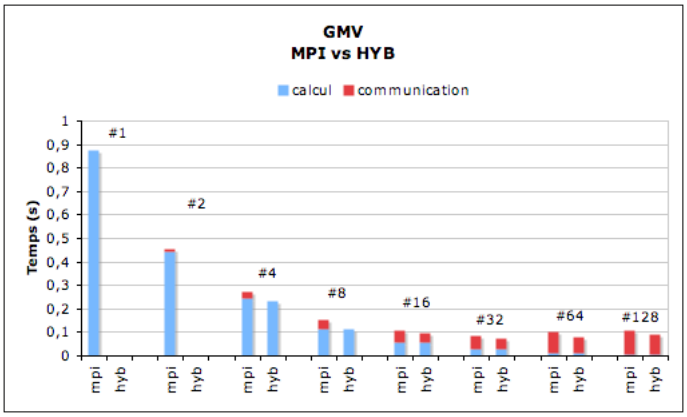


FIGURE 4.13 – Temps d'exécution (MPI et hybride) du benchmark GMV sur le Cluster-Machine.

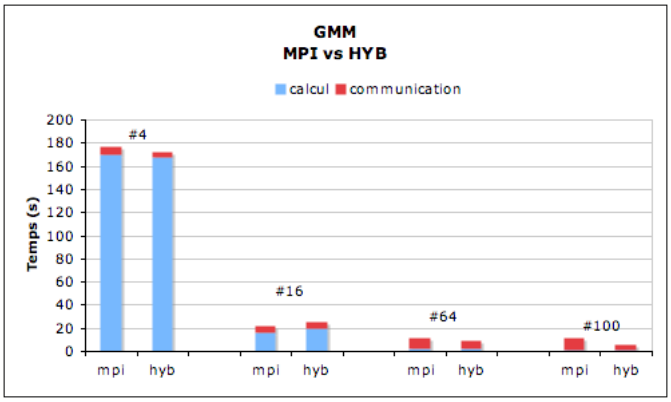


FIGURE 4.14 – Temps d'exécution (MPI et hybride) du benchmark GMM sur le Cluster-Machine.

l'efficacité et le passage à l'échelle des codes `BSP++` sur des architectures hybrides hétérogènes.

TABLE 4.3 – Temps d'exécution des différents benchmarks avec la version hybride MPI+Cell

Bench	1 Cell	2 Cells	3 Cells
$InProd(64 \times 10^4)$	0.032	0.017	0.013
$GMV(1024 \times 1024)$	1.14	0.627	0.424
$GMM(512 \times 512)$	3.55	1.853	1.25

4.9 Conclusion

La définition d'outils de programmation pour les architectures hybrides de type grappes de multi-cœurs et grappes équipés d'accélérateurs n'est pas une tâche aisée. En effet, la difficulté réside dans le compromis entre les performances et l'abstraction de la machine cible. Nous avons néanmoins montré qu'un tel outil est possible en se basant d'une part sur un modèle hiérarchique de haut niveau qui prend intuitivement en considération l'hybridation de la machine cible et d'autre part sur l'exploitation de l'aspect *générique* et de la *méta-programmation* des **Template C++**.

Notre bibliothèque `BSP++` tire parti de ces techniques et permet donc de concilier une haute abstraction d'utilisation et une efficacité en terme de performances. Ces principaux apports sont :

- Se basant sur un modèle hiérarchique, elle permet l'emboîtement des fractions de code pour la construction des codes hybrides et hiérarchiques à plusieurs niveaux, tirant ainsi profit des différents niveaux de parallélisme offert par ces architectures.

- Utilisant une interface utilisateur intuitive constituée d'un ensemble complet et réduit de primitives et de concepts, `BSP++` offre une haute abstraction et une facilité d'utilisation.

- Sur plusieurs benchmarks, les performances des versions `BSP++` montrent une égalité et un rapprochement des performances comparées à celles des versions écrites à la main. D'autre tests et applications seront présentés dans le chapitre 6 qui confirmeront la facilité, l'efficacité et le passage à l'échelle des programmes `BSP++`.

Une autre manière de juger la pertinence de `BSP++` est de déterminer si elle satisfait les exigences et contraintes énumérées par Skillicorn et Talia vu dans le chapitre précédent : basée sur une interface de haut niveau, les trois premiers critères, à savoir la facilité d'utilisation et de compréhension ainsi que l'indépendance de l'architecture, sont satisfaits. Les critères de prédiction et de performance sont les bases de construction de `BSP++`.

LE FRAMEWORK BSPGEN 5

5.1 Objectifs

Dans le chapitre précédent, nous avons montré la facilité de la programmation avec `BSP++`. Cette facilité découle, d'une part de son haut niveau d'abstraction lié à l'utilisation d'une interface fonctionnelle, intuitive et facilement intégrable avec les standards C/C++ et d'autre part de sa généricité. En effet, on a vu qu'avec le même code de base, on arrive à générer le code pour cinq architectures cibles différentes : SMP, cluster, Cell BE, cluster de SMP et cluster de Cell BE. Cependant, la génération du code pour les deux dernières architectures nécessite une petite intervention du programmeur. Effectivement, on a vu que le choix des parties de code pour une programmation hybride (MPI+OpenMP ou MPI+Cell) ainsi que l'insertion de la fonction $split$ et des itérateurs sont à la charge du programmeur.

Malheureusement, ce choix n'est pas une tâche simple et on peut même le qualifier de critique puisqu'il a un impact direct sur les performances du code généré. La programmation hybride est largement étudiée par la communauté scientifique. Dans [77, 4, 19], on a considéré la programmation MPI+OpenMP comme étant le modèle adéquat pour les architectures hybrides. On a montré une amélioration des performances en utilisant une programmation hybride sur des grappes de multiprocesseurs. Néanmoins, dans la littérature, il existe d'autres travaux

[78, 79] qui montrent la dégradation des performances en ajoutant OpenMP à des programmes MPI. Dans [4], les auteurs ont conclu que les performances d'un programme hybride dépendent d'une part de l'architecture cible (nombre de nœuds, nombre de cœurs/nœud, débit du réseau d'interconnexion, etc.) et d'autre part de l'application : les performances d'un programme hybride dépendent de la taille des données, des schémas de communication utilisés, du ratio calcul/communication, ... En effet, les facteurs influant sur les performances des architectures hybrides sont nombreux, complexes et interdépendants :

- **efficacité des communications MPI** : les performances des constructions MPI (point à point, diffusion, all-to-all, tailles des messages ...), les latences d'interconnexion et la bande passante sont des facteurs clés. Ces facteurs découlent des problèmes liés aux applications telles que les types de routine MPI (point à point, collective), la taille des messages et des problèmes liés à l'architecture de la machine cible tels que le type du réseau de connexion et la bande passante.

- **efficacité de la parallélisation du calcul** : les performances d'un code parallèle hybride peuvent être réduites à causes de différent facteurs. Dans le cas des grappes SMP, ce sont les primitives OpenMP de section critique, les frais généraux dus à la gestion des threads et le faux partage. Dans le cas des cluster de Cell BE, ce sont les transferts de données entre CPU et accélérateur et le démarrage du kernel.

- **interaction MPI et Threads (OpenMP ou Cell BE)** : les problèmes d'équilibrage des charges et les threads inactifs à l'intérieur d'une communication MPI réduisent l'efficacité parallèle d'un code hybride.

Pour obtenir un programme hybride efficace, on doit déterminer le nombre de processus MPI et le nombre de threads OpenMP/Processeur Cell BE qui doivent être lancés sur une plateforme donnée pour une taille de données fixée. Avec N nœuds, chacun d'eux ayant p cœurs, il est courant de lancer N processus MPI, chacun avec p threads. Il y a deux sources possibles de pertes de performance. Premièrement, comme le volume de communication entre processus (nœuds) augmente avec le nombre de processus, la surcharge conséquente pourrait devenir un inconvénient. Deuxièmement, la concurrence entre les threads (faux partage des données, surcoût et synchronisation explicite) est également un facteur qui limite le passage à l'échelle, ce qui conduit à la stagnation des performances ou un ralentissement. En plus de cet aspect performance, il y a un effort technique considérable à fournir pour le développement et le déploiement des programmes hybrides. Pour résoudre ces problèmes, les développeurs ont besoin de solutions en termes d'outils et des bibliothèques de parallélisation automatique.

Dans l'objectif de faciliter la programmation efficace de ce type d'architec-

ture, nous avons proposé un $Framework$, nommé BSPGen. BSPGen [80] se base sur le modèle de coût de BSP++, estime le temps d'exécution pour toutes les configurations possibles, détermine la succession des configurations la plus optimale et génère le code correspondant en utilisant la bibliothèque BSP++. BSPGen prend en entrée un simple fichier XML décrivant l'algorithme BSP (la succession des super-étapes) et un fichier contenant le code des fonctions séquentielles et produit en sortie les sources BSP++ adéquates ainsi qu'un $Makefile$ pour la compilation et le déploiement des différentes étapes sur différents niveaux de la hiérarchie de la machine cible.

5.2 L'existant

La programmation hybride a été intensivement étudiée dans la littérature sous différents aspects. Ils peuvent être classés en deux catégories : analyseur de code source et de performance et générateur de code.

5.2.1 Analyseur de code source et de performance

Il existe plusieurs travaux dans ce domaine. Cependant la plupart des outils [81, 82, 83, 84] ont pour but d'aider le programmeur à améliorer les performances de son code. En effet, ces outils évaluent les performances d'un code à l'exécution (runtime). Par exemple, l'outil PAPI [83] lance le code de l'utilisateur et utilise de nombreux compteurs pour calculer le taux d'utilisation du CPU (nombre de cycles, défauts de cache et de TLB, ...), le nombre d'accès à la mémoire et la quantité de données transférées pour chaque partie du code utilisateur.

La majorité des travaux sur l'analyse statique des codes vise l'aide du compilateur, avec les optimisations effectuées tel que : la prédiction des branchements et le préchargement des caches [85], ainsi que l'analyse des motifs d'accès des boucles sous le modèle polyédrique [86, 87, 88]. Toutes fois, il existe quelques travaux tels que [89] pour la prédiction des temps d'exécutions à la compilation. Dans ces travaux, les auteurs ont implémenté un analyseur pour prédire le temps d'exécution d'un code MPI. Ils ont développé un compilateur qui analyse le code source et repére les routines MPI pour estimer le temps de communication. L'estimation de temps de calcul est effectuée par le analyseur syntaxique qui compte le nombre d'opérations directement à partir du code source utilisateur.

Dans [90], avec le projet PERCS, pour estimer les performances des applications séquentielles et/ou MPI sur des architectures parallèles, les auteurs ont combiné deux systèmes de profils : $Profile\ Machine$ et $signature\ d'application$. Le profil machine est un système qui récolte les informations sur l'architecture cible. Le système est constitué de plusieurs benchmarks et tests pour déterminer

les caractéristiques du réseaux de communication et la puissance de calcul de la machine. La signature d'une application est un système dont le but est de catégoriser les applications en fonction des schémas de calcul et de communication. Les auteurs ont utilisé ces deux systèmes en introduisant la notion de $convolution$. Cette notion est un système de mappage qui permet de faire correspondre la signature de l'application avec le profil de la machine.

Basé sur l'idée de PERCS, Barbara Chapman et al dans [91] implémentent un outil pour estimer les performances d'un code hybride MPI+OpenMP. Dans leurs travaux, les auteurs ont utilisé leur propre compilateur nommé $OpenUH$ [92]. En utilisant ce même compilateur et dans d'autres travaux, ces auteurs [93] ont proposé un modèle pour estimer à la compilation le temps d'exécution d'un programme OpenMP. Ce modèle estime le temps de calcul CPU, le temps d'accès mémoire et les surcoûts des directives OpenMP.

5.2.2 Générateur de code

Divers outils et bibliothèques pour la génération des codes parallèles ont été développés et présentés dans la littérature. Parmi ces outils, on peut citer toutes les bibliothèques et langages de programmation parallèle déjà vus dans les chapitres précédents tels que [27, 35, 28, 66, 22, 67]. Tous ces outils ont pour objectif de fournir une abstraction et une simplicité de programmation. En plus de tous ces outils, on peut citer la bibliothèque $LLCOMP$ développée par [94]. Dans ce papier, les auteurs présentent un outil pour la génération automatique d'un code hybride MPI+OpenMP.

Le code généré par cette bibliothèque est basé sur un langage de programmation parallèle de haut niveau LLC [94]. LLC est un langage à base des squelettes algorithmiques et son utilisation se base sur des directives en C comme dans OpenMP. LLCOMP est une extension du langage LLC qui prend en considération les primitives OpenMP. L'outil $LLCOMP$ est un compilateur source à source qui, à partir d'un code séquentiel écrit en LLC, génère le code hybride MPI+OpenMP.

5.3 L'architecture globale de l'environnement BSPGen

L'idée de base de notre environnement consiste en une génération automatique de code hybride ainsi que la détection de la configuration la plus efficace à chaque étape de l'algorithme parallèle, pour une application donnée, sur une architecture donnée. Ces principes de base sont les suivants :

– Utiliser le modèle de coût de `BSP++` pour estimer le temps d'exécution de

chaque étape du programme.

– Déterminer la suite des meilleures configurations parmi toutes les configurations possibles.

– Générer le code correspondant à cette suite de configurations.

Pour l'estimation du temps d'exécution, `BSPGen` suit l'idée de [91] et celle de [90] dans le projet PERCS : utiliser un mécanisme de $profilage$ pour récolter les caractéristiques de la machine cible. Cependant, elles diffèrent du fait que nous utilisons une analyse du code source au lieu de mécanisme des $signatures\ d'applications$ et l'opération de $convolution$ précédemment décrits.

L'architecture globale de `BSPGen` est constituée de trois modules : un analyseur, un chercheur et un générateur. `BSPGen` prend en entrée un fichier XML contenant la description du code parallèle et un fichier contenant la liste des fonctions séquentielles de calcul constituant le programme et il génère en sortie le code hybride MPI+OpenMP ou MPI+OpenMP/Cell BE. Globalement, notre framework fonctionne comme suit : d'abord l'$Analyseur$ estime le temps d'exécution de chaque fonction dans la liste de l'utilisateur. Comme le temps d'exécution dépend de la taille des données et comme celle-ci est généralement inconnue à la compilation, l'$Analyseur$ génère une formule mathématique pour chaque fonction avec les tailles des données et le nombre de processeurs comme variables. Ensuite, le module $Recherche$ utilise ces formules et les informations extraites des motifs de communication depuis le fichier XML ainsi que les caractéristiques de la machine cible fournies par le système "$profiler$", pour estimer le temps d'exécution pour toutes les configurations possibles. Finalement, après la détermination du plus court chemin, le $Generateur$ produit le code correspondant en utilisant la bibliothèque `BSP++`. L'architecture globale du framework `BSPGen` est présentée dans la figure 5.1.

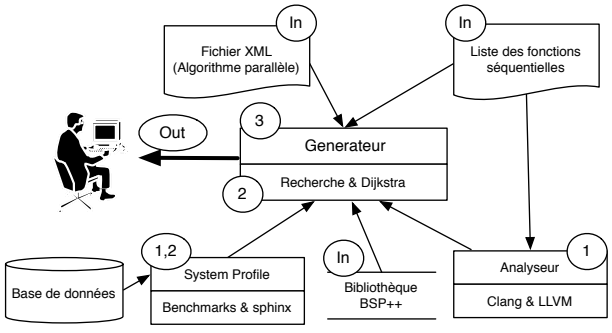


FIGURE 5.1 – L'architecture globale du framework BSPGen.

Dans la suite de ce chapitre, nous allons décrire le fonctionnement de chaque module de notre framework ainsi que le workflow global entre ces différents modules. Pour faciliter cette description, nous avons utilisé l'exemple du produit scalaire interne $inprod$ pour exposer les détails des entrées/sorties de chaque module.

5.3.1 Le module Analyseur

Comme signalé précédemment, pour estimer le temps d'exécution, nous nous basons sur le modèle de coût de `BSP++`. Le coût d'exécution de ce modèle est la somme des coûts de chaque super-étape. Par ailleurs, le coût d'une super-étape est simplement la somme des temps de calcul, de communication et de synchronisation. Cependant, comme les architecture cibles sont hybrides, il y a un autre paramètre à prendre en considération : le coût de changement de niveau T_{lc} (Time Level Changing). Ainsi le coût d'une super-étape peut être exprimé sous la forme :

$$\delta = T_{comp} + T_{comm} + T_{lc}$$

où T_{comp} correspond au temps de calcul asynchrone de chaque processeur $W_{i_{max}}$. T_{comm} représente le cumul des temps de communication et de synchronisation dans le modèle de base h * g+L.

Le rôle principal du module $Analyseur$ est d'affecter une valeur à chaque composante de cette formule. Le module $Analyseur$ est constitué de deux sous modules : le sous-module de calcul et celui de communication.

5.3.1.1 Sous-module de calcul

Ce module est principalement utilisé pour estimer le coût de T_{comp}. Estimer le temps d'exécution d'une fonction statiquement (à la compilation) est connu pour être un problème **indécidable** car le nombre d'itérations peut dépendre de la valeur d'une donnée et pas seulement de la taille des données. Cependant, la majorité des algorithmes de calcul scientifique, de traitement d'images et du calcul parallèle en général sont des algorithmes ***réguliers*** : le nombre d'itérations dépend uniquement de la taille des données et non de la valeur des données. Pour cette famille de programme, nous avons proposé un estimateur qui compte le nombre de cycles nécessaires à l'exécution d'un code source sur une architecture donnée.

L'utilisation d'un analyseur syntaxique de code source, comme dans [89] qui compte le nombre de cycles pour chaque ligne de code, ne donne pas une estimation précise. Ce manque de précision est dû à toutes les optimisations (modifications) que le compilateur effectue sur le code source comme par exemple

la fusion de boucles, l'élimination de code mort, etc. Pour une meilleure précision de l'estimation de temps de calcul, nous avons choisi de parser et d'analyser le code généré par le compilateur avec toutes les optimisations pour une exécution rapide (compilée avec l'option $-O3$). Pour ce faire, nous avons utilisé le compilateur $Clang$ [95] pour générer le code intermédiaire $bytecode$ pour chaque fonction dans la liste utilisateur des fonctions séquentielles appelées par l'algorithme parallèle BSP. Ensuite, nous avons implémenté une nouvelle passe dans le compilateur LLVM [96] qui, en utilisant le $bytecode$ généré par Clang, compte le nombre de cycles pour chaque fonction.

Les passes sont les parties les plus importantes dans le système LLVM, car elles sont responsables de toutes les transformations et des optimisations que réalise un compilateur [96]. Notre passe est basée sur le module $FunctionPass$ [97] qui est une sous-classe dérivée de la classe $PASS$ et qui a un comportement local exécuté pour chaque fonction indépendamment des autres fonctions dans le programme.

Globalement, notre passe fonctionne hiérarchiquement comme suit : pour chaque fonction dans le $bytecode$, pour chaque boucle dans la fonction, le analyseur syntaxique détermine le bloc de base. Ensuite, pour chaque instruction dans le bloc de base, il compte le nombre de cycles nécessaire à l'exécution de cette instruction en utilisant une base de donnée initialisée par le système $profiler$ (voir section 5.3.1.2) qui fait correspondre à chaque instruction de base le nombre de cycles correspondant.

Comme notre modèle estime le coût d'une façon pessimiste (pire cas), pour les blocs de test *if-else* par exemple, on compte le nombre de cycles pour chaque branche et on prend le maximum. Pour les boucles, on calcule le nombre de cycles du bloc de base de la boucle et on le multiplie par le nombre d'itérations. Souvent, le nombre d'itérations n'est pas connu à la compilation. De ce fait, notre passe génère une formule *analytique* où le nombre d'itérations (taille des données, nombre de processeurs ...) est un paramètre. La valeur de ce paramètre est connue à l'exécution et elle est récupérée à partir du fichier XML.

Le sous-module de calcul est lui même composé en deux parties : **Partie Processeur** et **Partie cache**.

Partie Processeur :

Cette partie du module estime le temps passé sur le CPU à faire du calcul. Elle compte le nombre d'opérations exécutées sur les unités de calcul du CPU. Ce modèle prend en considération la différence entre ces unités de calcul (FP, ALU, SSE,...). De plus, il considère les fonctions intrinsèques comme la SQRT

par exemple comme des opérations de base car les architectures des microprocesseurs les considèrent ainsi. Les opérations prises en compte par ce modèle incluent les opérations de chargement et de rangement (load/store) de/vers la mémoire en supposant qu'il n'y a pas de défauts de cache. Autrement dit, ce modèle prend en compte le nombre de cycles des opérations en supposant que les données se trouvent dans le cache L1 et il ne prend pas en considération les coûts des défauts de caches.

Le modèle de processeur peut être exprimé comme la somme des coûts C_i (nombre de cycles) de chaque opération i.

$$Cost_{processor} = \sum_{i}^{Ops} C_i$$

Notre analyseur syntaxique récupère les coûts matériels C_i à partir de la base de données du système profiler. Ces valeurs peuvent être obtenues soit à partir des manuels et des spécifications des fabricants de processeur ou en utilisant des petits benchmarks spécifiques [89].

Pour rappel, le code BSP de l'algorithme produit scalaire $inprod$ vu au chapitre précédent contient deux fonctions séquentielles. Le listing ci-dessous correspond à la liste des fonction séquentielles que l'utilisateur doit fournir comme fichier d'entrée à `BSPGen`. Le fichier contient deux fonctions : une pour le calcul du produit localement et l'autre pour faire l'accumulation des résultats partiels.

Listing 5.1 – liste des fonctions séquentielles : Exemple produit scalaire

```
1 double inner_product( vector<double> const& in)
2 {
3 return std::inner_product(in.begin(),in.end(),in.begin(),0.0);
4 }
5 double Global_sum( vector<double> const& in)
6 {
7  return std::accumulate(in.begin(),in.begin(),0.0);
8 }
```

Pour chacune des deux fonctions, le module $Analyseur$ appelle Clang pour générer le $bytecode$. Ensuite, il fait passer le byte code de chaque fonction dans notre passe LLVM qui génère une formule analytique pour chacune d'elles. Le code 5.2 représente le $bytecode$ du nid de boucles de la deuxième fonction dans la liste (la fonction Global_sum). L'appel au compilateur Clang est effectué via la commande suivante :

clang++ userfile.cpp -O3 -S -emit-llvm -o output.b

L'utilisation de l'option $-emit-llvm$ est pour indiquer que la sortie (le fichier $output.bc$) sera utilisé par le compilateur LLVM.

Listing 5.2 – le code intermédiare "bytecode" de la fonction Global_sum

```
1 for.body:          ; preds = %for.body, %bb.nph
2   %indvar = phi i64 [ 0, %bb.nph ], [ %tmp13, %for.body ];<i64>[#uses=1]
3   %res.09 = phi double [ 0.000000e+00, %bb.nph ], [ %add, %for.body ]
4   %tmp13 = add i64 %indvar, 1  ; <i64> [#uses=3]
5   %arrayidx = getelementptr double* %A, i64 %tmp13 ; <double*>[#uses=1]
6   %tmp4 = load double* %arrayidx       ; <double> [#uses=1]
7   %add = fadd double %res.09, %tmp4  ; <double> [#uses=2]
8   %exitcond = icmp eq i64 %tmp13, %tmp12; <i1> [#uses=1]
9   br i1 %exitcond, label %for.end, label %for.body
```

Pour le *bytecode* 5.2, le sous module de processeur compte 1 cycle pour l'instruction add de la ligne 4 et l'instruction $icmp$ de la ligne 8, et 2 cycles pour l'instruction $load$ de la ligne 6 et l'instruction $fadd$ de la ligne 7. Soit un total de 6 cycles dans un bloc de base d'une boucle de N itérations. Le analyseur syntaxique compte aussi 7 cycles à l'extérieur du nid de boucles correspondant à l'initialisation et retourne la formule numérique suivante :

$$Global\_sum = 7 + 6 * N$$

Pour les fonctions irrégulières où le nombre d'itérations dépend de la valeur des données, notre analyseur syntaxique suit la même stratégie et construit une formule numérique avec une variable pour chaque nid de boucles. La différence est qu'au lieu de remplacer la variable par la taille des données, elle est remplacée par la **complexité de la fonction** qui est passée comme paramètre dans le fichiers XML. Par exemple, pour la fonction de tri rapide (quick sort), l'analyseur retourne la formule suivante :

$$11 + 6 * N1 * N2$$

L'utilisateur dans le fichier XML décrit l'algorithme parallèle, en indiquant que la variable $N1$ correspond à la valeur n (taille des données) et $N2$ à la valeur $log(n)$ car la complexité de cette fonction est $n * log(n)$.

Notre framework donne aussi la possibilité au programmeur d'utiliser des fonctions déjà compilées en les considérant comme des boîtes noires. Dans ce cas, et comme il n'y a pas moyen d'accéder au code source pour estimer le temps d'exécution, l'utilisateur est obligé de donner le temps d'exécution comme paramètre dans le fichier XML.

Partie Cache

Le rôle de ce modèle est l'estimation de temps d'accès aux données dans les différents niveaux de la hiérarchie mémoire. Comme mentionné auparavant, le modèle de processeur suppose que tous les accès se font dans le niveau L1 de la hiérarchie mémoire. En fait, l'accès aux données peut engendrer des accès aux différents niveaux de la hiérarchie mémoire. En effet l'accès à une variable qui

ne se trouve pas dans le cache L1 nécessite un accès à un des nivaux supérieurs. Plus la donnée s'éloigne du processeur dans la hiérarchie, plus le temps d'accès augmente. Cette perte de temps dans les accès est liée aux **défauts de cache**. Comme le temps d'exécution dépend du nombre de défauts de cache à chaque niveau de hiérarchie, le module cache estime ce nombre pour chaque niveau. Le modèle de coût de ce sous-module est exprimé comme suit :

$$Cost_{cache} = \sum_{i}^{levels} (M_i * Penalty_i)$$

M_i est le nombre de défauts de cache de chaque niveau i. $Penalty_i$ est exprimé en nombre de cycles et il représente le nombre de cycles nécessaires pour accéder au niveau i de la hiérarchie mémoire. Comme les coûts des opérations C_i du modèle de processeur, $Penalty_i$ est initialisé dans le fichier système $profiler$ à partir des manuels des constructeurs.

Pour estimer le nombre de défauts de cache M_i, le analyseur syntaxique accumule l'**"empreinte"** de chaque instruction dans le $bytecode$. L'empreinte représente le nombre de données référencées (load/store) multiplié par la taille de la donnée référencée (β). Si le résultat dépasse la capacité du cache de niveau i, alors on suppose un défaut de cache à ce niveau. Le modèle de cache suit la même stratégie de fonctionnement que le modèle du processeur pour ce qui concerne l'analyse et le comptage. Analytiquement M_i est calculé comme suit :

$$M_i = (\sum_{j}^{inst} (ref_j * \beta_j))/Capacity_i$$

Ce sous-modèle d'estimation de défauts de cache n'est pas précis car il ne prend pas en considération la localité spatiale et temporaire des données dans les caches [98] et il n'utilise pas la notion de **réutilisation des caches**, de plus il ne prend pas en considération l'allocation des registres et le *spill* code généré par le compilateur. Mais, dans notre estimation, nous n'avons pas besoin de plus de précision car :

- Pour ce qui concerne le code du Cell, on n'utilise pas ce sous-modèle car il n'y a pas de caches sur le Cell BE et les codes sont écrits d'une manière qui garantit que les données dont on a besoin sont dans la mémoire locale (condition nécessaire pour le bon fonctionnement du Cell).

- L'estimation du temps de calcul T_{comp} est utilisé juste pour décider du nombre de processeurs et du mode (MPI ou hybride MPI+OpenMP) à utiliser, en calculant le ratio $\frac{T_{comp}}{T_{comm}}$. En effet, comme les deux termes de la fraction sont fonction du nombre de processeurs et qu'ils varient en sens

inverse, le calcul de ce ratio représente une bonne métrique pour vérifier le passage à l'échelle d'une application donnée sur une architecture donnée. Et comme les deux codes générés MPI et OpenMP par notre bibliothèque (`BSP++`) sont en style SPMD, ils ont les mêmes motifs d'accès à la mémoire, ce qui donne les mêmes comportements des accès aux caches. Par conséquent, cette estimation n'intervient pas dans le choix du mode.

Avec ces deux modèles, le analyseur syntaxique peut estimer le temps de calcul T_{comp} pour toutes les fonctions séquentielles et retourner une formule pour chaque fonction qui sera ensuite utilisée par le module $Recherche$ pour calculer le temps d'exécution pour toutes les configurations possibles. Pour ce faire, le module $Recherche$ a besoin d'une estimation du temps de communication T_{comm} qui sera fourni par le sous-module communication de l'analyseur.

5.3.1.2 Sous-module de communication

Ce modèle de communication exige les valeurs de g et L car il est basé sur la formule suivante h * g+L. Utilisés dans le modèle `BSP++`, ces deux paramètres dépendent du niveau de la hiérarchie de la machine BSP cible (MPI, OpenMP ou Cell BE) et du nombre de processeurs utilisés à chaque niveau [72]. Dans cette optique, ces deux paramètres sont exprimés en fonction du niveau de la sous-machine cible k et du nombre de processeurs P_k à chaque niveau k comme suit : $g_k(P_k)$ et $L_k(P_k)$ respectivement. En utilisant ces deux expressions, le temps de communication est :

$$T_{comm} = h_k * g_k(P_k) + L_k(P_k)$$

De manière similaire aux coûts matériels des opérations et aux pénalités des défauts de cache, les valeurs de g et L sont stockées dans le fichier système $profiler$. Mais contrairement aux deux premières valeurs où elles sont récupérées manuellement à partir des spécifications du constructeur, les paramètre de communication sont automatiquement recueillis par la partie *système profiler* du module de communication.

Le *système profiler* est appelé une seule fois pour chaque machine BSP et il génère un fichier contenant les valeurs de L et g en fonction du nombre de processeurs et du niveau dans la hiérarchie. Pour plus de précision, le nombre de processeurs P_k utilisés dans l'expression de L et g est lui même exprimé en fonction de deux paramètres : n le nombre de nœuds utilisés et c le nombre de cœurs dans chaque nœud sous forme $L_k(n_k, c_k)$. En effet, les temps de communication et de synchronisation ne sont pas identiques pour deux configurations différentes utilisant le même nombre de processeurs.

$$L_{mpi}(2, 4) \neq L_{mpi}(4, 2)$$

Pour déterminer les valeurs de L et g, le *système profiler* combine l'utilisation de quelques benchmarks spécifiques et d'un outil nommé $Sphinx$ [99]. L'outil Sphinx est une suite de microbenchmark parallèles, composés de noyaux pour les tests de performances des fonctions, routines et directives de MPI, OpenMP et pthreads. Il a été adapté de la suite des tests spécifiques de MPI_Karlsruhe (SKaMPI) par R. S. Bronis et autres membres du projet PSE/ASDE [100]. La suite complète est implémentée en C et a été exécutée sur une grande variété de plates-formes.

Le cœur des tests de Sphinx fournit un mécanisme flexible pour l'exécution des tests de performance. L'action mesurée, comme un message ping-pong, est accessible via un pointeur de fonction. Différents threads ou tâches peuvent exécuter des fonctions différentes, ce qui permet la mesure très complexe des actions parallèles. Le noyau Sphinx fournit un mécanisme flexible pour mesurer le temps des appels répétés (itérations dans la terminologie de Sphinx) de l'action. Sphinx mesure le temps pour plusieurs répétitions et donne en sortie plusieurs valeurs arithmétiques comme la moyenne et l'écart-type.

Comme le code généré par `BSPGen` est écrit avec `BSP++`, nous avons besoin d'évaluer les performances uniquement d'un sous ensemble des routines MPI et primitives OpenMP. Incluant `MPI_all_to_all` et `MPI_allgather` pour les primitives de communication de `BSP++`(`put` et `proj` voir chapitre 4), les $Barriers$ de synchronisation pour MPI et OpenMP et pour finir, évaluer l'surcoût de la directive $OMP\ PARALLEL\ SECTION$ qui est appelée au démarrage de l'environnement OpenMP. Cette dernière est utilisé pour estimer le T_{lc}. La version actuelle de l'outil Sphinx nous a permis d'évaluer toutes les composantes de ce sous-ensemble.

Pour l'évaluation de ces paramètres pour la version Cell BE, nous avons utilisé les mêmes benchmarks offerts par Sphinx en utilisant notre implémentation de CBE_MPI pour `MPI_all_to_all` et `MPI_allgather` ainsi que la barrière de synchronisation. L'estimation du temps de changement de niveau pour le Cell T_{lc} comprend le temps de lancement du kernel qui représente l'surcoût de lancement des threads sur les SPEs ainsi que le chargement de leur code et le temps de transfert des données entre le PPE et les SPEs. Cependant, comme on utilise des transferts asynchrones sous la notion de multi-buffering, on compte uniquement le temps de chargement de la première itération qui est exprimé comme la multiplication de la taille des données par le débit de transfert entre PPE et SPEs.

Après l'évaluation des caractéristiques architecturales de chaque niveau de la machine cible par le système $profiler$, le module de communication de l'analyseur utilise ces résultats pour estimer le temps de communication comme étant

la somme des temps de communication de chaque niveau :

$$T_{Comm} = \sum_{i}^{k} T_{SubComm_i}$$

où le temps de communication au i^{eme} niveau de la machine BSP est exprimé comme suit :

$$T_{SubComm_i} = h_i * g_i(P_i) + L_i(P_i) + T_{lci}(N)$$

5.3.2 Le module Graphe de recherche

Comme indiqué au début de ce chapitre, les performances d'un programme hybride dépendent de plusieurs facteurs. Principalement, ces performances sont fonction du nombre de nœuds utilisés (processus MPI) et du nombre de threads OpenMP / Cell BE processeur utilisés. Ces nombres eux mêmes sont conditionnés par les caractéristiques de l'architecture cible et celles de l'application en termes de taille de données, motifs de communication et ratio calcul/communication. Ainsi prédire les performances d'une application donnée sur une architecture hybride revient à déterminer les nombres de processus MPI, Threads OpenMP et nombre de Cell BE pour chaque étape du programme.

Un moyen pour trouver ces nombres serait de résoudre analytiquement le système des équations différentielles de la formule de coût du modèle `BSP++` sur chaque variable. Cette solution est très complexe car on doit déterminer :

- les valeurs de ces nombres qui donnent le minimum global sur l'exécution entière de programme.
- les valeurs de ces nombre correspondant à chaque étape de programme pour pouvoir personnaliser leur utilisation à l'exécution. Cette contrainte devient ingérable pour des programmes constitués de plusieurs étapes.

Un autre moyen plus simple en terme d'implémentation et plus efficace en terme de performance est la **recherche exhaustive** dans l'espace de toutes les configurations possibles. Pour se faire, on construit un graphe G et on explore toutes les configurations valides. Une configuration valide est une configuration cohérente : par exemple, une configuration où le nombre de threads OpenMP est supérieur au nombre de cœurs physiques dans le nœud est une configuration invalide. Les conditions de validité d'une configuration seront exposées au fur et à mesure.

On dénote le graphe $G = (V, E)$ le Graphe Acyclique Direct où V est l'ensemble des sommets du graphe et E représente l'ensemble des arcs du graphe. Chaque sommet représente une configuration d'exécution d'une super-étape avec

un nombre de nœuds utilisés, un nombre de processus MPI, un nombre de threads OpenMP et un nombre d'accélérateurs utilisés. Une configuration est représentée hiérarchiquement comme étant un $quintuplet\ V(s(n\{m\{o\{a\}\}\}))$ avec :

- s : le numéro de la super-étape à exécuter.
- n : le nombre de nœuds physiques utilisés.
- m : le nombre de processus MPI lancés par nœud physique.
- o : le nombre de threads OpenMP lancés par chaque processus MPI.
- a : le nombre d'accélérateurs lancés par chaque thread OpenMP/ processus MPI.

Le poids d'un arc $V(s_i(n\{m\{o\{a\}\}\}))- > V'(s_{i+1}(n\{m\{o'\{a\}\}\}))$ est le coût d'exécution de la super-étape (s_{i+1}) avec la configuration destination de l'arc ($V'(s_{i+1}(n\{m\{o'\{a\}\}\}))$), sachant que la super-étape précédente (s_i) a été exécutée avec la configuration $V(s_i(n\{m\{o\{a\}\}\}))$. Le coût d'un arc dépend uniquement de configuration de destination car :

- Grâce à la synchronisation au niveau le plus haut de la hiérarchie de la machine BSP (MPI) à la fin de la super étape s_i, on est sûr que quelle que soit la configuration de destination de l'arc (s_{i+1}), la configuration de source est au niveau le plus haut.

- Le coût des $splits$ et des surcoûts de lancement des sous-machines BSP sont inclus dans le coût d'exécution de la configuration s_{i+1}.

Le poid de ces arcs est estimé en utilisant les formules numériques retournées par l'$Analyseur$ et le schéma de communication récupéré à partir du fichier XML.

Le graphe *G* est construit hiérarchiquement. Premièrement, on commence par fixer le nombre de nœuds physiques utilisés. On a un *sous-graphe* pour chaque nombre de nœuds utilisés. Autrement dit, si le cluster est constitué de n nœuds physiques, alors on a n sous-graphes. Ensuite, pour chaque sous-graphe, on choisit le nombre de processus MPI à démarrer. Finalement, pour chaque processus MPI, on détermine le nombre de threads OpenMP et/ou accélérateurs qui doivent être activés.

Comme le cluster est *homogène* (tous les nœuds ont les mêmes caractéristiques matérielles et logicielles) et le modèle de programmation est de style SPMD, tous les nœuds qui sont activés doivent exécuter la même configuration. Formellement, en choisissant deux nœuds du cluster, sur les deux la même configuration est choisie. Le nombre de sommets dans notre graphe est fonction des caractéristiques de la machine cible (nombre de cœurs dans chaque nœud, nombre d'accélérateurs ...) et du nombre d'étapes constituant l'algorithme parallèle de l'application. Le calcul du nombre de sommets (configurations) du graphe est détaillé

comme suit :

$$card(V) = n * \alpha + 2$$

où n, est le nombre de nœuds physiques (nombre de sous-graphes) et α est le nombre de configurations valides dans un sous-graphe. Comme les sous-graphes sont disjoints, nous avons ajouté deux autres sommets virtuels pour permettre la conjonction de tous ces sous-graphes. Ces deux derniers sommets sont considérés comme le début et la fin du graphe G et leur coût est bien évidemment nul. Les configurations valides α sont décomposées en deux sous-ensembles :

- ensemble des configurations valides $V1$ pour les deux premiers niveaux de la hiérarchie de la machine BSP, à savoir MPI+ OpenMP. Une configuration valide dans cet ensemble nécessite que le nombre total des threads lancés dans un nœud est inférieur ou égal au nombre de cœurs physiques dans le nœud. Formellement, le nombre de processus MPI $m \times o$ le nombre de threads par processus doit être inférieur ou égal au nombre de cœurs C.

  $$V1 = \sum_{k=1}^{C} card\{(m,o) \in N^2 : m * o = k : k \leq C, 1 \leq o \leq C\}$$

  La configuration où chaque processus MPI lance un seul thread OpenMP n'a pas de sens en terme de performance et elle est prise en compte par notre modèle sous forme d'une configuration de pure MPI.

- ensemble des configurations valides $V2$ pour les derniers niveaux de la hiérarchie : niveau des accélérateurs. Une configuration dans cet ensemble exige la satisfaction de trois contraintes :
  - 1) le nombre d'accélérateurs à démarrer a doit être inférieur ou égal au nombre d'accélérateurs physiques A dans le nœud.
  - 2) un cœur (processus MPI ou Thread OpenMP) peut lancer un seul accélérateur et un accélérateur ne peut être lancé que par un seul cœur.
  - 3) Il faut que la configuration soit valide pour les conditions des deux premiers niveaux MPI/OpenMP. Autrement dit, le nombre de threads ne dépasse pas le nombre de cœurs

  Avec toutes ces conditions, $V2$ peut être exprimé comme suit :

  $$V2 = \sum_{j=1}^{A} card\{(k,a) : k = m * o \leq C, a \in N, k * a = j : j \leq A\}$$

  Cependant, cette formule inclut les configurations où un cœur démarre plus d'un accélérateur. Il faut soustraire ces configurations. Soit W le nombre total des configurations pour les trois niveaux hiérarchiques de la machine BSP hybride.

  $$W = V1 + V2 - (A - 1)$$

W représente le nombre de configurations valides pour une étape de programme BSP et pour un nombre fixe de nœuds utilisés (pour un sous-graphe). Le nombre de configurations valides (α) dans un seul sous-graphe pour une application de S super-étapes est égale à W, S fois.

$$\alpha = W * S + 2$$

Pour chaque sous-graphe, nous avons ajouté deux sommets virtuels (ce n'est pas une configuration d'exécution) qui agissent comme source (début) et puis (fin) de chaque sous-graphe. En remplaçant la valeur de α dans la première équation, on obtient le nombre de sommets du graphe G

$$card(V) = n * (W * S + 2) + 2$$

Le nombre de processus MPI ne peut pas être modifié au cours de l'exécution : le nombre de processus MPI est fixe et ne peut pas être modifié entre deux super-étapes. La même contrainte s'applique sur le nombre de nœuds lancés puisque le démarrage d'un nœud implique le lancement d'un processus MPI sur celui-ci. Ces contraintes impliquent qu'aucun arc existe entre deux sous-graphes différents et à l'intérieur du même sous-graphe, il n'existe pas d'arc entre deux configurations avec un nombre de processus MPI différent. De ce fait, le nombre d'arcs dans le graphe G est fonction du nombre de configurations valides dans un sous-graphe, du nombre de nœuds physique et du nombre de super-étapes dans l'application.

Pour déterminer le nombre d'arcs $card(E)$ du graphe G, en utilise la définition suivante :

$Vertex_n(s)$: un sous-ensemble de V qui contient tous les sommets d'un sous-graphe n avec la super-étape s. Le sous-ensemble $Vertex_n(s)$ contient toutes les configurations qui s'écrivent sous forme $(s, n\{-\{-\{-\}\}\})$.

Soit D l'ensemble des arcs entre deux super-étapes s_i et s_{i+1}.

$$D = \{(F, H) : F \in Vertex_n(s_i), G \in Vertex_n(s_{i+1}) : F.m = H.metF.n = H.n\}$$

D contient les paires (F,H) de configurations appartenant au même sous-graphe mais concernant deux étapes successives de l'application, sous la contrainte d'avoir le même nombre de processus MPI dans les deux configurations ($F.m = H.m$). Le nombre total d'arcs est calculé comme suit : l'ensemble D contient le nombre d'arcs pour 2 configurations et l'application contient S étapes. On a alors $(S - 1) * card(D)$ arcs dans un seul sous-graphe. En plus, à cause de l'utilisation de sommets virtuels dans chaque sous-graphe, on a W arcs du sommet *début* vers

les configurations de la première étape et le même nombre dans la dernière étape vers le sommet virtuel fin. Donc le nombre total d'arcs est :

$$card(E) = [(S-1) * card(D) + 2 * W + 2] * n$$

n est le nombre de nœuds physiques. Le $+2$ dans le nombre d'arcs et dû aux deux sommets virtuels de début et de fin du graphe global G.

Après la construction du graphe G et le calcul des poids en utilisant les formules numériques de l'$Analyseur$ et les motifs de communication déduits de fichier XML, on exécute l'algorithme de Dijkstra pour déterminer le chemin le plus court dans le graphe. Comme les poids des arcs représentent le temps d'exécution d'une configuration, le chemin le plus court donne la suite des configurations pour l'exécution la plus efficace en utilisant l'algorithme et les fonctions séquentielles de l'utilisateur.

Le fichier XML 5.3 correspond au fichier utilisateur décrivant l'algorithme BSP de notre exemple (le produit scalaire). Ce fichier inclut deux super-étapes : avec la première, chaque unité de calcul effectue un calcul local en appelant la fonction utilisateur $inner - product$ dont le code est représenté dans la liste des fonctions utilisateur 5.1. Suit une phase de communication décrivant le motif de communication. Dans notre exemple, ce motif correspond à une $projection$ des résultats partiels sur tous les processeurs. Avec la deuxième super-étape, chaque processeur calcule la somme globale en appelant la fonction $Global - sum$.

Listing 5.3 – Fichier XML décrivant l'algorithme BSP de l'exemple $innprod$

```
<program  name= inner-product>
 <step id =1>
  <Compute>
    <F_name> inner-product </F_name>
    <Arg> 6400000</Arg>
  </Compute>
  <Communicate>
    <Comm_type> Proj </Comm_type>
  </Communicate>
 </step>

 <step id=2>
  <Compute>
    <F_name> Global-sum </F_name>
      <Arg_Exp> NB_PROC </Arg_Exp>
  </Compute>
 </step>
</program>
```

Le chemin le plus court du graphe correspondant à notre exemple en utilisant les informations de la machine BSP (2 nœuds, 4 cœurs/nœud, 0 accélérateur) issues du *système profiler*, est illustré en gras dans la figure 5.2. Il montre que

la première étape est exécutée sur les deux nœuds avec 4 cœurs par nœud en utilisant le modèle hybride MPI+OpenMP. Cependant, comme la deuxième étape nécessite un temps de calcul négligeable (la taille des données correspond au nombre de processus MPI utilisés. Dans notre cas, il est égal à 2), elle est exécutée en mode pure MPI avec 1 seul processus sur chaque nœud.

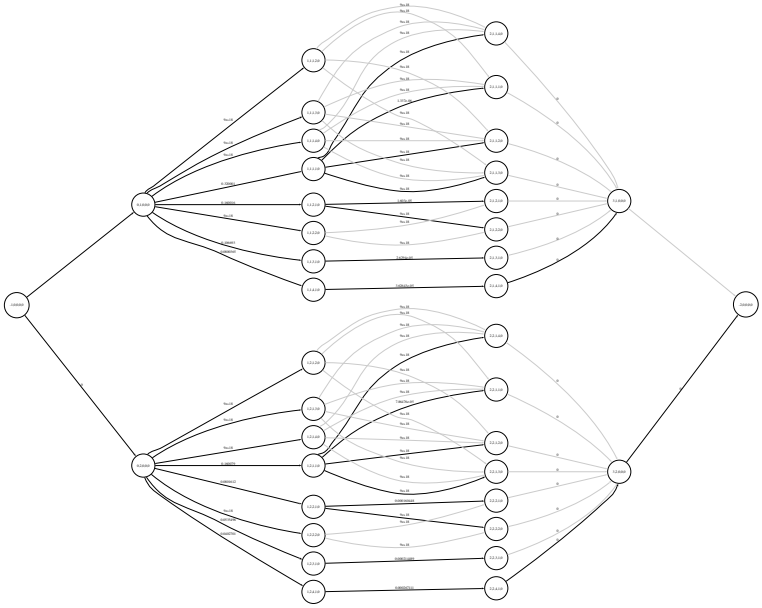


FIGURE 5.2 – Le graphe correspondant au produit scalaire (inner prod) de l'exemple. Les arcs en gras représentent le chemin le plus court.

5.3.3 Le module Générateur de code

Après la détermination de la suite des configurations constituant le chemin le plus court, le module *Generateur* produit le code correspondant en utilisant la bibliothèque `BSP++`. Les codes générés par ce module contiennent des primitives `BSP++` et des appels aux fonctions définies dans la liste des fonctions séquentielles utilisateur.

Les portions hybrides de code sont générées en suivant la stratégie d'hybridation introduite dans le chapitre précédent. Nous rappelons que cette stratégie consiste globalement à remplacer la fonction de calcul dans une super-étape d'un niveau supérieur par une fonction contenant toute la super-étape compilée avec le mode du niveau inférieur dans la hiérarchie de la machine BSP.

Toutefois, pour certains algorithmes, remplacer la fonction de calcul uniquement par la super-étape donne un code hybride sémantiquement incorrect. En effet, pour obtenir un code hybride correct pour ces algorithmes, il faut que la fonction de calcul soit remplacée par une fonction contenant **tout le code BSP** (toutes les super-étapes) et pas seulement la super-étape en question. Par exemple, pour l'algorithme du produit scalaire, la fonction hybride qui remplace la fonction de calcul dans la première étape nécessite l'appel aux deux super-étapes : le calcul local et l'accumulation des résultats après leur projection. Autrement, si on ne fait pas l'accumulation (qui constitue la deuxième super-étape dans l'algorithme BSP) dans la fonction hybride, le résultat de retour ne sera pas compatible avec la deuxième étape de code MPI.

Pour détecter cette catégorie d'algorithme où la stratégie d'hybridation prend en considération la totalité de code BSP dans la portion hybride, nous avons utilisé la notion **d'appel récursif**. Nous définissons la notion **d'appel récursif** comme étant la situation où une fonction de calcul dans une super-étape a le **même nom** que le nom du programme BSP défini dans le fichier XML. En utilisant cette notion, quand le module *chercheur* et/ou *générateur* détecte un appel récursif, il estime, et génère le code hybride en incluant tout le code BSP. Comme toute notion de récursivité a besoin d'un point d'arrêt, la nôtre se base sur le niveau de la hiérarchie d'hybridation dans la machine BSP. En effet, quand le dernier niveau d'hybridation dans la hiérarchie de la machine détecte un appel récursif, il appelle seulement et directement la fonction de calcul utilisateur.

Listing 5.4 – le code BSP++ hybride du produit scalaire : Partie OpenMP

```
1  double omp_inner_prod( vector<double> const& in, int argc, char ** argv)
2  {
3    double value;
4    BSP_HYB_START(argc, argv)
5    {
6      result_of_split< vector<double>,linear() > v = split(in);
7      *r = inner-product (*v);
8      result_of_proj<double> exch = proj(r);
9      // step 2
10     value = Global-sum( exch.begin() );
11     synch();
12   }
13   BSP_HYB_END()
14   return value;
15 }
```

Les Listings 5.4 et 5.5 présentent le code hybride pour l'exemple produit scalaire généré par le module *Generateur* en utilisant les primitives et les concepts de la bibliothèque `BSP++`. Dans le code 5.4, la fonction *omp_inner_prod* montre un exemple d'appel récursif. Dans le fichier XML 5.3, la fonction de calcul de la première super-étape a le même nom que dans le programme global. De ce fait, lors de la génération/estimation de code hybride, la fonction est remplacée par

la totalité du programme BSP.

Listing 5.5 – le code BSP++ hybride du produit scalaire : Partie MPI

```
1  int main (int argc, char** argv)
2  {
3     par< vector<double> > data;
4     par< double >         result;
5     omp_set_num_threads(4);
6     *result= omp_inner_prod( *in, argc, argv);
7      result_of_proj<double> exch = proj(result);
8     //  step 2
9     *result = Global-sum ( exch.begin() );
10    synch();
11 }
```

Les codes 5.4 et 5.5 ainsi qu'un script d'exécution sont la sortie de notre framework pour l'algorithme produit scalaire dont la description BSP est décrite dans le fichier XML. Ces codes sont générés pour une exécution sur une architecture donnée constituée de deux nœuds chacun avec quatre cœurs et ne contenant aucun accélérateur de type Cell BE. Notez que c'est la même stratégie et les mêmes étapes à suivre pour la génération de code sur des architectures hybrides équipées d'accélérateurs de type Cell BE. En effet, le code généré en sortie est basé sur la bibliothèque `BSP++` qui est, comme souligné auparavant, une bibliothèque multi-plateformes.

5.4 Évaluation de performance

5.4.1 Évaluation du modèle de prédiction

Dans cette section, nous évaluons la précision de notre modèle de communication pour les architecture hybrides. Autrement dit, on évalue la précision des paramètres de communication g et L de chaque niveau de la hiérarchie fournis par les outils utilisés (Sphinx).

La table 5.1 donne la comparaison entre les temps de communication mesurés et prédits pour quatre benchmarks dans leur version hybride. L'architecture cible utilisée dans ces expériences est un cluster de quatre nœuds du site de Bordeaux de la grille de calcul grid5000 [76].

Comme le cluster de Bordeaux est un quadri-cœurs, la version hybride de nos benchmarks utilise quatre threads OpenMP par processus MPI. L'erreur entre le temps estimé et le temps réel varie de 4% à 17% (à l'exception du produit scalaire $inprod$). Ceci montre que le sous-modèle de communication du modèle `BSP++` est suffisamment précis pour nos besoins. Pour le produit scalaire, le gros

TABLE 5.1 – Comparaison entre les temps de communication mesurés et prédits pour différents benchmarks

$Bench$	$CPUs$	$Temps_{Estim}\ sec$	$Temps_{mesu}\ sec$	$Erreur\%$
$InProd$	4	$5\ 10^{-5}$	$5.4\ 10^{-5}$	7
	8	$2.3\ 10^{-4}$	$3.9\ 10^{-4}$	40
	16	$3.4\ 10^{-4}$	$5.2\ 10^{-4}$	34
GMV	4	$2.4\ 10^{-3}$	$2.9\ 10^{-3}$	17
	8	$3.6\ 10^{-3}$	$3.2\ 10^{-3}$	11
	16	$4.2\ 10^{-3}$	$4.6\ 10^{-3}$	8
GMM	4	0.50	0.44	12
	8	1.80	2.03	11
	16	2.6	3.01	13
$PSRS$	4	$2.5\ 10^{-3}$	$2.4\ 10^{-3}$	4
	8	$5.2\ 10^{-3}$	$4.8\ 10^{-3}$	8
	16	$6.1\ 10^{-3}$	$5.5\ 10^{-3}$	10

écart entre le temps de prédiction et le temps d'exécution (jusqu'à 40%) est dû au schéma de communication de ce benchmark. En effet, et comme indiqué à plusieurs reprises, le produit scalaire exhibe un motif de communication trop simple qui se limite à la transmission d'un **seul élément** à tous les autres processeurs.

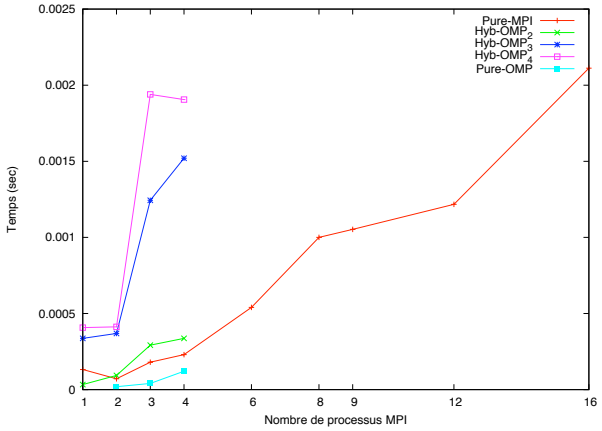
5.4.2 Evaluation des résultats

Dans cette section, nous évaluons les performances de la solution fournie par notre outil `BSPGen` en terme de nombre de processus MPI et de threads OpenMP à chaque étape du programme. En d'autres termes, nous comparons la meilleure solution générée par l'outil et le temps d'exécution de toutes les configurations correctes possibles.

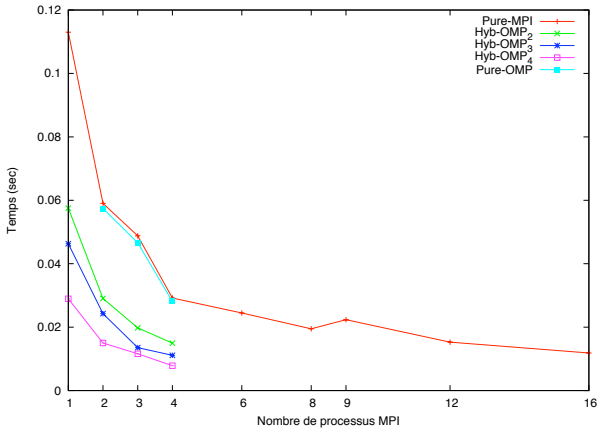
Les résultats étant très semblables, nous discuterons uniquement les performances de deux benchmarks : 1 - le produit scalaire $Inprod$, parce qu'il est le plus simple et qu'il a le même comportement en terme de stratégie d'hybridation et de performances que GMM et GMV. 2 - le tri par échantillon $PSRS$ car il possède un ratio des temps calcul/communication intéressant et plus important car il utilise plusieurs appels récursifs dans sa version hybride.

Pour le produit scalaire et avec une petite taille de données, les performances diminuent quand le nombre de processeurs augmente. Le temps de communication est plus grand que le temps de calcul en raison de la synchronisation. `BSPGen` détermine la meilleure configuration, qui consiste à l'utilisation uniquement d'un seul nœud en mode mémoire partagée (1 nœud, 0 MPI, 2 OMP) tel

que présenté dans la figure 5.3(a) avec un temps prévu de $6,5\times\ 10^{-5}$ secondes et un temps mesuré de $7,1\times\ 10^{-5}$ secondes. Dans toutes les figures, les légendes $Hyb-OMP_i$ représentent les codes hybrides avec un nombre i de threads OpenMP par processus MPI.



(a) taille =16000 éléments



(b) taille =64×10^6 éléments

FIGURE 5.3 – Les temps d'exécution des différentes configurations possibles pour le benchmark Inprod avec deux tailles de données.

Cependant, avec des tableaux très volumineux, comme le temps de calcul est plus grand que le temps de communication et que ce dernier est meilleur avec un modèle hybride, l'outil choisit la solution hybride (4 nœuds, 1 MPI, 4 OMP). La figure 5.3(b) montre que cette configuration est la meilleure. La table 5.2 montre les temps.

Pour le benchmark $PSRS$, la situation est assez différente. En effet, l'algorithme BSP contient quatre super-étapes dont trois utilisent la notion d'appel *ré-*

TABLE 5.2 – Comparaison entre les temps mesurés et prédits pour les meilleures configurations.

$Bench/taille$	$Temps_{Estim}\,(sec)$	$Temps_{mesu}\,(sec)$	$Erreur\ en\%$
$InProd/Petite$	$6.5\ 10^{-5}$	$7.5\ 10^{-5}$	8
$InProd/Grande$	$1.0\ 10^{-2}$	$1.3\ 10^{-2}$	16
$PSRS/Petite$	$4\ 10^{-3}$	$3.6\ 10^{-3}$	10
$PSRS/Grande$	2.46	2.66	7.5

cursif à la fonction ($Trier$) . Tel que discuté précédemment, chaque appel à cette fonction $Trier$ est remplacé par l'ensemble du programme BSP. Cela augmente enormément le temps de calcul. L'avantage du temps de communication du programme hybride ne compense pas l'augmentation du temps de calcul.

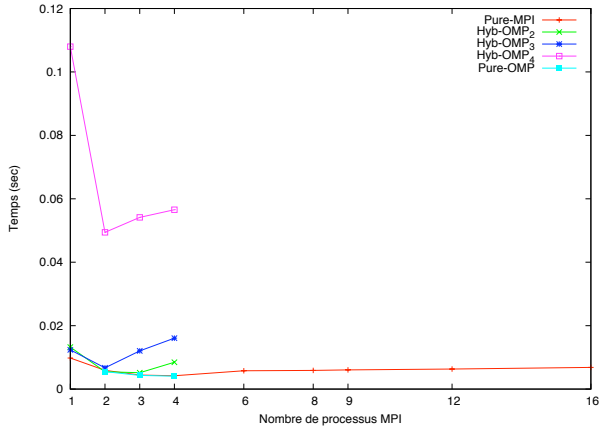
Avec des tableaux de petite taille, la version pure OpenMP avec 4 cœurs est la plus performante comme le montre la figure 5.4(a). Pour une taille de données plus consistante, la configuration choisie par notre outil correspond à celle utilisant tous les nœuds en mode pure MPI, vu que la version hybride ne compense pas sa perte en temps de calcul (voir figure 5.4(b)).

5.5 Conclusion

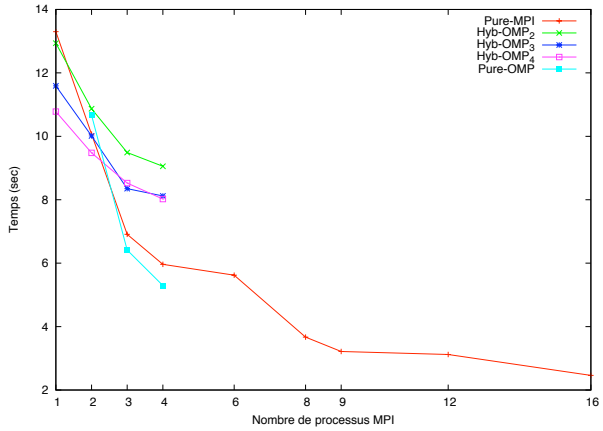
Déterminer le nombre de processeurs à utiliser par une application donnée sur une architecture donnée ainsi que leur mode de programmation est une tâche à la fois critique et importante dans la programmation parallèle. En effet, en plus d'avoir une dépendance directe entre les configurations utilisées et les performances, la conception et la programmation d'un code parallèle dépend également de l'architecture cible.

Dans ce chapitre, nous avons proposé un framework permettant la génération automatique d'un code parallèle ainsi que la détermination des configurations les plus performantes. Notre approche combine une analyse statique réalisée par le module $Analyseur$ et une analyse dynamique à l'exécution réalisée par un système de $profile$. L'analyse statique effectuée par l'*Analyseur* donne l'estimation du temps de calcul, tandis que les benchmarks d'exécution du Sphinx servent à récupérer les informations de la machine cible et de son réseau de communication. Après avoir déterminé les configurations les plus optimales par un module de *Recherche*, le module de *Génération* se charge de la production du code correspondant en utilisant la bibliothèque `BSP++` présentée dans le chapitre précédent.

En plus de cette combinaison, en se basant sur l'aide de l'utilisateur pour la



(a) taille =81920 éléments



(b) taille =8192×10^4 éléments

FIGURE 5.4 – Les temps d'exécution des différentes configurations possibles pour le benchmark PSRS avec deux tailles des données.

description des différents schémas de communication, `BSPGen` est capable de produire un code parallèle hybride à partir d'une liste de fonctions séquentielles constituant sa partie de calcul.

ÉVALUATION DE PERFORMANCES SUR APPLICATIONS 6

Grâce aux travaux présentés dans les deux chapitres précédents, nous disposons désormais d'outils pour faciliter la programmation des machines de calcul haute performance. Nous disposons d'une part d'une bibliothèque générique de haut niveau permettant l'expression des applications sous le modèle BSP indépendamment de l'architecture cible. Nous disposons d'autre part d'un outil pour la génération des codes (purs et hybrides) directement à partir de la description de l'algorithme parallèle.

Dans ce chapitre, nous allons tester la facilité d'utilisation de nos outils ainsi que leurs performances en terme de passage à l'échelle et en comparant les résultats avec des résultats existants pour des codes écrits manuellement par des experts en programmation de l'architecture cible. Ces tests sont effectués en utilisant différentes applications de domaines différents allant de la statistique et de

la vérification jusqu'au traitement d'images et au calcul scientifique en passant par la bioinformatique. Le spectre des ces domaines ainsi que de ces applications est extrêmement vaste : nous avons donc, pour chaque application, donné dans un premier temps une brève description pour faciliter la compréhension de la problématique et ensuite nous avons exposé avec plus de détails les algorithmes parallèles sous-jacents ainsi que leur code `BSP++` et leur performances temporelles. Notez bien que dans ces tests, l'outil `BSPGen` est utilisé uniquement pour générer le code parallèle de base et non pas pour déterminer les configurations les plus optimales.

6.1 Application dans le domaine de la vérification

L'objectif de la vérification du modèle probabiliste est de déterminer la probabilité de satisfaction d'une propriété donnée temporelle dans un système probabiliste. Comme les systèmes probabilistes sont très utilisés dans les modèles et les protocoles de communication, les algorithmes de routage dans les réseaux et algorithmes distribués, etc, il est primordial d'être en mesure de vérifier efficacement leur exactitude. Malheureusement, la plupart des méthodes de model checking ont l'inconvénient d'être soumises au phénomène appelé explosion combinatoire de l'espace des états.

Pour surmonter ces problèmes, [101, 102, 103, 104, 105] ont conçu des approches entièrement différentes qui économisent de la mémoire et du temps tout en vérifiant des grands systèmes. L'idée est d'utiliser des méthodes d'approximation, essentiellement basées sur l'échantillonnage des chemins d'exécution pour la vérification du système probabiliste. En utilisant une méthode d'échantillonnage pour la vérification des système probabilistes, telle que la méthode $APMC$ (**Approximate Probabilistic Model Checking**) décrite dans [101] et [102], il est possible de s'approcher, avec une erreur contrôlée, de la probabilité qu'une propriété temporelle soit vraie dans un système probabiliste.

6.1.1 Description de l'approche approximative dans la vérification

Dans cette application, avec des membres de l'équipe Grand-Large de l'INRIA, nous nous sommes intéressés au problème de vérification quantitative. C'est à dire au problème du calcul de la probabilité qu'un système probabiliste, modélisé comme une chaîne de Markov, satisfasse une formule donnée linéaire en temps. Un des premiers algorithmes pour ce problème a été proposé par Courcoubetis et Yannakakis [106]. L'algorithme transforme étape par étape la chaîne de Markov et la formule, en éliminant un par un les connecteurs temporels, tout en préservant la probabilité de satisfaction de la formule.

L'élimination des connecteurs temporels est effectuée en résolvant un système linéaire d'équations de la taille de la chaîne de Markov. De toute évidence, cet algorithme souffre de problèmes de complexité en taille [101, 102].

Pour de nombreuses propriétés linéaires en temps, la satisfaction par un chemin d'exécution de longueur finie implique la satisfaction pour toute extension de ce chemin. Cette propriété est appelée $monotone$. Dans [102] il est démontré que la probabilité de satisfaction de la monotonie ou anti-monotonie pour une propriété linéaire en temps peut être approchée avec un schéma d'approximation aléatoire. Étant donné un temps discret ou un temps continu de la chaîne de Markov (DTMC ou CTMC) $\mathcal{M}$ et une propriété ψ monotone, nous approchons la probabilité $Prob[\psi]$ de l'ensemble des chemins d'exécution satisfaisant la propriété ψ par un algorithme de point fixe obtenu par itération sur un schéma d'approximation aléatoire pour $Prob_k[\psi]$. $Prob_k[\psi]$ est la mesure de la probabilité associée à l'espace probabiliste des chemins d'exécution de longueur finie k. Nous adaptons la notion de schéma d'approximation aléatoire pour des problèmes de comptage, qui est due à Karp et Luby [107] pour obtenir l'algorithme d'échantillonnage aléatoire suivant $\mathcal{GAA}$. Il utilise le générateur probabiliste G sur $\mathcal{M}$ pour calculer une bonne approximation de $Prob_k[\psi]$.

Algorithme d'Approximation Générique $\mathcal{GAA}$
Entrée : $G, k, \psi, \varepsilon, \delta$
Sortie : ε-approximation de $Prob_k[\psi]$
$N := \ln(\frac{2}{\delta})/2\varepsilon^2$
$A := 0$
Pour $i = 1$ à N faire
Générer un chemin aléatoire σ de taille k
Si ψ est vrai sur σ Alors $A := A + 1$
Return $Y = A/N$

Pour résumer, nous utilisons le générateur probabiliste pour générer des chemins aléatoires et calculer une variable aléatoire Y qui se rapproche de $p = Prob_k[\psi]$. Notre approximation va être correcte car $|Y - p| < \varepsilon$ (erreur additive), avec un degré de confiance égal à $(1 - \delta)$

6.1.2 APMC en pratique

Le model checker APMC implémente l'algorithme décrit ci-dessus (voir par exemple [108]). On a utilisé la version APMC 3.0. Le langage d'entrée de APMC est le même que PRISM* [1]. Les modèles sous la vérification de APMC sont des

1. * http ://www.prismmodelchecker.org/

chaînes de Markov discrètes ou continues en temps. En utilisant APMC, nous calculons une valeur approximative de la probabilité d'une propriété temporelle sur un système probabiliste. Fondamentalement APMC fonctionne en générant des chemins aléatoires dans l'espace probabiliste sous-jacent du système et calcule une variable aléatoire qui estime la probabilité de la formule. Comme indiqué dans [108], nous utilisons à cet effet un *diagram* : une représentation succincte du système. Cette notion de représentation succincte est d'une importance extrême pour APMC, car elle permet l'utilisation d'un espace mémoire très limité.

Le logiciel original APMC se compose de plusieurs éléments indépendants : l'analyseur, la bibliothèque de base et l'outil de déploiement. Ce dernier n'est plus en utilisation depuis que nous avons remplacé les outils de déploiement ad-hoc par une version utilisant `BSP++` pour la génération du code parallèle.

L'analyseur est un simple programme $Lex/yacc$ qui analyse et parse le code PRISM et les formules LTL . Il appelle ensuite la bibliothèque de base d'APMC pour produire une représentation interne réduite du modèle (linéaire en fonction de la taille du fichier de modèle) et des propriétés (linéaires en fonction de la taille du fichier de propriété). La bibliothèque produit alors un générateur et un vérificateur en code C (le générateur et le vérificateur sont des fonctions séquentielles autonomes écrites en C). La boucle principale du code produit par la bibliothèque de base d'APMC consiste à générer un chemin (c'est à dire un ensemble de configurations) de longueur donnée et l'évaluation des propriétés (formules) sur chaque chemin. Le nombre d'itérations de cette boucle est un paramètre de notre programme. La sortie du programme est le nombre de chemins générés, le nombre de chemins où la formule est vraie, et le nombre de chemins où la formule est fausse. C'est pourquoi il semble facile d'utiliser des architectures hautement parallèles pour l'exécution simultané de plusieurs programmes (générateur et vérificateur) et récolter leurs résultats.

La parallélisation d'APMC avec `BSP++` consiste essentiellement à générer et à vérifier les chemins de façon indépendante. Cette approche est également utilisée par PRISM pour la distribution du simulateur (voir par exemple [109]). Nous expliquons en détail dans la section suivante comment on a conçu et implémenté la version `BSP++` d'APMC.

6.1.3 APMC avec BSP++

Comme souligné précédemment, l'algorithme BSP pour APMC est constitué d'une seule super-étape à l'intérieur d'une boucle qui itère sur le nombre de chemins satisfaisants la formule. La super-étape en elle même est composée de l'appel aux deux fonctions séquentielles de génération et de vérification des chemins suivi de l'étape de communication pour collecter le nombre de chemins qui satis-

font la formule. La fonction génératrice produit des chemins aléatoires de taille k (passé comme paramètre) et en utilisant ces chemins, la fonction de vérification évalue la satisfaction de la formule. Le code pur (MPI ou OpenMP ou Cell) produit par le générateur de `BSPGen` est illustré par le listing 6.1.

Listing 6.1 – le code BSP++ d'APMC avec la version pure

```
1  int main (int argc, char** argv)
2  {
3     par< vector<int> > ParArg;
4     ParArg=vector<int>(3);
5     int rank=pid();
6     if(rank==0)
7     {// lire les arguments et les stoker dans
8      (*ParArg)[0]= strtol(argv[0]);
9      ...
10    }
11   // transfert de ces paramètres pour tous les processus
12       en appelant la primitive put
13
14    par<pair<vector<int>,vector<int> > >  PAR;
15    PAR=make_pair(vector<int>(NB_VERIF), vector<int>(NB_VERIF) );
16
17   // début de noyau de code
18   for(;;)
19    {
20      Generate_path(rank,(*ParArg)[0],(*ParArg)[2], &path);
21      Verify_path(&path, (*PAR).first[0],(*PAR).second[0]);
22      // partie communication de super-étape
23
24      par<boost::fonction<pair<vector<int>, vector<int> > (int)> > s
25           = bind(f_com,bl::_1,PAR);
26      result_of::put<boost::function<pair<vector<int>,
27           vector<int> > (int) > >::type recv;
28      recv= put(s);
29
30      NB_TRUE[p]=accumulate((*recv).first[p],0);
31      NB_FALSE[p]=accumulate((*recv).second[p],0);
32
33      //pour toutes les propriétés, on fait la somme
34      for(p=0;p<NB_VERIFY_GOAL;p++)
35      max=NB_TRUE[p]+NB_FALSE[p];
36
37      if(max>*ParArg[1]) break;
38    }
39 }
```

Dans le listing 6.1, le cœur du code exécute une boucle infinie dont chaque itération consiste à exécuter la super-étape de notre algorithme BSP. Les deux fonctions $Generate\_path$ et $Verify\_path$ correspondent aux fonctions séquentielles en C générées par la bibliothèque de base du logiciel APMC. Après l'échange des résultats partiels entre les processeurs, on teste leur somme avec le nombre de chemins à trouver passé comme argument à notre application. En cas de succès, on quitte la boucle et on termine le code.

Notez bien que ce code correspond aux versions pures MPI et OpenMP et c'est exactement le même code pour la version pure Cell BE. Autrement dit, dans

la version Cell BE, il n'y a pas l'utilisation des $itrateurs$ pour les transferts des données entre la mémoire principale (RAM) et les mémoires locales des SPE. En effet, comme les chemins sont générés aléatoirement par chaque SPE, il n' y a pas de transferts de données entre PPE et SPEs.

Les versions hybrides générées par `BSPGen` se basent sur la stratégie d'hybridation de `BSP++`. Cependant, dans cette algorithme, comme il n'y a pas de transferts de données entre les deux niveaux de l'hybridation (sauf le retour des nombres des chemins corrects et faux), les codes générés ne contiennent pas de fonctions $split$.

6.1.4 Expérimentations

Nos expériences consistent en la vérification de propriétés temporelles sur deux modèles différents. Le premier est le modèle du dîner des philosophes [110], pour lequel on vérifie la propriété d'atteignabilité (*c-à-d* la propriété de la forme $true\ U\ \phi$ pour certains ϕ de premier ordre). Etant très bien connu, ce modèle nous permet de nous assurer qu'il n'y a pas de comportement étrange dans le processus de vérification. Le paramètre de ce modèle est le nombre de philosophes qui interagissent ensemble. Le second est le modèle d'un réseau de capteurs (Sensors Network [111]). Pour ce deuxième modèle, la même propriété d'atteignabilité est vérifiée. Le paramètre de ce modèle est la taille de la grille de la communication, ce qui signifie que le modèle $SN(\ X\ )$ contient X capteurs. Les deux modèles ont une représentation explicite : la taille est exponentielle par rapport aux paramètres X(nombre de philosophes ou de capteurs). La taille des chemins est prise comme argument pour les deux modèles. Pour le premier modèle, la taille des chemins est de 900 alors qu'elle est de 8000 pour le second. Les deux modèles ont été exécutés avec différentes tailles de problème.

Pour faciliter la compréhension, les résultats sont présentés en utilisant la métrique de *ralentissement*. Le ralentissement d'une exécution sur une machine donnée est défini comme le temps d'exécution en utilisant *n-cœurs* multiplié par le nombre de cœurs n. En utilisant cette métrique, le temps d'exécution global pour *n-cœurs* reste constant lorsqu'on a une accélération linéaire. Lorsque l'efficacité parallèle décroît (ralentissement) tandis que le nombre de cœurs augmente, le temps global d'exécution augmente. Avec une accélération super-linéaire, le temps d'exécution global diminue lorsque le nombre de cœurs augmente.

Nos expériences ont été effectuées sur les trois plateformes décrites dans le chapitre 4. Pour rappel, **AMD-Machine** est une machine de 16 cœurs à mémoire partagée. **Cluster-Machine** est un cluster du site de Bordeaux de la grille grid5000. Dans ces expériences, on a utilisé 64 nœuds pour un total de 256 cœurs. **Cell-Machine** est un petit cluster de trois Cell BE connecté par un réseau de 100

Mbit/s.

6.1.4.1 Performance des versions MPI

Les figures 6.1(a), 6.1(b) , 6.2(a) et 6.2(b) montrent le ralentissement de la version MPI de APMC avec le modèle de dîner des philosophes et le modèle de réseaux de capteurs sur la **AMD-Machine** et **Cluster-Machine** respectivement. Plus précisément, ces résultats montrent le temps moyen (en secondes, moyenne, parmi environ 120000 chemins) utilisés pour la vérification de la formule sur un chemin par rapport au nombre de cœurs utilisés pour la vérification.

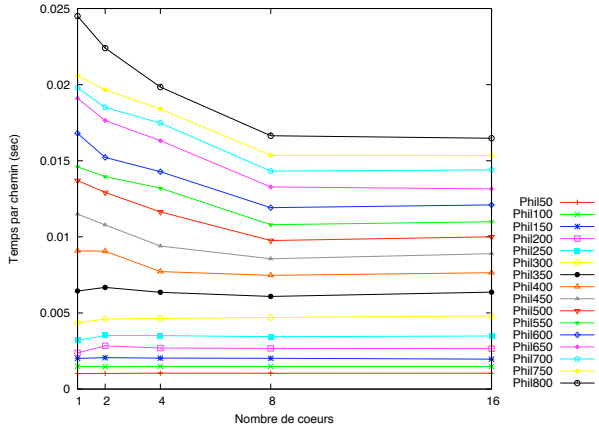
Sur la **AMD-Machine**, nous obtenons une accélération super-linéaire de 1 à 8 cœurs et une accélération linéaire jusqu'à 16 cœurs. Cela signifie que la parallélisation passe à l'échelle parfaitement : il n'y a pas surcoût lié à l'utilisation de `BSP++`. L'accélération super-linéaire est due au fait qu'on calcule un temps moyen sur la totalité des chemins. Sur le **Cluster-Machine**, les chiffres montrent une accélération linéaire jusqu'à 128 cœurs et un petit ralentissement de 128 à 256 cœurs. Le ralentissement est dû au temps de synchronisation qui augmente avec le nombre de cœurs.

6.1.4.2 Performance de la version OpenMP

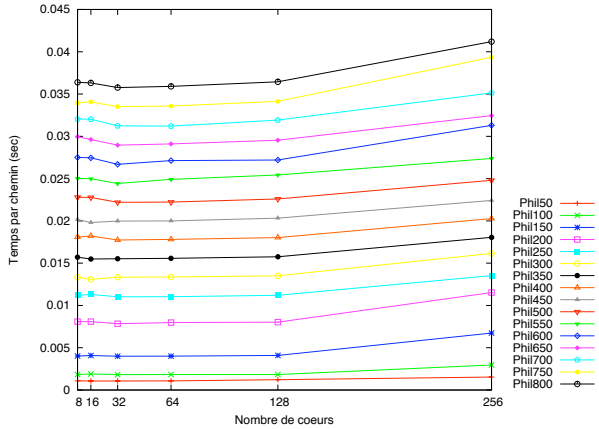
La version OpenMP d'APMC conduit à des ralentissements différents pour les deux modèles. Ces résultats sont présentés dans les figures 6.3(a) et 6.3(b). Les résultats montrent une accélération super-linéaire pour le modèle du dîner des philosophes et une accélération linéaire pour le modèle de réseau de capteurs. Par rapport à la version MPI, il n'y a aucune différence entre le temps d'exécution sur 1 à 8 cœurs, car les deux expériences (MPI et OpenMP) sont menées sur une architecture à mémoire partagée. Sur 16 cœurs, la version OpenMP obtient un petit avantage car la synchronisation et la communication sont plus rapides en OpenMP qu'en MPI.

6.1.4.3 Performance de la version Cell

Comme la taille du code source pour les deux modèles augmente avec la taille du modèle, et les mémoires locales des SPEs sont de 256 Ko, nous avons été en mesure d'exécuter seulement une petite série de modèles sur le processeur Cell : la taille des modèles sont de 5 et 6 philosophes et 4 et 9 capteurs pour les SN en utilisant des chemins de tailles réduites (La taille du chemin est respectivement de 28 et 69 pour les philosophes et les réseaux de capteurs).



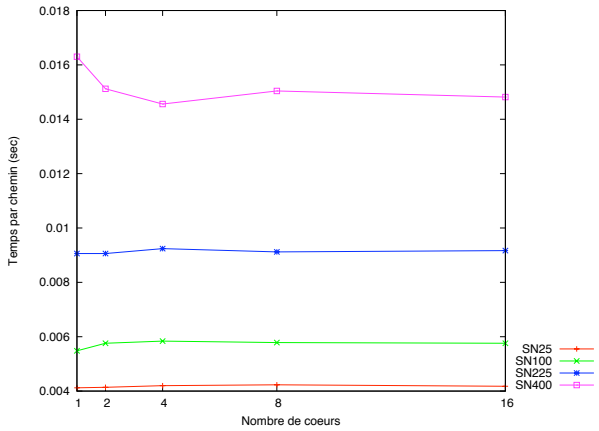
(a) AMD-Machine



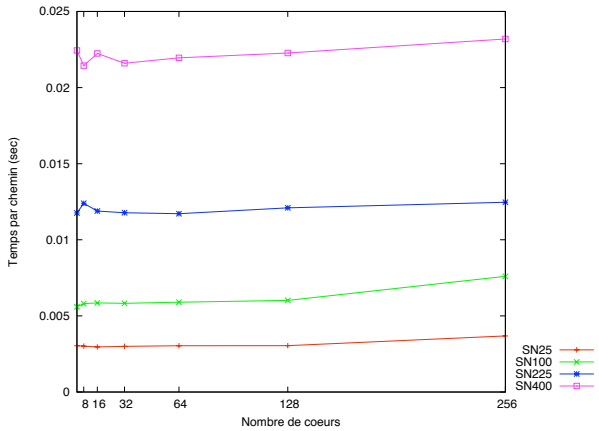
(b) Cluster-Machine

FIGURE 6.1 – Ralentissement de la version MPI - problème du dîner des philosophes.

Les figures 6.4(a) et 6.4(b) représentent la comparaison entre les temps d'exécution sur le processeur Cell et la **AMD-Machine** avec la version OpenMP pour les deux modèles. Les résultats montrent que la parallélisation sur le Cell passe à l'échelle parfaitement, mais les performances sont très médiocres par rapport à un processeur multi-cœurs normal (jusqu'à 17 fois plus lent que la version OpenMP). Le passage à l'échelle prouve qu'il n'y a pas de sur-coût lié à l'utilisation de `BSP++`. Les performances médiocres sur le Cell sont dues à l'architecture de ce dernier. En effet, les opérations d'un code APMC sont de type $contrle$. Ce type d'opérations ne peut pas être vectorisé facilement et efficacement, alors que les performances sur le processeur Cell exigent une telle vectorisation [112].



(a) AMD-Machine



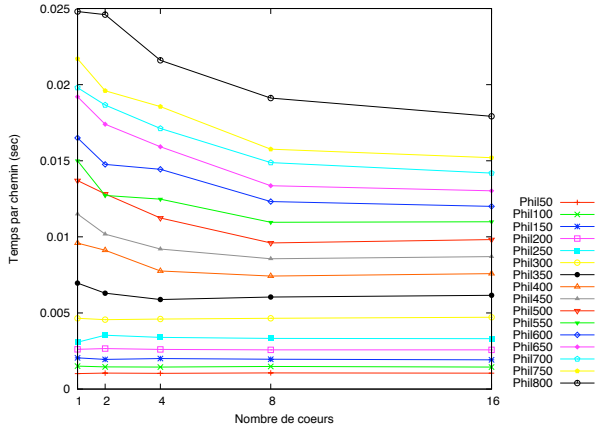
(b) Cluster-Machine

FIGURE 6.2 – Ralentissement de la version MPI - problème du réseau de capteurs.

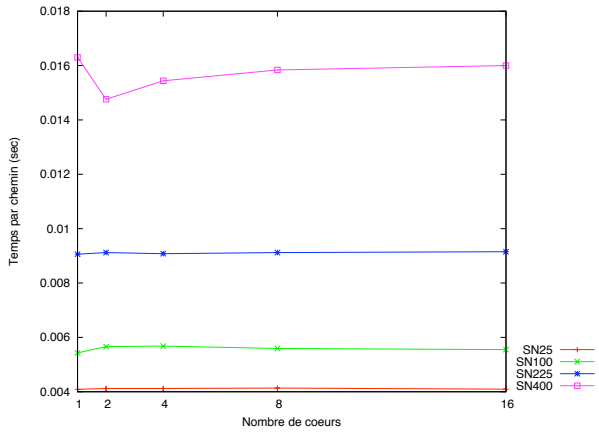
6.1.4.4 Performances des versions hybrides

Comme les nœuds de **Cluster-Machine** sont des bi-processeurs bi-cœurs, chaque nœud (processus MPI) lance quatre threads OMP. Par exemple, sur les 256 cœurs, il y a 64 MPI $\times$ 4 threads OpenMP. Comparée à la version MPI, la version hybride MPI/OpenMP obtient de meilleures performances. La réduction du temps de synchronisation en utilisant une synchronisation à deux niveaux est le point clé de cette amélioration.

Le gain de la version hybride est de 50% par rapport à la version MPI pour le modèle des philosophes, avec 50 philosophes et de 10% pour 800 philosophes . L'explication de la diminution du gain est la suivante : lorsque le nombre de philosophes est petit, le ratio communication/calcul est grand (parce que le nombre



(a) Problème du dîner des philosophes



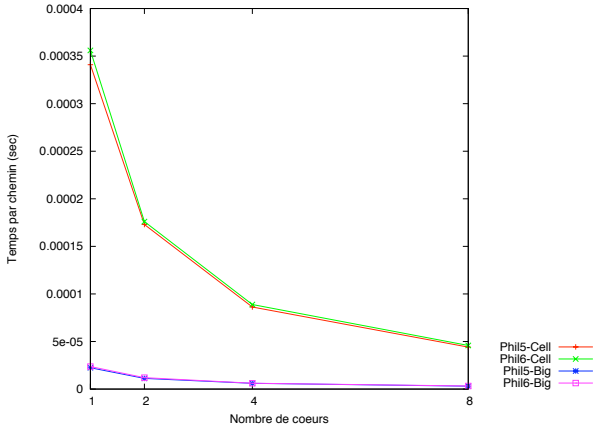
(b) Problème du réseau de capteurs

FIGURE 6.3 – Ralentissement des versions OpenMP sur AMD-Machine pour les deux modèles.

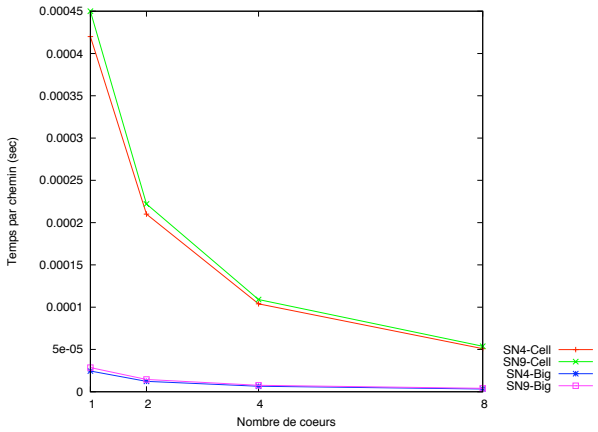
de modules est exactement le nombre de philosophes), ce qui donne un avantage pour la version hybride. Au contraire, lorsque le nombre de philosophes est élevé, la partie calcul domine la partie communication, ce qui diminue l'avantage de l'utilisation d'OpenMP pour la communication.

Les résultats de ralentissement dans les figures tracées par 6.5(a) et 6.5(b) montrent que la version hybride MPI+OpenMP obtient une accélération linéaire jusqu'à 256 cœurs.

Pour la version hybride MPI+Cell, même si les performances sur un seul Cell sont médiocres, le passage à l'échelle avec MPI (utilisation de 3 processus MPI lancés chacun sur un processeur Cell) est bien évidemment parfaitement assuré.



(a) Problème de diner des philosophes



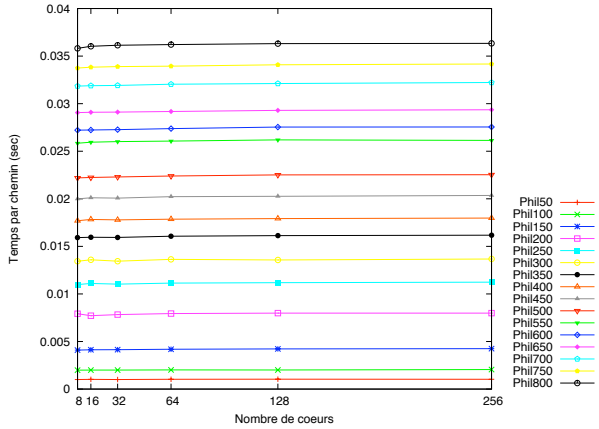
(b) Problème de réseau de capteurs

FIGURE 6.4 – Comparaison des temps d'exécution entre les versions Cell et OpenMP.

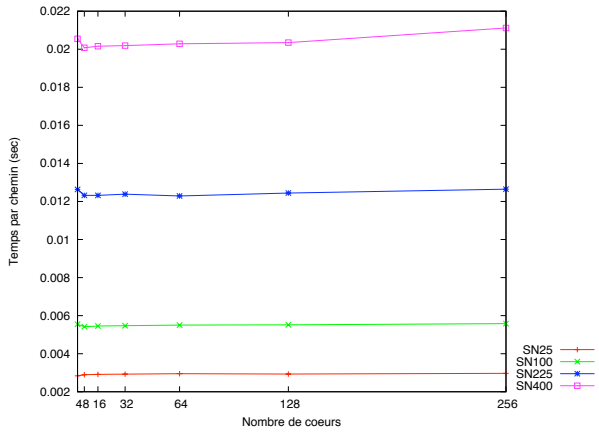
On a une accélération proche de 3 sur la **Cell-Machine**.

6.2 Application dans le domaine de la bioinformatique

Pour notre deuxième application, nous nous sommes intéressé au domaine de la bioinformatique qui est un domaine de recherche interdisciplinaire qui implique l'informatique, la biologie, les mathématiques et les statistiques [113]. Son principal objectif est d'analyser les données de séquences biologiques et le contenu du génome afin d'obtenir la structure fonctionnelle de celles-ci ainsi que des informations évolutives.



(a) Problème du dîner des philosophes



(b) Problème du réseau de capteurs

FIGURE 6.5 – Ralentissement des versions hybrides MPI+OpenMP sur Cluster-Machine pour les deux modèles.

Une fois une nouvelle séquence biologique découverte, ses caractéristiques fonctionnelles et structurelles doivent être établies. Pour ce faire, la nouvelle séquence découverte est comparée avec toutes les séquences qui composent la base de données biologique, à la recherche de similitudes. La comparaison des séquences est, par conséquent, l'opération la plus élémentaire dans le domaine de bioinformatique. L'algorithme le plus précis pour exécuter les comparaisons par paires est celui proposé par $Smith$ et $Waterman$ (SW) [114], qui est basé sur la programmation dynamique, avec une complexité **quadratique** en temps et en espace. Ceci peut facilement conduire à des temps d'exécution extrêmement élevés et à des besoins en mémoire énormes, puisque la base de données biologique est en croissance exponentielle et les séquences biologiques peuvent être extrêmement longues. Le calcul parallèle peut être utilisé pour produire des résultats plus rapidement, en réduisant considérablement le temps d'obtention des résul-

tats avec l'algorithme SW.

6.2.1 Comparaison des ADNs

Une séquence biologique est une molécule d'acides nucléiques ou de protéines. Elle est représenté par une liste ordonnée de résidus, qui sont à base de nucléotide (pour les séquences d'ADN ou d'ARN) ou d'acides aminés (pour les séquences de protéines). Une séquences d'ADN est traitée comme des chaînes composées par des éléments de l'alphabet $\sum = \{A, T, G, C\}$. Comme deux séquences d'ADN sont rarement identiques, la comparaison de celles-ci est en fait un problème de filtrage approximatif [113]. Pour comparer deux séquences, nous avons besoin de trouver le meilleur alignement entre elles, ce qui revient à placer une séquence au-dessus de l'autre et chercher la correspondance entre les caractères similaires [113].

Étant donné un alignement entre deux séquences s et t, un score lui est associé comme suit : pour chaque colonne, on associe (a) un gain *ma* si les deux caractères sont identiques (*match*) ; ou (b) une pénalité *mi* si les caractères sont différents (*mismatch*) ; ou (c) une pénalité *g*, si l'un des caractère est un espace (*gap*). Le score est l'addition de toutes ces valeurs. Le score maximal est appelé la $similitude$ entre les séquences. La figure 6.6 présente un alignement possible entre deux séquences d'ADN et son score associé. Dans cette figure, $ma = +1$, $mi = -1$ et $g = -2$.

A	C	T	T	G	T	C	C	G
A	$-$	T	T	G	T	C	A	G
$+1$	-2	$+1$	$+1$	$+1$	$+1$	$+1$	-1	$+1$

$score = 4$

FIGURE 6.6 – Exemple d'alignement et de score.

6.2.2 Description de l'algorithme SW

L'algorithme SW [114] est une méthode exacte basée sur la programmation dynamique pour obtenir le meilleur alignement pour une paire de séquences. L'algorithme est quadratique en temps et en espace et il est composé de deux phases : la création de la matrice de similarité et l'obtention de l'alignement.

La première phase de l'algorithme SW reçoit comme entrée les séquences s et t et construit la matrice de similarité. Cette matrice est noté $A_{m+1,n+1}$, où $A_{i,j}$ contient le score entre le préfixe des séquences $s[1..i]$ et $t[1..j]$. Au début, la première ligne et la première colonne de la matrice sont remplies avec des zéros. Les

	*	G	A	A	G	C	T	A
*	0	0	0	**0**	0	0	0	0
G	0	1	0	0	**1**	0	0	0
C	0	0	0	0	0	**2**	0	0
T	0	0	0	0	0	0	**3**	1
G	0	1	0	0	1	0	1	2
A	0	0	2	1	0	0	0	2
C	0	0	0	0	0	1	0	0
C	0	0	0	0	0	1	0	0
T	0	0	0	0	0	0	2	0

FIGURE 6.7 – Matrice de similarité pour les séquences s et t.

éléments restants de A sont obtenus à partir de l'équation6.1.

$$A_{i,j} = \max \begin{cases} A_{i-1,j-1} + (\text{if } s[i] = s[j] \text{ then } ma \text{ else } mi) \\ A_{i,j-1} - \text{g} \\ A_{i-1,j} - \text{g} \\ 0 \end{cases} \tag{6.1}$$

La phase 2 est exécutée pour obtenir le meilleur alignement. L'algorithme commence à partir de la cellule qui a la plus grande valeur dans $A_{i,j}$, en suivant les flèches jusqu'à ce que la valeur zéro soit atteinte. La figure 6.7 présente la matrice de similarité pour l'obtention d'un alignement entre deux séquences, avec un $score = 3$.

Dans l'algorithme SW, la plupart du temps est consacré au calcul de la matrice A (première phase) et c'est la partie qui est habituellement parallélisée. Le modèle d'accès présenté par le calcul matriciel est non uniforme et la stratégie de parallélisation qui est traditionnellement utilisée est la méthode du front d'onde (wavefront) [115], car les calculs qui peuvent être effectuées en parallèle évoluent à mesure que la vague évolue sur la diagonale.

La Figure 6.8 illustre la méthode du front d'onde en utilisant une technique de partition par colonne pour quatre processeurs. Au début, seulement le processeur $P1$ effectue les calculs (Figure 6.8.a). Lorsque $P1$ termine le calcul des valeurs d'un bloc, il envoie sa colonne frontière pour $P2$, qui peut commencer à calculer (Figure 6.8.b). Dans la Figure 6.8.c, le parallélisme maximal est atteint.

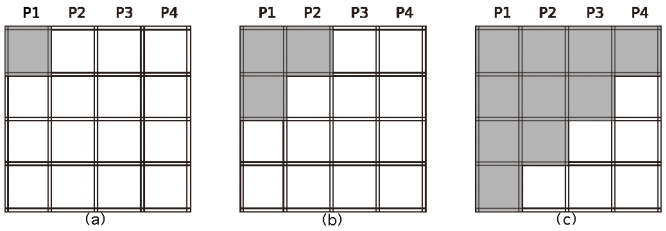


FIGURE 6.8 – La méthode de la vague (wavefront).

6.2.3 Portage de SW avec BSP++

6.2.3.1 Algorithme parallèle de SW

Dans une collaboration avec des chercheurs de université de Brasilia, nous avons développé un code parallèle SW [116] qui calcule le score optimal avec des pénalités, selon l'équation (6.1). Étant donné deux séquences d'ADN d'entrée s et t, l'algorithme parallèle calcule la matrice A en utilisant la méthode du front d'onde avec une approche par colonnes. Chaque processeur Pi calcule un sous-ensemble de colonnes de A (un bloc) et communique avec son processeur voisin de droite quand il termine le traitement d'un bloc.

La figure 6.9 illustre la répartition du travail entre les processeurs P. Dans cette figure, nous avons utilisé quatre processeurs, chacun calcule $M/4$ colonnes. Les séquences s et t sont placées dans l'axe vertical et horizontal, respectivement. Au début, le processeur $P0$ commence le calcul et les autres processeurs restent inactifs puisque les dépendances de données doivent être respectées. Lorsque $P0$ termine le calcul du son premier bloc, il envoie sa colonne frontière à $P1$, qui peut commencer à calculer à son tour. Dans le même temps, $P0$ commence le calcul de son second bloc. Lorsque $P1$ termine le calcul de son premier bloc, il envoie sa colonne frontière à $P2$, qui peut commencer le calcul. $P1$ commencera le traitement de son second bloc car il a reçu la colonne de deuxième bord du $P0$, et ainsi de suite. Dans notre version parallèle de base, chaque processeur Pi stocke la séquence entière s et une partie de la séquence t. Et en outre, il alloue un bloc de taille $Ha \times Hb$ (figure 6.9).

Afin de comparer les séquences longues, nous avons utilisé une version avec des blocs cycliques, où le traitement est divisé en plusieurs partitions, et chaque partition est traitée comme dans la figure 6.9. Par exemple, s'il y a 4 partitions, chaque processus va calculer P blocs dans la partition 1, puis P blocs dans la partition 2 et ainsi de suite, de manière cyclique.

Il y a aussi un paramètre x qui spécifie le nombre de lignes qui seront calculées à l'intérieur de chaque bloc à chaque itération. Fixer une valeur faible pour x aug-

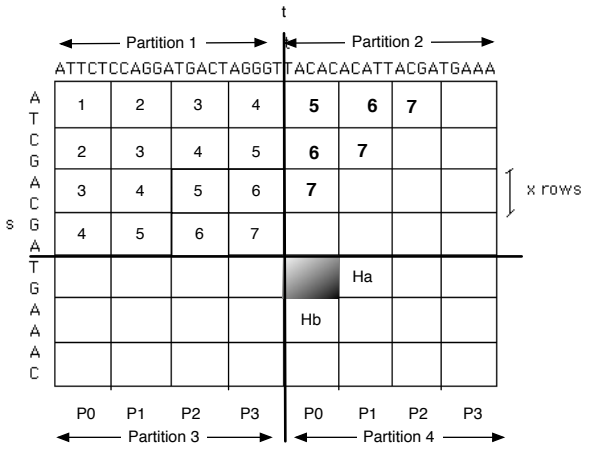


FIGURE 6.9 – Le schéma d'exécution parallèle de SW avec 4 partitions et 4 processeurs (P0 - P3). Les numéros indiquent l'antidiagonale qui s'exécute.

mente potentiellement le parallélisme, mais augmente également le nombre de messages entre les nœuds. Des valeurs énormes provoqueraient l'effet contraire.

6.2.3.2 SW avec BSP++

L'algorithme BSP développé pour le programme parallèle SW de la section précédente contient une seule super-étape dans une boucle qui itère sur la séquence s. A l'itération i, le processeur j calcule la matrice de similarité locale en utilisant les blocs des séquences t, s et la colonne frontière reçue du processeur *j-1* à l'itération *i-1*. Le listing 6.2 montre la fonction `Comm_fun` , qui implémente le schéma de communication utilisé par la primitive `put` de `BSP++`. Dans ce schéma, chaque processeur p, sauf le dernier, envoie sa colonne frontière de droite au processeur `p+1`.

Listing 6.2 – La fonction du communication de SW

```
1 vector<double> Comm_fun(int dest, vector<double> &val, int pid, int nbProc)
2 { if (pid!= nbProc-1)
3       if(dest==pid+1) return val;
4  return vector<double>();
5 }
```

Le listing 6.3 montre la boucle principale du programme `BSP++` de SW. La fonction $SW()$ est la fonction séquentielle qui calcule un bloc de taille $Ha \times Hb$ (figure 6.9). Tous les processeurs sauf (pid = 0) utilisent les valeurs reçues du voisin de gauche à l'itération précédente. A la fin de la super-étape (à chaque itération), tous les processeurs communiquent en appelant *Comm_fun* en utilisant la primitive `put` de `BSP++`. Ce code est utilisé pour générer les versions mono-

architecture à savoir : MPI, OpenMP et Cell BE. Cependant, la faible quantité d'espace mémoire disponible sur chaque SPE (uniquement 256 Ko) impose une contrainte sévère sur la taille des séquences à comparer avec la version Cell de `BSP++`.

Listing 6.3 – Le code BSP++ de la boucle principale de SW pour les versions MPI, OpenMP et Cell

```
#include <bsppp/bsppp.hpp>
int bsp_main(int argc, char** argv)
{
 result_of::put<boost::function<vector<double> (int) > >::type recv;
 for(i=0; i< iterations; i++)
  {
   if(pid()==0) SW(data,border_column_s,border_row_t,&next_s,seq_s,seq_t);
   else SW(data, recv(pid()-1),border_row_t,&next_s,seq_s,seq_t);
   memcopy(border_row_t, data[val]) ;
    // communication part; call to the Comm_fun
    par<boost::function<vector<double> (int)>> msg= Comm_fun(ALL,next_s, pid(),
          size() );
    recv=put(msg);
  }
}
```

Notez que le problème d'avoir une zone de stockage qui est plus petite que la taille nécessaire pour le traitement d'un bloc est un problème générique, qui peut se produire dans de nombreuses plateformes, telles que CellBEs ou GPU. Pour s'attaquer à ce problème, nous avons utilisé la solution de partition par blocs cycliques décrite précédemment. Le code généré par `BSPGen` pour la version Cell utilise la notion d'itérateur et de la boucle de $split$. Dans ce code, les séquences sont stockées dans la mémoire de PPE et les SPE itèrent sur les partitions cycliquement. (Voir le listing 6.4).

Pour les versions hybrides, les codes générés utilisent bien évidemment la stratégie d'hybridation décrite dans les chapitres 4 et 5. Les codes conçus ont une hiérarchie à deux niveaux qui correspond directement aux architectures cibles, avec une communication inter-nœud au niveau 1 et intra-nœud au niveau 2. Au niveau 1, le calcul de la matrice de similarité est divisé entre les P nœuds qui composent l'architecture et chaque nœud calcule $(n \times m)/p$ cellules, où n et m sont les tailles des séquences t et s. A l'intérieur de chaque nœud no_i, le calcul est encore divisé en c_i parties (blocs), où c_i est le nombre total de cœurs/SPE à l'intérieur d'un nœud no_i.

En utilisant exactement la même idée d'hybridation, le code généré pour la version MPI+Cell utilise le code 6.4 dans le deuxième niveau compilé avec l'option $-DBSP\_CELL\_TARGET$. Dans cette version, un Kernel est appelé à chaque itération dans la boucle principale du code MPI. Nous avons également introduit des copies de la ligne et colonne frontières de/vers la mémoire d'accélérateur puisque, avec l'architecture de CellBEs actuelle, le contenu de la mémoire

de l'accélérateur n'est pas conservé entre chaque appel du Kernel (scratchpad memory) [116]. Pour mesurer l'impact de cette version, nous l'avons exécuté avec des séquences de tailles 1 KBP et 10 KBP (voir le tableau 6.1) sur la plate-forme CellBE (section 6.2.4). Pour ces séquences, un surcoût énorme a été introduit par les appels du kernel et les copies multiples (94 % et 93 %, respectivement).

Listing 6.4 – Le code BSP++ de la boucle principale de SW avec la version Cell pour des séquences longues, en utilisant la technique de partitionnement.

```
1  int bsp_main(int argc, char** argv, address_seq_s, address_seq_t)
2  {
3  remote_iterator_input<char> s_it=remote_iterator_input<char>(2,Part_size_s,
4            vector_slicer(Part_size_s));
5  remote_iterator_input<char> t_it=remote_iterator_input<char>(2,Part_size_t,
6            vector_slicer_cyclic(Part_size_t, nb_bloc));
7   result_of::put<boost::function<vector<double> (int) > >::type recv;
8  s_it=address_seq_s;
9  t_it=address_seq_t;
10 for(i=0; i< iterations; i++)
11 {    // use the current buffer
12     seq_s.base=(*s_it).begin();
13      for (j=0, j<nb_bloc,j++)
14     {
15       seq_t.base= (*t_it).begin();
16      if(pid()==0) SW(data,border_column_s,border_row_t,&next_s, seq_s,seq_t);
17      else SW(data, recv(pid()-1),border_row_t,&next_s,seq_s,seq_t);
18      memcopy(border_row_t, data[val]) ;
19      // communication part; call to the Comm_fun
20      par<boost::function<vector<double> (int)>> msg= Comm_fun(ALL,next_s, pid(),
21            size() );
22      recv=put(msg);
23      t_it++; //ask for the next block
24     }
25     s_it++;
26  }
27 }
```

En raison de cette surcharge énorme, nous avons conçu manuellement en utilisant `BSP++` (cette version n'est pas générée par `BSPGen`) une nouvelle version hybride *MPI+CellBE*. Cette nouvelle stratégie utilise une synchronisation **légère** avec un appel unique et *Asynchrone* au Kernel de l'accélérateur dans la boucle principale du niveau 1 (Listing 6.6). Cet appel unique a les avantages suivants : a) l'utilisation du même code de base de la version *Pure Cell* pour le niveau 2 de la construction hybride et b) de bien meilleures performances car il n'y a pas de surcharge pour le démarrage/l'arrêt du kernel et le nombre de copies des colonnes frontières est très faible. La différence entre la version synchrone (semblable à celle de MPI+OpenMP) et la version avec une synchronisation légère (proposée spécifiquement pour la version multi-Cell) est illustrée par la figure 6.10.

Néanmoins, cette nouvelle version synchrone-légère introduit une synchronisation entre le niveau 1 et niveau 2 en raison de la structure du front d'onde dans chaque niveau. Cette synchronisation est définie comme suit : Le SPE pid 0

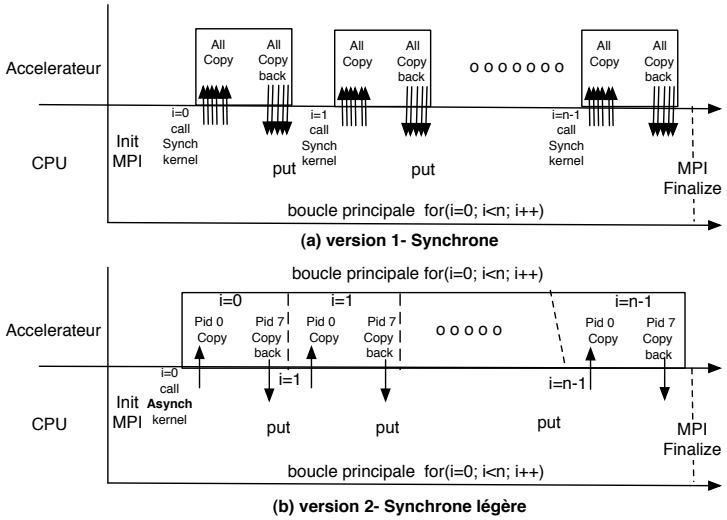


FIGURE 6.10 – Les deux stratégies d'hybridation *BSP++MPI+CellBE* : la version synchrone (a) et synchrone-légère (b).

attend une notification du PPE pour commencer le calcul (lignes 10 et 6 dans les listing 6.5 et 6.6). Quand le PPE aura le contenu correct de $block\_col\_s$ reçu de son voisin MPI de gauche, il notifie ce SPE. En outre, le PPE attend une notification du dernier SPE pour envoyer le contenu de $bloc\_col\_s$ à son voisin MPI de droite (lignes 18 et 12 dans les listings 6.5 et 6.6). La figure 6.11 illustre le schéma de communication et de synchronisation entre les deux niveaux pour cette nouvelle version du code hybride MPI+Cell.

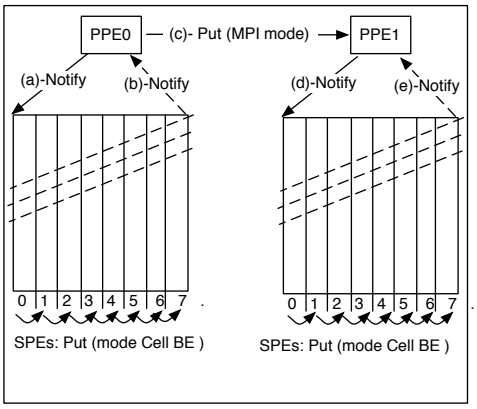


FIGURE 6.11 – Schéma de communication de la version synchrone légère Multi-Cell.

Listing 6.5 – La fonction Async (synchrone légère) dans la version BSP++MPI+Cell

```
int Async_Function(argc, argv, vector <T> &border_col_s, vector <T> &
        next_border_col_s, vector <T1> &seq_s, vector <T1> &seq_t)
{ // declaration of iterators
for(i=0; i< iterations; i++)
 {  seq_s.base=(*s_it).begin();
    for (j=0; j<nb_bloc;j++)
    { seq_t.base= (*t_it).begin();
     // waiting for a notification from level 1
     if(pid()==0){
      if(j==0) {
        message_0=wait_notify(PPE);
        Ait=border_col_s.begin();}
    SW(data, Ait, border_row_t,&nxt_border_col_s_level2, seq_s, seq_t);}
    else
   SW(data, recv(pid()-1), border_col_s,&next_border_col_s_level2, seq_s, seq_t);
    memcopy(border_row_t, data[val]) ;
    if(pid()==size()-1)&&(j==nb_bloc-1)
    { // notify level 1
       nextAit=next_border_col_s.begin(); nextAit++;
       notify(PPE,message);}
    par<boost::function<vector<double> (int)>> msg= Comm_fun(ALL,
             next_border_col_s_level2, pid(), size() );
    recv=put(msg);
    t_it++;
   }
  s_it++;
 }
}
```

Listing 6.6 – Partie MPI de BSP++MPI+Cell

```
#include <bsppp/bsppp.hpp>
int bsp_main(int argc, char** argv)
{ result_of::put<boost::function<vector<double> (int) > >::type recv;
 for(i=0; i< iterations; i++){
      // notify the SPE 0
       notify(pid(0),message_0);
   if(pid()==0)&&(i==0){ // unique asynchronous call
   Async_Function(argc, argv, border_col_s,&next_border_col_s, seq_s,seq_t); }
   else if (i==0){ // unique asynchronous call
   Async_Function(argc, argv, recv(pid()-1),&next_border_col_s, seq_s, seq_t);}
    // Waiting for a notification from the last SPE
  wait_notify(pid(last_SPE),message_last_SPE);
  par<boost::function<vector<double> (int)>> msg= Comm_fun(ALL,next_border_col_s,
           pid(), size() );
     recv=put(msg);
  }
}
```

6.2.4 Expérimentations

En plus des trois plateformes et protocoles d'expérimentation déjà présentés, nous avons utilisé une autre plateforme pour aller plus loin dans l'évaluation de l'efficacité et du passage à l'échelle de `BSP++`. En effet, cette dernière plateforme montre l'adéquation de `BSP++` avec les architectures *Petascale*.

Le **Hopper-Machine** est le premier cluster *petaflops* de NERSC en Californie. Ce cluster est un Cray XE6 de 153216 cœurs et de 217 To de mémoire. Il est composé de 6384 nœuds : chaque nœud est un un bi-processeur de deux fois douze cœurs AMD **MagnyCours** de 2.1 GHz et une mémoire **NUMA** de 32 Go partagée sur 4 DDR3. Hopper a occupé la 5 ème place dans la liste Top500 de novembre 2010. Ces nœuds sont interconnectés par un réseau **Cray Gemini Network** en tore 3D.

Dans nos tests, nous avons utilisé des séquences d'ADN réelles extraites du site du NCBI (www.ncbi.nlm.nih.gov). Les noms des séquences et leurs tailles sont indiquées dans le tableau 6.1. Les deux dernières entrées dans cette table ont été exécutées uniquement sur l'**Hopper-Machine** à cause de leur énorme taille.

Comparison	Taille	Nombre d'accession	Nom
1K×1K	1,440 BP	NC_004991.1	*Acetobacter pasteurianus plasmid*
	2,747 BP	NC_005026.1	*Bacteroides fragilis* IB143
10K×10K	10,280 BP	AY352275.1	*HIV-1 isolate* SF33 from USA
	10,035 BP	AF133821.1	*HIV-1 isolate* MB2059 from Kenya
18K×18K	17,904 BP	AF079780.2	*Tupaia paramyovirus*
	18,324 BP	AF017149.2	*Hendra virus*
50K×50K	57,473 BP	NC_001715.1	*Alomyces macrogynus mitochondrion*
	56,574 BP	AF494279.1	*Chaetosphaeridium globosum mitochondrial DNA*
85K×85K	85,603 BP	AB15353.1	*Bacillus subtilis subsp. natto pLS32 str. IAM 11631*
	85,603 BP	NC_015149.1	*Bacillus subtilis subsp. natto pLS32*
150K×150K	162,114 BP	NC_000898.1	*Human Herpesvirus* 6B
	171,823 BP	NC_007605.1	*Human Herpesvirus* 4
500K×500K	542,868 BP	NC_003064.2	*Agrobacterium tumefaciens*
	536,165 BP	NC_000914.1	*Rhizobium sp.*
1000K×1000K	1,044,459 BP	CP000051.1	*Chlamydia trachomatis*
	1,072,950 BP	AE002160.2	*Chlamydia muridarum*
5000K×5000K	5,302,044 BP	AE016879.1	*Bacillus anthracis str. Ames*
	5,303,436 BP	AE017225.1	*Bacillus anthracis str. Sterne*
23000K×23000K	23,340,370 BP	NT_033779.4	*Drosophila melanogaster chromosome 2L*
	24,894,269 BP	NT_037436.3	*Drosophila melanogaster chromosome 3L*

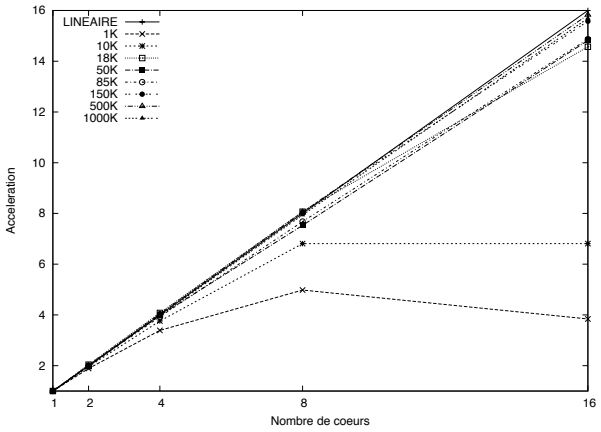
TABLE 6.1 – Détails sur des séquences réelles. La taille est de 1 KBP à 23 MBP.

6.2.4.1 Performances des différentes implémentations

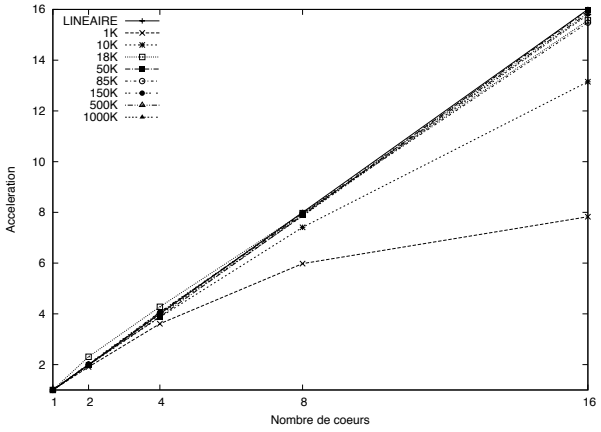
Pour comprendre les accélérations présentés dans cette section, il est intéressant de remarquer que notre algorithme `BSP++SW` utilise tous les éléments de traitement pour calculer une seule matrice de similarité. Par conséquent, il implique une quantité considérable de communications inter-processeurs.

Sur la **AMD-Machine**, nous avons exécuté les versions *BSP++MPI* et *BSP++OMP* générées par notre framework. Les figures 6.12(a) et 6.12(b) montrent les accélérations obtenues pour chaque version respectivement.

Comme prévu, les accélérations obtenues avec la version OMP sont meilleures que celles obtenues avec la version MPI. Aussi, pour des séquences de 1 K, les accélérations sur 16 cœurs ne passent pas à l'échelle (3,84 pour *BSP++MPI* et 7,83 pour *BSP++OMP*). Cela arrive parce qu'il n'y a pas assez de calculs pour compenser le surcoût des communications. D'autre part, les comparaisons de séquences de plus de 10 K atteignent des accélérations qui sont proches de 16, montrant que les deux versions passent à l'échelle jusqu'à 16 cœurs pour des séquences de grandes tailles.



(a) version MPI

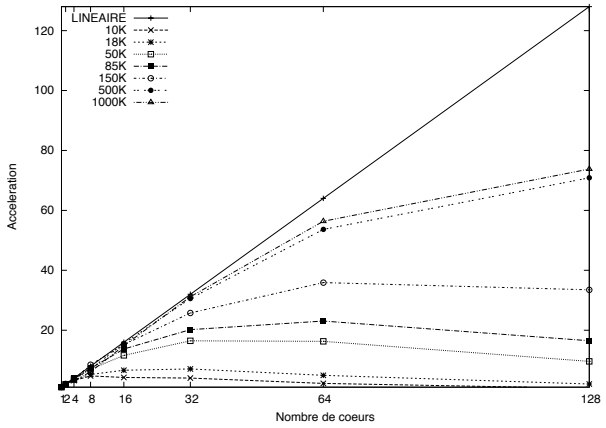


(b) version OpenMP

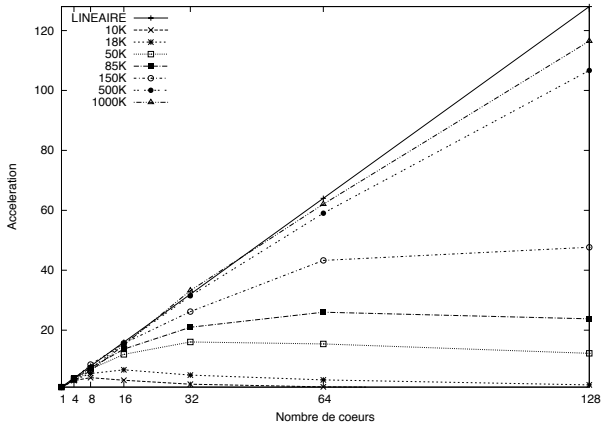
FIGURE 6.12 – Accélérations des versions BSP++ de SW sur AMD-Machine.

Afin de voir si nos versions `BSP++` sont appropriées pour un plus grand nombre de cœurs, nous avons exécuté les mêmes comparaisons sur le **Cluster-**

Machine de 128 cœurs, avec les versions *BSP++MPI* et *BSP++MPI+OMP*.



(a) version MPI



(b) version Hybride

FIGURE 6.13 – Accélérations des versions BSP++ de SW sur Cluster-Machine.

Pour calculer les accélérations (figures6.13(a) et 6.13(b)), le temps d'exécution sur un seul processeur avec la version *BSP++MPI* a été utilisé comme référence car, dans ce cas, la primitive *put* est vide et le code effectue essentiellement un appel à la fonction séquentielle SW (ligne 7 dans le listing 6.3). Sur le **Cluster-Machine**, la meilleure accélération a été obtenue avec la version hybride. Avec cette version, sur 128 cœurs, une accélération de 116 a été atteinte pour la comparaison des séquences de 1000 K x 1000 K, réduisant le temps d'exécution de $12487,79$ sec (un cœur) à $171,56$ sec (128 cœurs). La même comparaison avec le même nombre de cœurs, utilisant la version MPI atteint une accélération de 73. Cette différence dans les accélérations est due au fait que la version hybride profite d'un schéma de communication à deux niveaux qui correspond parfaitement à l'architecture multi-cœurs. Pour les séquences dont la taille est inférieure ou

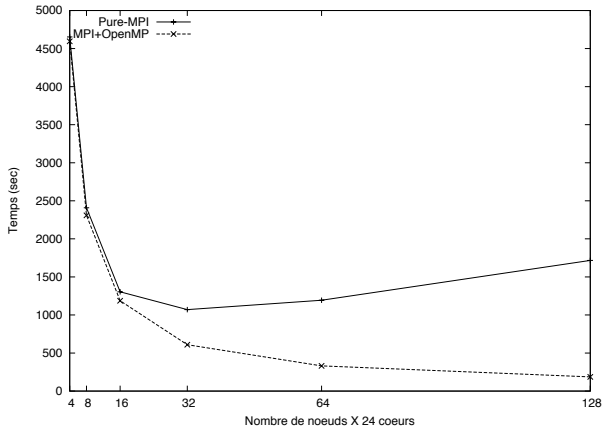
égale à 150 K, les performances ne passent pas à l'échelle pour 128 cœurs. Ceci est une indication que la version hybride MPI+OpenMP est une stratégie appropriée pour comparer des grandes séquences en réduisant considérablement le temps d'exécution.

Dans l'objectif de tester l'efficacité et le passage à l'échelle de `BSP++` sur les architectures *peta-scale*, nous avons effectué des comparaisons de séquences de tailles énormes (5 MB et 23 MB) sur le **Hopper-Machine** avec ***3072*** cœurs et ***6144*** cœurs respectivement. Avant de présenter nos résultats, notez bien que c'est une architecture NUMA de 24 cœurs par nœud. L'utilisation d'un style de programmation SPMD offert par `BSP++` dans tous les niveaux de la hiérarchie (on rappelle que le code OpenMP généré est du style OpenMP-SPMD : chapitres 4 et 5), nous a permis d'utiliser une configuration de 24 threads OpenMP par processus MPI avec une accélération de $23,78$ comparé à un seul thread OpenMP. Ces performances sont dues au fait que chaque thread utilise ses propres variables stockées dans son espace mémoire (RAM associé à son processeur avec accès rapide).

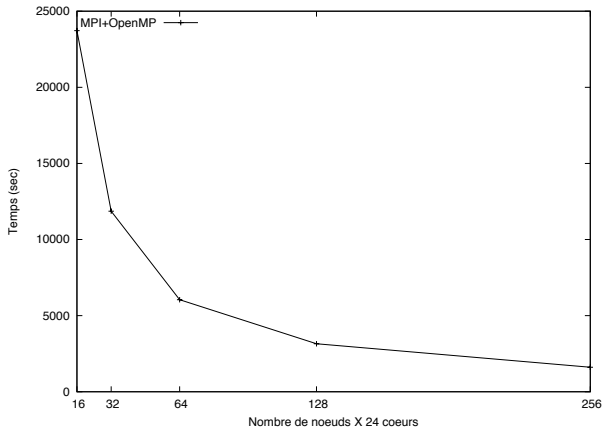
A cause de la limitation dans le temps d'utilisation et de réservation sur **Hopper-Machine**, les tests avec les séquences de 5 Mo sont effectués sur un nombre de cœurs allant de 96 (4 nœuds) à 3072 cœurs (128 nœuds). Pour les séquences de 23 Mo, le nombre de cœurs utilisés est entre 384 et 6144 cœurs. La figure 6.14(a) montre la comparaison en terme de performance et de passage à l'échelle entre les versions pure MPI et hybride sur les séquences de 5 Mo. Comme prévu, la version hybride est plus performante et plus adaptable aux machines avec un grand nombre de processeurs. En effet la version MPI a une accélération linéaire jusqu'à 384 processus MPI. Au delà de ce nombre, on a une décélération et une dégradation des performances due à l'augmentation des temps de communication et de synchronisation. Par contre, la version hybride a une accélération presque linéaire jusqu'à 3072 cœurs réduisant le temps d'exécution de $4595,89$ secondes sur 96 cœurs à 189 secondes sur 3072 cœurs.

La figure 6.14(b) montre les performances en terme de temps d'exécution de la version hybride avec les séquences de taille énorme : 23 Mo. Les résultats montrent une accélération linéaire jusqu'à 6144 cœurs. L'accélération obtenue en passent de 16 nœuds (384 cœurs) à 256 nœuds (6144) cœurs est de l'ordre de 15.

En utilisant la technique des partitions par blocs cycliques, on a été capable d'exécuter des comparaisons de taille 85 K x 85 K sur la plateforme **Cell-Machine** en utilisant un CellBE (QS20), deux CellBEs (QS22) et trois CellBEs (QS20&QS22), avec 8, 16 et 24 SPEs, respectivement. La figure 6.15 présente les accélérations obtenues pour ces comparaisons. Le temps d'exécution avec un seul CellBE (version *BSP++ CellBE* du listing 6.4) a été utilisé comme référence pour calculer ces ac-



(a) taille de séquences = 5 Mo



(b) taille de séquences = 23 Mo

FIGURE 6.14 – Temps d'exécution des versions MPI et hybride MPI+OpenMP sur *Hopper-Machine* avec des séquences de taille énorme.

célérations. Pour les séquences de taille 50 K et 85 K, les accélérations obtenues ont été très bonnes (2,6 et 2,8 respectivement), montrant ainsi que nos versions CellBE sont également en mesure de passer à l'échelle et de réduire considérablement les temps d'exécution.

Enfin, nous avons mesuré le surcoût introduit par la synchronisation supplémentaire entre les deux niveaux mis en œuvre dans la version *BSP++MPI+Cell* (Listings 6.5 et 6.6) en comparant son temps d'exécution (en secondes) avec la version mono-Cell du listing 6.4 (Tableau 6.2). Dans cette comparaison, un seul CellBE a été utilisé. De ce tableau, nous pouvons voir que le surcoût est faible pour de longues séquences (environ 7%), montrant que notre stratégie de synchronisation légère est appropriée pour des grappes de CellBE.

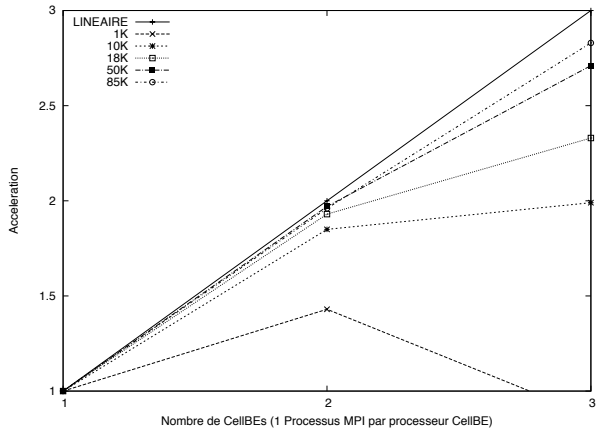


FIGURE 6.15 – Accélérations de la version BSP++ MPI+Cell sur Cell-Machine.

$versions$	$1K$	$10K$	$18K$	$50K$	$85K$
$Mono - Cell$	0.037	0.71	2.31	22.65	52.05
$Multi - Cell$	0.044	0.81	2.54	24.28	56.03
$surcot$	15%	11%	9%	7%	7%

TABLE 6.2 – Surcoût de la synchronisation PPE-SPE

6.2.4.2 Comparaison avec l'existant

Le tableau 6.3 liste onze propositions d'implémentations parallèles de SW. Sept de ces propositions sont optimisées pour le CellBE et les quatre restantes sont exécutées sur des grappes. Dans [117], un cluster de PS3 est utilisée, où MPI est utilisé pour la communication entre les PS3. Toutes les approches à base de grappes utilisent MPI pour la communication entre les nœuds et dans [118], ils utilisent MPI et OpenMP.

Dans la colonne 3, le type de comparaison est fourni, qui peut être entre deux séquences (*seq* x *seq*) ou entre une base de données génomiques et une séquence (*query* x *dbase*). Dans ce cas, chaque processeur compare la même séquence (*query*) avec un sous-ensemble des séquences de la base de données. Par conséquent, même si une grande quantité de comparaisons est faite, la taille des matrices de similarité n'est pas grande et elle est locale à chaque processeur.

La colonne 4 fournit le grain de calcul. Tous les comparaisons *seq x seq* sont des comparaisons à grain fin où tous les processeurs participent au calcul d'une matrice de similarité unique. Quand une séquence est comparée à une base de données (*query x dbase*), des comparaisons gros grain sont faites où chaque processeur calcule indépendamment un ensemble de matrices de similarité, sans communi-

cation avec les autres processeurs. Dans le tableau 6.3, toutes les approches qui permettent de comparer *query x dbase* emploient des calculs à gros grain et deux d'entre elles ([119], [117]) emploient également des calculs à grain fin pour de longues séquences.

La taille maximale des séquences comparées est indiquée dans la colonne 6. Quand une comparaison *seq x dbase* est faite, la taille de la séquence de requête la plus longue est fournie. Il y a une restriction sévère sur la taille des séquences pour le CellBE, où la séquence la plus longue avait 8 K de résidus [119]. Dans les grappes, les séquences comparées sont beaucoup plus longues.

La colonne 8 énumère les accélérations maximales rapportées par ces papiers. Ces accélérations ne peuvent pas être utilisées pour une comparaison directe pour de nombreuses raisons. Tout d'abord, certaines approches effectuent une comparaison gros grain[120] [121] [122] [123] [119] [117] [118], où ils calculent plusieurs petites matrices, alors que d'autres [124] [125] [126] [127] calculent des matrices de similarité qui sont beaucoup plus grandes et nécessitent la communication inter-processeurs. Deuxièmement, certaines approches qui fournissent l'alignement [124] [125] [126] [127] exécutent une variante de l'algorithme SW. Enfin, chaque approche utilise une référence différente pour le calcul des accélérations.

Pour mesurer les performances de SW, la métrique $GCUPS$ (*milliards de cellules mises à jour par seconde*) est souvent utilisée. Cette métrique calcule la vitesse à laquelle les cellules de la matrice de similarité sont mis à jour. GCUPS varie entre $0,42$ et $8,00$ pour le CellBE et entre $0,25$ et $1,45$ pour les grappes. Les GCUPS marqués d'un "*" ne sont pas fournis directement par les auteurs, mais calculés par nous même en utilisant les tailles des séquences et les temps d'exécution signalés.

Dans le tableau 6.3, nous pouvons voir que toutes les approches pour le CellBE imposent des restrictions sévères sur la taille des séquences. Aussi, les meilleures accélérations sont obtenues lorsque seul le score est fourni. En moyenne, les accélérations obtenues avec un gros grain sont meilleures que celles obtenues avec un grain fin. À l'exception de la notre, toutes les approches dans ce tableau ont été conçus et optimisées pour une plate-forme cible. Leur portage vers d'autres plateformes impliquerait une quantité énorme de reprogrammation.

Dans la dernière ligne, nous récapitulons les détails de notre approche. Nous avons été en mesure de générer du code pour MPI, OpenMP, MPI+OpenMP, CellBE et MPI+CellBE. Les accélérations et les GCUPs obtenus pour toutes les versions `BSP++` de SW sont comparables à ceux rapportés dans la littérature,

Papier	Plateforme	Comparaison	Grain	Sortie	Taille-Max Compar.	# Éléments	Meilleure Accélération	GCUPs
Cell/BE								
[120]	CellBE	query x dbase	coarse	score	4,000	6 SPEs	—	8.00
[121]	CellBE	query x dbase	coarse	score	2,048	16 SPEs	84x	—
[122]	CellBE	query x dbase	coarse	score	852	6 SPEs	30x	3.66
[123]	CellBE	query x dbase	coarse	score	1,024	8 SPEs	64x	—
[124]	CellBE	seq x seq	fine	score (SPE) align.(SPE)	3,584	16 SPEs	07x	0.80
[119]	CellBE	query x dbase	coarse and fine	score(SPE) align.(PPE)	8,196	16 SPEs	20x	—
[117]	cluster of CellBE	query x dbase	coarse and fine	score(SPE) align.(PPE)	3,584	84 SPEs	55x	0.42
Cluster								
[125]	cluster (MPI)	seq x seq	fine	score,align.	24,894,250	64 cores	33x	*1.45
[126]	2 grappes (MPI)	seq x seq	fine	score,align.	816,394	20 procs	14x	*0.37
[127]	cluster (MPI)	seq x seq	fine	score,align.	1,100,000	60 procs	39x	*0.25
[118]	cluster (hybride)	query x dbase	coarse	score	2,000	24 cores	14x	*4.38
Nos propositions								
Cette Thèse	cluster (MPI)	seq x seq	fine	score	1,072,950	128 cores	73x	6.53
	cluster (hybride)				1,072,950	128 cores	116x	10.41
	OpenMP				1,072,950	16 cores	16x	0.40
	CellBE				85,603	8 SPEs	—	0.14
	cluster de CellBE				85,603	24 SPEs	(8 :24) 2.8x	0.37
	hopper(MPI)				5,303,436	3072 cores	260x	3.09
	hopper(hybride)				24,894,269	6144 cores	5664x	15,5

TABLE 6.3 – Vue comparative des approches qui implémentent SW sur des plateformes HPC. Les lignes en gris mettent en évidence les approches qui sont semblables aux nôtres (seq x seq).

montrant que nos versions sont compétitives avec les implémentations existantes écrites à la main spécifiquement pour chaque architecture. Nos versions de SW ont été en mesure de comparer des séquences énormes sur des plates-formes à base de grappes. Grâce à une technique de partition, nous avons été en mesure de comparer des séquences beaucoup plus longues que celles des approches précédentes sur le CellBE. Les accélérations parfaites de la version MPI+CellBE (2,8 x pour 3 CellBEs contre 1 CellBE) montrent que la stratégie de synchronisation légère proposée est appropriée.

6.3 Application dans le domaine de traitement d'images

Dans les deux applications précédemment présentées, nous avons vu que nos versions pour le Cell BE, en plus de la facilité de leur génération en utilisant `BSP++/BSPGen`, passent à l'échelle linéairement. Cependant, nous avons souligné aussi leurs performances absolues médiocres comparées à la version multi-cœurs généralistes. Ces résultats sont dus à l'aspect du code (séquentiel) de ces applications. En effet, ces applications ne sont pas destinées aux processeur *"**vectoriels**"* qui exigent que le code soit explicitement écrit avec des instructions

vectorielles SIMD. Dans cette section, nous présentons une application dans le domaine de traitement d'images. Les fonctions séquentielles de cette application sont spécifiquement optimisées pour le processeur Cell en incluant des opérations SIMD.

La caractéristique commune des algorithmes de traitement d'images bas niveau est l'utilisation massive des opérateurs de base. En effet, le schéma typique d'une application de traitement d'images est l'appel répétitive à des opérateurs linéaires et locaux au niveau du pixel. L'algorithme de Harris [128] pour la détection de contours est un cas d'étude intéressant car il permet diverses stratégies de mise en œuvre et d'optimisations [10] dans le chaînage des opérateurs.

6.3.1 Description de l'algorithme de Harris

L'algorithme de Harris et Stephen [128] pour la détection des points d'intérêt est une variante améliorée de l'algorithme de Moravec [129] pour la détection des angles, utilisé dans le traitement d'images pour la détection de mouvement, l'appariement d'images (matching), le suivi (tracking), la reconstruction en 3D et la reconnaissance d'objets. Un coin peut être défini comme l'intersection de deux bords. Un point d'intérêt peut être défini comme étant un point qui a une position bien définie et qui peut être détecté. Ainsi, le point d'intérêt peut être un coin, mais aussi un point isolé avec une intensité maximale ou minimale, une fin de ligne ou un point sur une courbe dont la courbure est localement maximale. La figure 6.16 illustre l'utilisation de l'algorithme.

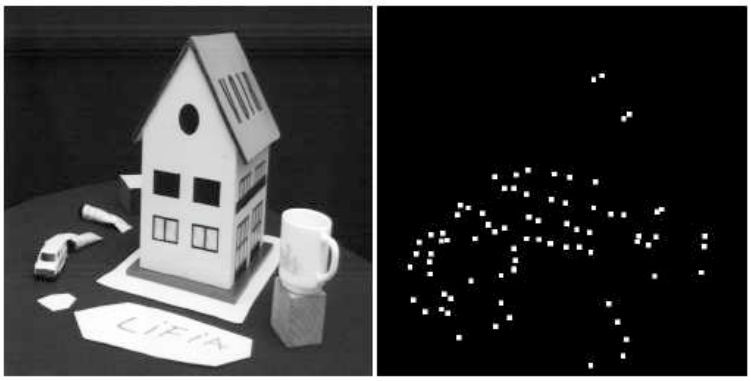


FIGURE 6.16 – Illustration de l'effet de l'algorithme de Harris.

L'algorithme est essentiellement une succession d'opérateurs locaux implémentants une forme discrète d'une autocorrélation S, donnée par :

$$S(x,y) = \sum_{u,v} w(u,v)[I(x,y) - I(x-u, y-v)]^2$$

où (x, y) est l'emplacement d'un pixel avec une valeur de couleur $I(x, y)$, et u, $v \in \{1, 2, 3\}$ modélisent le déplacement sur chaque dimension. A un point donné (x, y) de l'image, la valeur de $S(x, y)$ est comparée à un seuil approprié, et la décision est en fonction de la nature du pixel (x, y). Le processus est réalisé en appliquant quatre opérateurs discrets, à savoir $Sobel$ (S), $multiplication$ (M), $Gauss$ (G), et $Coarsity$ (C). La figure 6.17 montre une vue d'ensemble du flux des tâches.

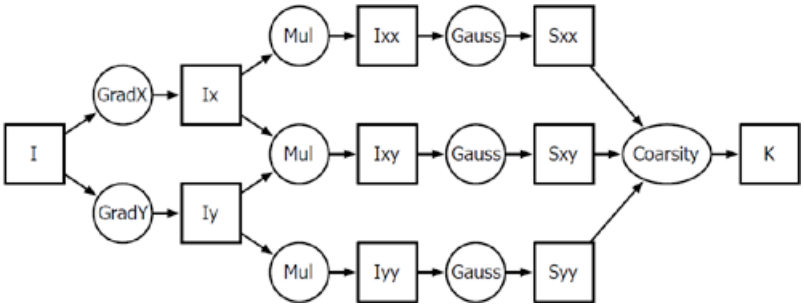


FIGURE 6.17 – Diagramme de flux de l'algorithme de Harris.

6.3.2 Parallélisation de l'algorithme de Harris

Il existe plusieurs stratégies pour paralléliser l'algorithme de Harris en utilisant la parallélisation d'exécution des opérateurs et le chaînage entre eux. Autrement dit, elles se basent sur quel code (suite d'opérateurs) exécuté par quel SPE et la communication (transfert des données) entre ces différent SPE pour satisfaire la dépendances des données entre ces différents codes (opérateurs) [10]. Parmi ces stratégies, on peut citer le **Schéma conventionnel SPMD** et le **Schéma demi-chaînage** [10]. Dans les figures suivantes décrivant ces stratégies , S désigne l'opérateur de Sobel, M la multiplication, G et H l'opérateur de Gauss et Harris respectivement. Les rectangles gris représentent une unité de SPE.

Schéma conventionnel SPMD : Dans le modèle de programmation SPMD conventionnel représenté par la figure 6.18, tous les SPU exécutent le même programme / code . Chaque SPE exécute un opérateur sur toute l'image avant de procéder à l'opérateur suivant. Par exemple, il ne sera pas question de la commencer l'opérateur de multiplication jusqu'à ce que tous les SPE aient terminé l'exécution de l'opérateur de Sobel et la totalité de l'image ait été transférée de nouveau dans la mémoire principale du PPE.

Schéma demi-chaînage : Dans cette version représentée par la figure 6.19, nous avons fusionné deux opérateurs successifs en paires, Sobel avec la multiplication, et Gauss avec Harris. Ainsi, on obtient deux fonctions, qui peuvent être

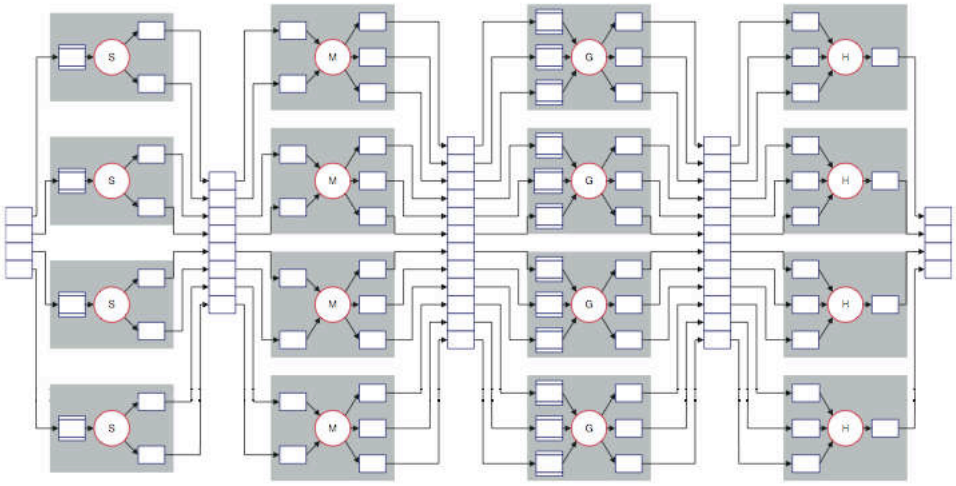


FIGURE 6.18 – Schéma conventionnel SPMD.

reproduites quatre fois pour remplir l'ensemble des SPE. Dans cette version, les SPE de rang pair exécutent la première fonction et ceux de rang impair exécutent la deuxième. Contrairement à la version précédente, celle-ci ne charge pas plusieurs fois l'image.

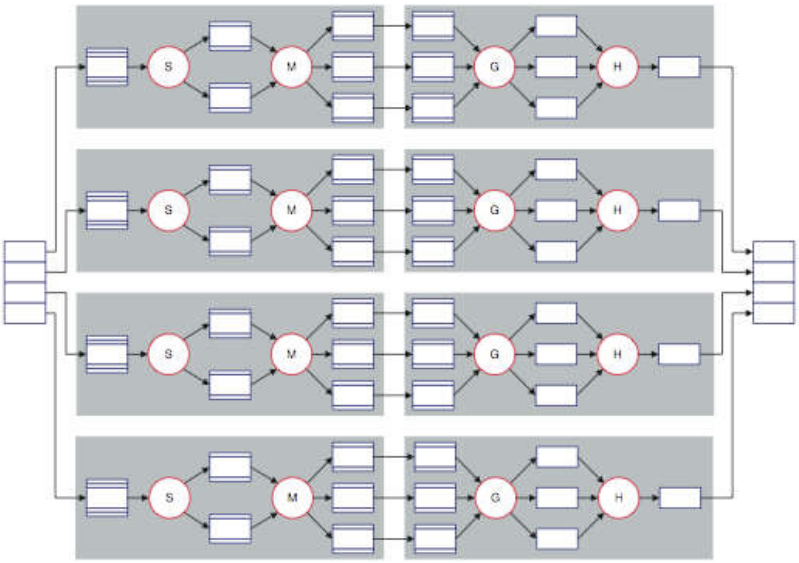


FIGURE 6.19 – Schéma demi-chaînage.

6.3.3 Algorithme de Harris avec BSP++

Nous avons implémenté les deux stratégies précédemment décrites avec BSP++. Cependant pour des raisons de performance et similarité dans leur conception, nous ne présentons que la deuxième version qui est bien évidement la plus performante.

La version BSP de cette stratégie contient une seule super-étape qui inclut exclusivement les deux fonctions de calcul [75]. En effet chaque SPU exécute soit la première fonction soit la deuxième selon son *rang*. Ensuite, il entame la phase de communication dans laquelle soit il transmet des donnée soit il reçoit des données. Le code 6.7 représente le pseudo-code BSP++.

Listing 6.7 – La version BSP++ du pseudo-code de la stratégie demi-chainnage de l'algorithme Harris

```
1  int bsp_main(int argc, char** argv, vector_2D<float> & IN, vector_2D<float>& OUT)
2  {
3  remote_iterator_input_2D<float> IN_it=remote_iterator_input_2D<float>(2,Bloc_X,
4          Bloc_Y, vector_slicer_2D());
5  remote_iterator_output_2D<float> OUT_it=remote_iterator_output<float>(2,Bloc_X,
6          Bloc_Y, vector_slicer_2D());
7
8  result_of::put<boost::function<vector_2D<float> (int) > >::type recv;
9  IN_it=IN.begin();
10 OUT_it=OUT.begin();
11 for(i=0; i< iterations; i++)
12  {
13   if(rank%2==0)
14    {
15     Function1_paire(&(*IN_it[0]), &(*OUT_it[0]), Bloc_X, Bloc_Y,...) ;
16     IN_it++;
17    }
18   else if(i>0)
19         {
20        Function2_impaire(&(*recv(rank-1)[0]), &(*OUT_it[0]), Bloc_X, Bloc_Y,... );
21        OUT_it++;
22         }
23      par<boost::function<vector_2D<float> (int) > > s= send_(*IN_it, rank);
24      recv=put(s);
25  }
26  return 0;
27 }
```

Le code 6.7 contient des constructions BSP++ spécifiques à l'implémentation Cell. Notamment, on trouve les *itérateurs* fonctionnant en mode *double buffering* pour recouvrir le temps de calcul des temps de transfert des données entre les SPE et la mémoire principale de PPE. A partir de ce code, on remarque bien qu'à l'itération i les SPU du rang pair exécutent la fonction $Function1\_paire$, alors que les SPUs de rang impair exécutent la fonction $Function2\_impaire$ en utilisant comme entrée les résultats de la communication de l'itération $i-1$. La fonction $send\_$ est un $Fonteur$ décrivant le schéma de communication précédemment présenté.

Le code précédant montre bien comment BSP++ contribue à faciliter la programmation et le déploiement des applications parallèles sur des architectures hybrides et hétérogènes. Notamment, la facilité de tester plusieurs stratégies/algorithmes de parallélisation d'une application donnée sur une architecture spéciale comme le Cell BE.

6.3.4 Performances

Nous avons mesuré le temps d'exécution de la version BSP++ de l'algorithme de Harris avec deux tailles différentes d'image. La table 6.4 rapporte des résultats sous forme de trois métriques. En plus du temps total d'exécution en secondes, nous avons utilisé deux autres métriques bien connues dans le domaine de traitement d'images à savoir le *nombre de cycles par point (CPP)* et *le nombre d'images par secondes (FPS)*. A partir de ces résultats, on constate le passage à l'échelle de la version hybride MPI+Cell sur la machine **Cell-Machine**. Pour les performances absolues de la version BSP++ comparé à une version écrite à la main, nous avons comparé les temps d'exécution de notre version sur 1 seul Cell avec une version écrite à la main par un coauteur dans le papier [75]. Pour une image de taille 512×512, la version manuelle utilisant des transferts DMA-2D exhibe un temps total de $0,0085$ secondes. Avec une différence de moins de 4%, la version BSP++ offre des performances proches de celles d'un code d'expert.

TABLE 6.4 – Temps d'exécution des versions BSP++ de l'algorithme Harris.

	image 256			image 512		
	temps (s)	CPP	FPS	temps (s)	CPP	FPS
1 Cell (8 SPEs)	0.0063	307	158	0.0088	108	114
2 Cells (16 SPEs)	0.0034	166	294	0.0047	57	213
3 Cells (24 SPEs)	0.0024	117	416	0.0031	37	323

6.4 Benchmark de calcul scientifique

En plus des benchmarks et des tests de calcul scientifique déjà présentés et testés avec BSP++/BSPGen dans les chapitres précédents, nous allons présenter dans cette section un autre benchmark bien connu de la communauté de calcul scientifique et de l'informatique parallèle : le benchmark CG de la suite NAS de la NASA.

Les benchmarks parallèle NAS (*Numerical Aerodynamic Simulation*) NPB est

un petit ensemble de programmes conçus pour aider à évaluer les performances des super-ordinateurs parallèles. Ces benchmarks sont issus des applications de calcul de dynamique des fluides (CFD). Ils se composent de cinq noyaux et trois pseudo-applications [5, 4].

Le benchmark CG (Conjugate Gradient) est souvent utilisé pour évaluer les performances de la machine et comparer les caractéristiques des différents modèles de programmation. Il résout un système linéaire non structuré par la méthode du gradient conjugué. CG est assez gourmand en mémoire. Il teste des communications longue distance irrégulières et une multiplication matrice-vecteur non structurée. CG utilise la méthode de puissance inverse pour trouver une estimation de la plus grande valeur propre d'une matrice creuse symétrique positif définie avec un motif aléatoire des différents zéros.

6.4.1 Version parallèle

Nous avons utilisé la version MPI (NPB2.3) du benchmark CG qui est une version très optimisée. Cette version MPI de CG accepte un nombre de processeur en puissance de deux qui sont mappés sur une grille de lignes et colonnes. Le sous-programme **makea** génère une sous-matrice (n/nprows par n/npcols) pour chaque processeur. Les vecteurs p et q sont dupliqués et distribués. Plus précisément, les vecteurs sont dupliqués sur les lignes et dans chaque ligne, ils sont distribués. La partie la plus gourmande à la fois en temps de calcul et de communication est la multiplication matrice-vecteur **q=A.p**. Après le calcul des résultats partiels, chaque processeur transmet ces résultats le long de sa ligne et effectue une sommation de tous les résultats.

Cette version est écrite en Fortran. Donc pour élaborer la version `BSP++`, nous avons dans un premier temps créé la version C en utilisant un compilateur source à source : **f2c**. A partir de la version C+MPI, nous avons extrait les portions de calcul séquentielles et les schémas de communication. En utilisant ces fonctions séquentielles et en réécrivant les schémas de communication avec la primitive `put`, nous avons conçu les versions `BSP++`.

6.4.2 Version BSP++

La version originale MPI contient plusieurs étapes et utilise différents schémas de communication. Le plus important, comme mentionné précédemment, est l'échange des sous vecteurs en ligne. Dans cette version, l'échange se fait par des $Send$ et des $Receive$ à l'intérieur d"une boucle qui itère sur le nombre de processeurs par ligne.

Dans la version BSP++, ce schéma de communication consiste simplement en un **seul** appel à la fonction put où chaque processeur renvoie sa valeur locale du vecteur q à tous les processeurs sur la même ligne. Le code 6.8 illustre cette fonction de transfert. Dans ce code, les paramètres $nbproc\_col$ et id_proc_row correspondent au nombre de processeurs en colonne dans la grille et à l'identifiant du processeur appelant sur la ligne respectivement.

Listing 6.8 – La fonction utilisateur décrivant le schéma de communication de CG.

```
1 vector<double> sender( int ind, par<vector<double> > const& v,
2                        const int nbproc_col, const int  id_proc_row)
3 {
4   if(ind/nbproc_col==id_proc_row)  return *v;
5   else return vector<double>();
6 }
```

6.4.3 Performances

En utilisant BSPGen, nous avons généré les deux versions MPI et hybride MPI+OpenMP et nous avons comparé leurs performances en utilisant le cluster de $grid5000$ avec 64 cœurs. Dans nos tests, la classe A du benchmark CG a été utilisée. La figure 6.20, montre la comparaison des temps de calcul et de communication des versions MPI et MPI+OpenMP en fonction du nombre de cœurs. A partir de ces résultats, on peut clairement voir que les deux versions ont presque les mêmes performances en temps de calcul vu que les deux modèles utilisent le style SPMD. Cependant, concernant le temps de communication, la version hybride est de loin la plus performante. En effet cette version a un gain allant de 45% à 62% de 4 à 64 cœurs respectivement.

En ce qui concerne les performances absolues de nos versions BSP++, nous les avons comparées avec les versions écrites et optimisées à la main effectué par les auteurs de [4]. Nos deux versions exhibent les mêmes comportements en fonction du nombre de processeurs que ceux des versions du papier en question. De plus, nos versions affichent des ratios temps calcul/ communication équivalents à ceux des versions écrites à la main. Deux raisons empêchent une comparaison directe des temps d'exécutions :1) utilisation de machines avec des fréquences différentes. Cependant, les deux sont des grappes de multiprocesseurs. 2) utilisation de langages sources différents(Fortran vs C/C++).

6.5 Conclusion

Nous avons implémenté et testé **plusieurs applications** différentes appartenant à des domaines différents sur plusieurs architectures de **CHP**. Pour chacune de ces applications - **APMC, SW et Harris** -, nous avons été en mesure

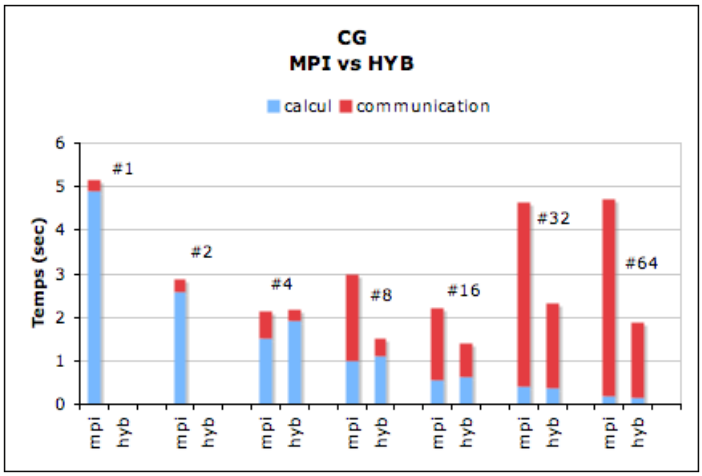


FIGURE 6.20 – Temps d'exécution des deux versions BSP++ MPI et MPI+OpenMP du benchmark CG avec la classe A.

de générer 5 versions différentes en utilisant un seul code de base. Ces versions ont été exécutées sur des architectures variées allant d'un simple nœud multi-cœurs à mémoire partagée (semblable aux futures machines de bureau) jusqu'aux grappes Téraflops avec des caractéristiques architecturales spécifiques à cette classe de machines (des réseaux dédiés et des mémoires **NUMA**) en passant par des grappes de taille moyenne et des grappes d'accélérateurs de type Cell BE. Ces tests ont en particulier mis en évidence les points suivants :

D'une part, les performances de nos versions sont comparables à l'existant et aux codes écrits manuellement et spécifiquement pour l'architecture cible. De plus, on a vu dans ce chapitre que nos versions ont une accélération linéaire et un passage à l'échelle jusqu'à 6000 cœurs pour l'algorithme SW. D'autre part, l'indépendance architecturale ainsi que la haute abstraction de nos outils facilitent énormément la conception et le développement des codes parallèles. En effet comme souligné auparavant, la réflexion sur la conception d'une version parallèle consiste simplement à l'élaboration des motifs de communication et à la décomposition en plusieurs super-étapes. A titre indicatif, le temps de conception de la version parallèle de base sous forme BSP de ces applications (à partir du code séquentiel) est de l'ordre de quelques jours et en utilisant `BSPGen` et/ou directement `BSP++`, le temps de développement est de quelques heures.

Ces résultats valident donc la pertinence et l'efficacité en terme de performances et de productivité de notre plate-forme de développement.

CONCLUSIONS ET PERSPECTIVES 7

7.1 Bilan

Dans le cadre de cette thèse, nous avons proposé une nouvelle plate-forme de développement pour les machines de calcul haute performance. Les principales caractéristiques que nous avons considérées pour cette plate-forme sont : 1) la haute abstraction et l'indépendance de l'architecture cible permettant une facilité d'utilisation ; 2) l'efficacité en offrant des performances comparables à une implémentation spécifique à l'architecture cible ; 3) la prédiction des performances en se basant sur un modèle de coût, pour faciliter l'optimisation des algorithmes ainsi que le choix de l'architecture cible pour une exécution performante.

Notre recherche s'est basée sur une combinaison de plusieurs approches : 1) l'étude et la comparaison des modèles et outils existants ; 2) la proposition de nouvelles idées et leur implémentation et 3) des expérimentations avec des applications réalistes de différents domaines pour démontrer le bon fonctionnement et l'efficacité de notre plate-forme. Ce chapitre reprend, de manière synthétique, les apports de chacune de nos réalisations et présente nos perspectives de recherche.

La programmation et le déploiement des applications sur une machine hybride et hétérogène n'est pas une tâche triviale. Divers outils et modèles ont été proposés pour permettre la programmation de telles machines, tant de bas niveau que de haut niveau. Les principaux défauts de ces outils résident dans le compromis entre l'expressivité et les performances du code qu'ils produisent. Dans cette optique, nous avons cherché à définir des techniques permettant la conception des outils de développement parallèle à la fois **expressifs et efficaces**.

La technique retenue utilise un mécanisme assez peu exploité du langage C++ dans le domaine de parallélisme : *la programmation générique* via les *template* - ou patrons -. En utilisant **un système d'évaluation**, le compilateur génère un code très proche d'un code C écrit et optimisé à la main tout en conservant une inter-

face utilisateur très expressive. Nous avons donc proposé une bibliothèque C++ : BSP++ [72, 112, 116], qui gère le développement de code parallèle hybride et hiérarchique. En utilisant le modèle de coût de BSP++, nous avons développé un framework : BSPGen [80], qui tente de prédire les performances d'une application donnée sur une architecture donnée et génère ensuite le code parallèle le plus performant à partir d'une liste de fonctions séquentielles.

La haute abstraction de la bibliothèque BSP++ et le modèle hiérarchique sous-jacent, qui est en adéquation directe avec les caractéristiques matérielles des machines hybrides, ont montré une facilité d'utilisation et un haut niveau d'expressivité tout en restant efficace. Pour ce qui concerne le framework, on a vu que BSPGen génère un code parallèle hybride à partir d'une liste de fonctions séquentielles. Il combine une analyse statique réalisée par son module d'$analyse$ et une analyse dynamique à l'exécution réalisée par un système de $profiler$ pour estimer le temps d'exécution. En utilisant cette estimation, il génère la suite des configurations les plus optimales.

L'efficacité et les performances de nos outils ont été montrées par une série de tests sur plusieurs architectures hybrides en utilisant différentes applications allant de simple benchmarks de calcul scientifique à de véritables applications dans des domaines différents comme la vérification [112] et la bioinformatique [116]. De plus, avec cette dernière application, les résultats obtenus sur une architecture $Petaflops$ utilisant des mémoires NUMA, montrent que nos outils sont utilisables pour ce type d'architecture. Dans tous les cas, pour toutes les applications, les résultats obtenus sont comparables à ceux existants pour des codes écrits à la main et optimisés pour chaque architecture.

7.2 Perspectives

A partir de ces résultats, plusieurs perspectives et axes de recherches dans différentes directions sont envisageables.

Au niveau de la bibliothèque BSP++, une extension pour de nouvelles architectures cibles est concevable. Par exemple, étendre BSP++ sur les MPSoC (Multiprocessor System-on-Chip). Comme pour la version du Cell, en se basant sur une implémentation MPI pour les MPSoC, on développe une extension des primitives BSP++ pour cette architecture. De plus, cette extension pour les architectures parallèles embarquées est une perspective prometteuse. En effet, se basant sur le modèle de coût BSP++, on peut estimer le temps d'exécution *pire cas* qui est la métrique de base pour les applications temps réel strict sur ce type d'architectures.

Pour un niveau supplémentaire dans l'abstraction et la simplicité d'utilisation, nous envisageons une **STL parallèle avec `BSP++`** comme dans STAPL [130]. L'idée consiste en la conception et le développement de conteneurs et d'algorithmes parallèles `BSP++` de haut niveau. Par exemple, on peut imaginer un conteneur **BSP_Vector** qui se déploie automatiquement sur la hiérarchie de la machine BSP cible. Ainsi, l'appel à un algorithme **BSP_Sum** par exemple, sur le BSP_vector effectue toutes les imbrications nécessaires (MPI/OpenMP/Accélérateur) pour faire la somme du vecteur. Avec le même objectif d'abstraction, l'augmentation de l'aspect fonctionnel de `BSP++` est possible en le définissant comme un EDSL (Embedded Domain Specific Language). Dans cette optique et en utilisant le mécanisme proto de C++, l'utilisateur décrit la totalité de son programme sous forme de fonctions et primitives proto-BSP++.

Au niveau de l'outil `BSPGen`, actuellement, l'analyse et l'estimation de temps de calcul se font uniquement en statique et de manière ad-hoc (on estime le temps de calcul et le temps d'accès mémoire avec la même stratégie). Pour plus de précision, nous pensons introduire le modèle polyédrique pour la détermination des configurations d'accès dans les nids de boucles.

Avec le même objectif de précision, on envisage l'utilisation d'un système dynamique pour l'estimation du temps d'exécution. Ce système permet de modifier la configuration (nombre de threads, accélérateurs) en cours d'exécution en se basant sur les performances obtenues.

Pour étendre nos outils aux processeurs graphiques (GPU), nous examinerons l'utilisation d'un compilateur source à source pour la génération d'un code GPU à partir du code utilisateur comme dans PGI [32] et HMPP [131]. Ensuite, en se basant sur l'idée de base de l'analyseur et en l'adaptant pour l'utilisation d'un code PTX, nous estimerons le temps de calcul d'un Kernel. A partir de cette estimation et de celle du temps de transfert des données, `BSPGen` pourra inclure les GPUs comme accélérateurs dans la construction du graphe des configurations.

Les résultats obtenus ont montré la facilité de la programmation des architectures hybrides en utilisant nos outils. Cependant, la "*difficulté*" se situe en niveau algorithmique. En effet, pour plus d'efficacité et d'optimisation, la tâche principale du programmeur consiste en la conception d'algorithmes BSP qui optimisent le nombre du supers-étapes. Dans le but d'automatiser cette tâche, nous voulons offrir un module pour la détermination de l'algorithme optimal, en utilisant une programmation génétique ou encore, une recherche exhaustive à partir du graphe de dépendances des tâches.

ANNEXE : NOTIONS C++ ET *Templates* A

Dans la partie qui suit, nous allons introduire certaines notions techniques avancées de programmation C++, notions qui apparaissent régulièrement dans cette thèse : les *Templates*, les foncteurs, les lambda fonctions et le protocole `result_of`.

A.1 Templates

Les *templates* - ou modèles - sont un mécanisme du langage C++ dont le but principal est de fournir un support pour la programmation dite **générique**, favorisant ainsi la réutilisation de code paramétrable. Un template définit une famille de classes ou de fonctions paramétrées par une liste de valeurs ou de types.

A.1.1 Classe template

Le listing A.1 présente la définition et l'utilisation d'une classe de tableaux $template$. Cette classe de tableaux utilise deux paramètres template (ligne 1) : T qui définit le type des éléments stockés dans le tableau et N qui définit la taille du tableau. Lors de l'utilisation de cette classe, l'utilisateur précise la valeur de ces deux paramètres et **instancie** la classe template (lignes 8 et 9).

Plusieurs points sont à noter :
– L'instanciation d'un $template$ a lieu lors de la compilation et génère un code temporaire dans lequel les divers paramètres templates sont remplacés par leurs valeurs effectives. Ce code intermédiaire est ensuite compilé normalement. Contrairement au macro-définitions, toutes le vérifications syntaxiques et sémantiques sont effectuées.

– Une classe template n'existe que lorsque ses paramètres du modèle sont fixés. Ainsi la déclaration :

array a ;

n'a pas de sens et provoque une erreur de compilation.
– Deux instances du même patron utilisant des paramètres templates différents produisent deux types différents. Par exemple, le type *array<double, 3>* est incompatible avec le type *array<int, 8>*.

Listing A.1 – Exemple de classe template

```
1  template<typename T, int N> class array
2  {
3    public:
4    static const size_t size= N;
5
6     array() {}
7    ~array() {}
8     array( const array<T,N>& src) ;
9     array<T,N> operator=(const array<T,N>& src);
10
11    // ...
12 };
13  array<double, 3> t1;
14  array<int, 8>    t2;
```

A.1.2 Fonction template

Il est aussi possible de définir des fonctions templates. le listing A.2 définit une fonction calculant la somme des éléments d'un tableau. Cette fonction accepte des arguments de n'importe quel type atomique (int, double,...) ou d'un type défini par l'utilisateur, à condition que celui-ci fournisse une surcharge pour les opérateurs «=» et «+=». On note aussi que l'appel de la fonction à la ligne 8, ne requière pas une spécification explicite du type du tableau, le compilateur étant capable d'inférer ce dernier en analysant l'appel.

Listing A.2 – Exemple de fonction template

```
1  template<int N, typename T> inline T sum (T* array)
2  {
3    T r = 0;
4    for(int i=0; i<N; i++) r += array[i];
5    return r;
6  }
7   double a[] = {1.0, 3.3, 5.12, 8.3};
8   double s= sum<4>(a);
```

A.2 Foncteur

La plupart des algorithmes de la bibliothèque standard, ainsi que quelques méthodes des classes qu'elle fournit, donnent la possibilité à l'utilisateur d'appliquer une fonction aux données manipulées. Ces fonctions peuvent être utilisées

pour différentes tâches, comme pour comparer deux objets par exemple, ou tout simplement pour en modifier la valeur.

Cependant, la bibliothèque standard n'utilise pas ces fonctions directement, mais a plutôt recours à une abstraction des fonctions : les foncteurs. Un foncteur n'est rien d'autre qu'un objet dont la classe définit l'opérateur fonctionnel '()' : `:operator()(...)`. Les foncteurs ont la particularité de pouvoir être utilisés exactement comme des fonctions puisqu'il est possible de leur appliquer leur opérateur fonctionnel selon une écriture similaire à un appel de fonction. Cependant, ils sont un peu plus puissants que de simples fonctions, car ils permettent de transporter, en plus du code de l'opérateur fonctionnel, des paramètres additionnels dans leurs données membres. Les foncteurs constituent donc une fonctionnalité extrêmement puissante qui peut être très pratique en de nombreux endroits. En fait, comme on le verra plus loin, toute fonction peut être transformée en foncteur. Les algorithmes de la bibliothèque standard peuvent donc également être utilisés avec des fonctions classiques moyennant cette petite transformation.

Les algorithmes de la bibliothèque standard qui utilisent des foncteurs sont déclarés avec un paramètre template dont la valeur est celle du foncteur permettant de réaliser l'opération à appliquer sur les données en cours de traitement. Au sein de ces algorithmes, les foncteurs sont utilisés comme de simples fonctions, et la bibliothèque standard ne fait donc pas d'autre hypothèse sur leur nature. Cependant, il est nécessaire de ne donner que des foncteurs en paramètres aux algorithmes de la bibliothèque standard, pas des fonctions. C'est pour cette raison que la bibliothèque standard définit un certain nombre de foncteurs standards afin de faciliter la tâche du programmeur.

Listing A.3 – Exemple d'un foncteur

```
1  bool MaFonctionDeTri(const type & rl,const type & rr)
2  {
3  return rl < rr;
4  }
5
6  struct MonFoncteurDeTri
7  {
8  bool operator()(const type & rl,const type & rr)
9  {
10 return rl < rr;
11 }
12 };
```

L'exemple précédent montre la simplicité d'écriture de foncteurs. En effet, il suffit de placer le code de la fonction dans un opérateur() ayant le même type de retour et en prenant les mêmes paramètres. Comme précédemment indiqué, l'opérateur () d'une structure est résolu à la compilation et peut être mis en ligne, alors qu'un pointeur sur fonction utilise toujours une indirection à l'exécution et empêche quelques optimisations.

A.3 Lambda fonction

La Boost Lambda Library (BLL) est une bibliothèque template C++ qui implémente une forme d'abstractions lambda pour le langage C++. Le terme provient de la programmation fonctionnelle et le lambda-calcul, où une abstraction lambda définit une fonction anonyme. La principale motivation pour BLL est de fournir des moyens souples et commodes pour définir des objets fonction sans nom pour les algorithmes de la STL. La ligne suivante trie les éléments d'un certain conteneur STL.

Listing A.4 – Exemple d'une lambda fonction

```
1 for_each(a.begin(), a.end(), std::cout << _1 << ' ');
```

L'expression std : : cout«_1 «" ", définit un objet fonction unaire. La variable _1 est le paramètre de cette fonction, un $Placeholder$ (espace réservé) pour l'argument effectif. Au sein de chaque itération de for_each, la fonction est appelée avec un élément effectif comme argument. Cet argument réel substitue le placesholder et le «corps» de la fonction est évaluée.

L'essence de BLL est de permettre la définition d'objets fonction sans nom, comme celui ci-dessus, directement à partir de l'appel d'un algorithme de la STL.

A.3.1 Placeholders

La BLL définit trois types d'espace réservé : placeholder1_type, placeholder2_type et placeholder3_type. BLL a une variable espace réservé prédéfinie pour chacun de ces type : _1, _2 et _3. Toutefois, l'utilisateur n'est pas obligé d'utiliser ces placeholders. Il est facile de définir des espaces réservés avec des noms alternatifs. Ceci est fait en définissant de nouvelles variables de types placeholder. Par exemple :

Listing A.5 – Définition de Placeholder

```
1 boost::lambda::placeholder1_type X;
2 boost::lambda::placeholder2_type Y;
3 boost::lambda::placeholder3_type Z;
```

Avec ces variables définies, X += Y*Z est équivalent à _1+= _2 * _3. L'utilisation des placeholders dans l'expression lambda détermine si la fonction qui en résulte est d'arité nulle, unaire, binaire ou ternaire.

A.3.2 Expression Bind

Une expression $Bind$ retarde l'appel d'une fonction. Si cette fonction cible est n-aire, alors la liste d'argument-bind doit contenir n arguments. Dans la version actuelle de la BLL, on doit avoir 0<=n<= 9. Pour les fonctions membre, le nombre

d'arguments doit être au plus 8, car l'argument objet prend une position dans la liste. Fondamentalement, la liste d'arguments-bind doit être une liste d'arguments valables pour la fonction cible. Tout argument peut être remplacé par un placeholder, ou plus généralement par une expression lambda. Notez également que la fonction cible peut être une expression lambda. Le résultat d'une expression Bind est soit une arité nulle, unaire, binaire ou un objet fonction ternaire.

Le type de retour du foncteur lambda créé par l'expression $Bind$ peut être donné comme un paramètre Template spécifié explicitement, comme dans l'exemple suivant :

Listing A.6 – Définition d'une expression Bind

```
1 Function<double( double)> half_cos= bind<double>(&cos, _1/2)
```

Cela n'est nécessaire que si le type de retour de la fonction objectif ne peut être déduite. Dans notre exemple, comme la fonction cos existe avec des surcharges pour $double$ et int le type doit être spéficié.

A.4 Protocole result_of

La norme C++98 exige que les objets fonction unaires et binaires définissent un membre de type `result_type` spécifiant le type retourné par l'opérateur(). Bien que cette méthode fonctionne bien pour les objets fonction simple (elle a été adoptée même par des bibliothèques très flexibles et contraignantes telles que Boost.Bind), elle est incompatible avec les objets fonction utilisants un opérateur() $template$. Le type de retour de ces objets fonction dépend d'ensemble des types d'arguments. Ainsi, il est impossible de définir ce type de retour au moment de l'écriture du code. De telles fonctions ne correspondent pas au modèle objet fonction de la bibliothèque standard, et exigent donc la définition d'une nouvelle méthode pour déduire ces types de retour. On parle alors d'objets fonction *polymorphes* qui sont plus génériques que les objets fonction standards.

Pour ce faire, le standard prévoit l'utilisation du protocole **result_of**. Ce protocole définit deux méthodes pour déduire le type de retour de l'opérateur d'appel de fonctions (*opérateur()*) : la première utilise *result_type* pour la compatibilité avec les objets fonction simple, et la seconde utilise un membre $Template$ plus flexible, pour une utilisation avec des fonctions plus complexes. Les deux codes suivant illustrent ces deux méthodes.

Listing A.7 – Exemple de result_type

```
1  struct is_even
2 {
3   typedef bool result_type;
4   template<class T> result_type operator() (T const & t)
5    { return t%2 ==0;
6    }
7 };
```

Le code A.7 définit une classe is_even qui détermine si une valeur est paire ou non. Le type de retour est simple et connu puisque quel que soit le résultat de la fonction, son type de retour est un booléen (true or false). Dans ce cas, l'utilisation du result_type est suffisant pour déterminer le type de retour.

Pour illustrer le fonctionnement et l'intérêt du protocole, l'exemple A.8 expose ceux-ci.

Listing A.8 – Exemple de result_of

```
1  struct convert
2 {
3  template<class Sig> struct result;
4
5  template<class This, class T>
6  struct result<This(T)>
7  {
8  typedef typename mpl::if_<is_arithmetic<T>, std::String,
9                                              double
10                                               >::type type;
11 };
12  template<class T>
13  std::string operator() (T const& t) const
14  {
15  }
16  double operator() (std::String & s) const
17  {
18  }
19 };
20
21 std::result_of<convert(std::String) >::type x= convert("123");
22 std::result_of<convert(double) >::type y= convert(123);
```

Dans cet exemple, la classe $convert$ contient deux surcharges de l'opérateur() : la première prend string comme paramètre et l'autre un template. De ce fait, le type de retour de l'appel au foncteur $convert$ n'est pas connu lors de son instanciation. Pour résoudre ce problème, le type de retour du foncteur est exprimé en utilisant le protocole result_of comme présenté par les lignes 21 et 22. Toute fois cette exemple reste simple puisque le type de retour peut etre déduit directement. Pour mettre l'accent sur l'outilité de ce protocole la fonction $convert$ doit être plus générique. Le code A.9 illustre la fonction $alpha\_convert$ qui calcul le double du paramètre d'entrée. Comme le type d'entrée peut être une chaine ou un double le type de retour doit être résolut automatiquement.

Listing A.9 – Exemple 2 de result_of

```
1 template< class T>
2 typename result_of<convert(T)>::type
3 alhpa_convert(T const & s)
4  {
5   convert f;
6   return f(s)+f(s);
7  }
```

ANNEXE : ARCHITECTURE ET COMPILATION DU CELL BE B

Dans ce chapitre, nous allons présenter en détail l'architecture du processeur Cell ainsi que la procédure de compilation utilisée.

B.1 Architecture

Le Cell BE est un processeur multicœurs hétérogènes, avec un processeur PowerPC multithreads de base et jusqu'à huit cœurs (SPE) conçus pour l'exécution des tâches de calcul intensif. Les SPEs ont un accès rapide aux mémoires locales de 256 Ko et disposent de DMA pour transférer les données depuis la mémoire principale. Les unités du calcul SIMD sont omniprésentes sur le processeur Cell BE. Le PPE possède VMX (ou AltiVec, l'extension SIMD du standard PowerPC, et les SPE utilisent uniquement un jeu d'instructions SIMD.

On peut avoir une idée de l'organisation du Cell BE à partir de la figure B.1. Elle montre les différentes parties et comment le Cell BE est en communication avec l'extérieur, grâce à ses contrôleurs de mémoire et d'entrées/sorties.

B.1.1 Architecture EIB

Le processeur Cell BE est construit autour du bus d'interconnexion (EIB), qui est un bus interne haute vitesse pour déplacer des données entre les composants du processeur, donnant à chaque unité fonctionnelle (chacune des entités des SPEs, le PPE, l'interface mémoire, et les deux unités I/O) une bande passante du 25,6 Go/s dans chaque sens. La bande passante théorique de EIB est de 200 Go/s et ne peut être atteinte que pour un code très bien optimisé.

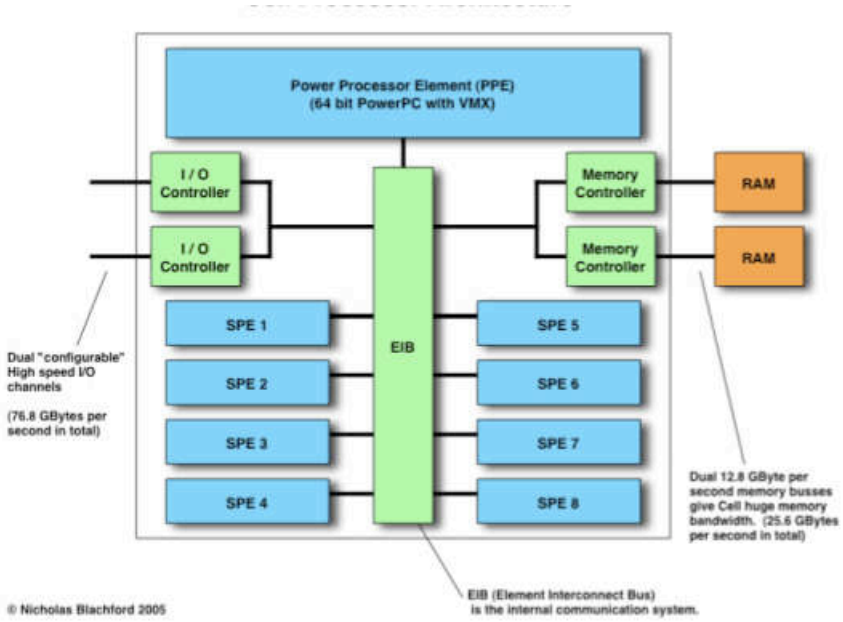


FIGURE B.1 – Une vue globale de l'architecture Cell BE.

B.1.2 Architecture PPE

Le PowerPC, habituellement appelé PPE, est une version multithread du PowerPC. Ce processeur RISC 64 bits a également l'extension vectorielle/SIMD. Tout programme écrit pour un processeur PowerPC devrait fonctionner sur le Cell Broadband Engine. La figure B.2 montre une vue très simplifiée du PPE. Le rôle du PPE est crucial dans l'architecture du Cell car il exécute d'une part le système d'exploitation, et contrôle d'autre part toutes les autres ressources, y compris les SPE (il n'y a pas de système d'exploitation dans ces unités).

B.1.3 Architecture SPE

Chaque Cell a 8 SPE (Synergistic Processor Elements). Ce sont des processeurs RISC 128 bits, et ils sont spécialisés pour les applications de calcul intensif SIMD. Ils sont constitués de deux unités principales. La figure B.3 montre une vue très simplifiée du SPE.

– Le SPU (Synergistic Processor Unit) implémente un ensemble d'instructions SIMD, spécifiques au Cell BE. Chaque SPU est indépendant et a son propre compteur de programme. Les instructions sont récupérées dans sa mémoire locale (LS). Les données sont également chargés et rangées dans la LS. Du point de vue du SPU, la LS est uniquement un espace de stockage non protégé. Il n'y a pas de cache sur le SPU. Ainsi, il n'y a pas de contrôle

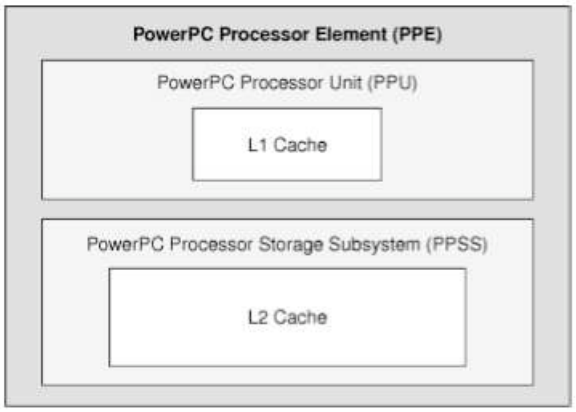


FIGURE B.2 – Diagramme simplifié du PPE.

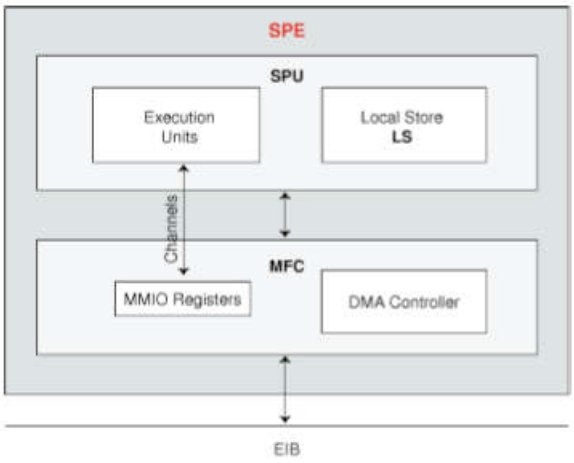


FIGURE B.3 – Diagramme simplifié du SPE.

de défauts de cache : il est du devoir du programmeur de s'assurer du bon utilisation de la mémoire. Si un accès illégal est effectué, le SPU s'arrête simplement et affiche une erreur de type **Segmentation fault**

– Le contrôleur de débit de la mémoire (MFC). C'est l'interface entre les SPU et le reste de la puce Cell. En fait, le MFC interface le SPU avec EIB. En plus de registres MMIO, il contient un contrôleur DMA. Pour charger ou ranger des données dans la mémoire principale, le SPU fait des requêtes asynchrones au contrôleur DMA du MFC. Le contrôleur satisfait la requête en même temps que les SPU continuent leur calcul.

B.2 Compilation

B.2.1 Code PPE

La programmation sur le PPU est identique à celle d'un PPC normale. Écrire du code qui utilise uniquement le PPU n'est pas différent de l'écriture de code sur toute autre architecture avec un système d'exploitation GNU/Linux.

Lorsque le SDK est installé, le répertoire par défaut pour les bibliothèques est /opt/IBM/. Comme son nom l'indique, ppu32-gcc, aussi appelé ppu-gcc est une version de gcc adaptée à la PPU. Ainsi, les options données au compilateur sont les options habituelles. La compilation du code écrit pour le PPE se fait via la commande suivante :

```
/opt/IBM/toolchain-3.2/ppu/bin/ppu32-gcc helloworld-ppu.c -o hello
```

B.2.2 Code SPE

Le SPE et PPE n'ont pas la même architecture, bien qu'il y ait de grandes similitudes dans le jeu d'instructions. Les programmes s'exécutant sur le SPU doivent être compilées spécifiquement pour cette architecture. Au lieu d'utiliser le compilateur gcc, il faut utiliser spu-gcc. Ce dernier est une adaptation de gcc pour le Cell. La compilation du code écrit pour le SPE se fait via la commande suivante.

```
/opt/IBM/toolchain-3.2/ppu/bin/spu32-gcc helloworld-spu.c -c hello_spu
```

On remarque bien qu'on utilise des codes séparés pour chaque architecture (bien que le même code puisse être compilé par les deux compilateurs). Cette séparation est utilisée dans la pratique car le PPE et les SPE n'effectuent pas les mêmes tâches. De plus, la compilation pour le SPE utilise l'option -c pour la génération de code objet et ne génère pas l'exécutable. La section suivante mettra en évidence la génération du code exécutable

B.2.3 Édition de liens

Pour être exécuté, le code généré par spu-gcc doit être intégré dans le code que pourrait utiliser un PPU. En fait, nous devons relier la fonction principale, mis en œuvre dans notre code précédent fonctionnant sur le SPU, dans un objet du PPU. Dans notre exemple, le programme helloworld-spu.c a une fonction $main$. Nous devons créer une bibliothèque liée statiquement qui rend possible de se référer à cette fonction à partir du PPU en reliant le code PPU à cette bibliothèque. Pour ce faire, on utilise la commande suivante :

```
/opt/IBM/toolchain-3.2/ppu/bin/ppu-embedspu -m32 hello_spu hello_spu-embed.o
```

Nous avons presque terminé : maintenant que nous avons le code pour le PPU, il suffit de le compiler. Le lancement des threads sur le SPU consiste simplement à exécuter le code PPU. La procédure de compilation est plus ou moins équivalente à :

```
/opt/IBM/toolchain-3.2/ppu/bin/ppu32-gcc -c hello-ppu.c

/opt/IBM/toolchain-3.2/ppu/bin/ppu32-gcc -o test hello-ppu.o ...
                    spu/hello_spu-embed.o -lspe
```

La bibliothèque de linkage *libspe* contient l'ensemble des fonctions offertes au PPU, afin de pouvoir gérer ce qui se passe sur le SPU, comme par exemple l'utilisation de la fonction *spe_create_thread* qui permet la création et le lancement des threads sur les SPU. On peut éventuellement résumer tout ce processus de compilation par l'image suivante B.4.

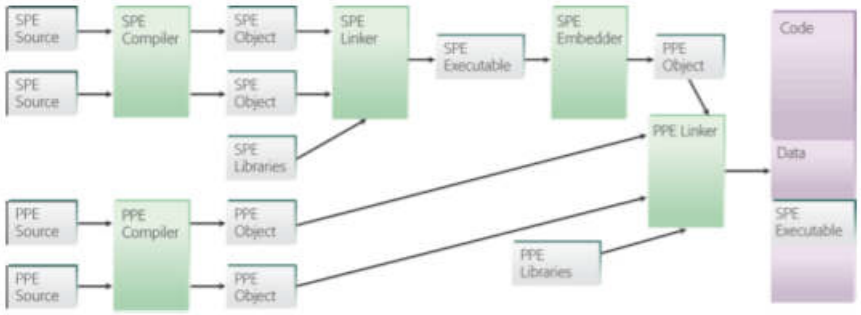


FIGURE B.4 – Processus de compilation pour le Cell BE.

TABLE DES FIGURES

LISTE DES TABLEAUX

BIBLIOGRAPHIE

[1] The BlueGene/L Team. An overview of the bluegene/l supercomputer. *Procceding of ACM Supercomputing Conference*, 2002.

[2] D.E Culler T Anderson and David Patterson. A case for networks of workstations :now. In *In IEEE Micro*, 1995.

[3] R. Felderman A. Kulawik C. Sietz J. Seizovic N. Boden, D. Cohen and W. Su. Myrinet a gigabit-per-second local-area network. *IEEE Micro*, pages 29–36, 1995.

[4] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. *Supercomputing '00 : Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 12, 2000.

[5] Khaled Hamidouche, Franck Cappello, and Daniel Etiemble. Comparaison de mpi, openmp et mpi+openmp sur un nœud multiprocesseur multic$\frac{1}{2}$urs amd à mémoire partagée. *Rencontres francophones du Parallélisme (RenPar'19), Toulouse, France*, page 8, 2009.

[6] Igor N. Kozin. Evaluation of clearspeed accelerators for hpc. *Technical Report DL-TR-2009-001*, pages 1–15, 2009.

[7] Nvidia Corporation. Cuda c programming guide version 4.0. http ://developer.download.nvidia.com/compute/DevZone/docs/html /C/doc/-CUDA_C_Programming_Guide.pdf.

[8] Xue-Jun Yang, Xiang-Ke Liao, Kai Lu, Qing-Feng Hu, Jun-Qiang Song, and Jin-Shu Su. The tianhe-1a supercomputer : Its hardware and software. *Journal of Computer Science and Technology*, pages 344–351, 2010.

[9] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, andM. Pham Y. Masubuchi, J. Pille, and K Yazawa. Overview of the architecture, circuit design, and physical implementation of a ?rst-generation cell processor. *IEEE Journal of Solid-State Circuits*, page 179 ?196, 2006.

[10] Tarik Saidani, Lionel Lacassagne, Joel Falcou, Claude Tadonki, and Samir Bouaziz. Parallelization schemes for memory optimization on the cell processor : a case study on the harris corner detector. *In Book :Transactions on high-performance embedded architectures and compilers III*, 2011.

[11] Ovidiu GHERMAN, Ioan UNGUREAN, tefan Gheorghe PENTIUC, and Oana VULTUR. Data communications in an hpc hybrid cluster and performance evaluation. *10th International Conference on DEVELOPMENT AND APPLICATION SYSTEMS*, page 175 ?178, 2010.

[12] MPI Forum. Mpi : A message-passing interface standard. *International journal of Supercomputer Application*, pages 8 :165–416, 1994.

[13] Jack Dongarra W. Jiang R Manchek A Geist, A Beguelin and V. Sundream. Pvm parallel virtual machine, a user's guide ans tutorial for networked parallel computing. *MIT press, Combridge*, 1994.

[14] Mickael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computing*, 21(9) :948–963, 1972.

[15] T.C. Chiang. Multicore parallel computing with openmp. *SVU/ Academic Computing, Computer Centre*, 2007.

[16] R. Noraonha and D.K. Panda. Improving scalability of openmp applications on multicore systems using large page support. *IEEE, International Parallel and Distributed Processing Symposium*, page 8, 2007.

[17] Géraud Krawezik and Franck Cappello. Performence Comparaison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors. *ACM Symposium on Parallel Algorithms*, pages 118–127, 2003.

[18] Alan J. Wallcraft. SPMD OpenMP versus MPI for Ocean Models. *concurrency : Practice and Experience*, 12 :1155–1164, 2000.

[19] Nikolaos Drosinos and Nectarios Koziris. Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters. *Parallel and Distributed Processing Symposium, International*, 1 :15a, 2004.

[20] L Giraud. Combining Shared and Distributed Memory Programming Models on Clusters of Symmetric Multiprocessors : Some Basic Promising Experiments. *International Journal of High Performance Computing Applications*, 16 :425–430, 2002.

[21] A. E. Eichenberger, J. K. O ?Brien, K. M. O ?Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Syst. J., 45*, page 59 ?84, 2006.

[22] P. Bellens, J. M. Peres, R. M. Badia, and J. Labarta. Cellss : A programming model for the cell be architecture. In *Proc. of the 2006 Supercomputing Conference (SC06)*, 2006.

[23] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband enginetm processor. *IBM Syst. J., 45*, page 85 ?102, 2006.

[24] Khronos OpenCL Working Group. The opencl specification version 1.0. http ://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf.

[25] Cedric Augonnet. Starpu : un support executif uni ?é pour les architectures multic$\frac{1}{2}$urs hétérogènes. *Rencontre francophone pour le parallelisme, Renpar19*, pages 1–8, 2009.

[26] L.V. Kale. Performance and productivity in parallel programming via processor virtualization. *In Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, 2004.

[27] D. M. Kunzman and L. K. Kale. Towards a framework for abstracting accelerators in parallel applications : Experience with cell. In *Proc. of the 2009 Supercomputing Conference (SC09)*, 2009.

[28] K. Fatahalian et al. Sequoia : Programming the memory hierarchy. In *Proc. of the 2006 Supercomputing Conference (SC06)*, 2006.

[29] Barcelona Supercomputing Center Team. Smp superscalaire (smps) user's manuel. July 2007.

[30] M. Sato, T. Boku, M. Yokokawa, and H. Iwashita. Xcalablemp application program interface. *XcalableMP Speci ?cation Working Group*, November 2009.

[31] Beatrice Creusillet and HPC Project. Automatic task generation on the scmp architecture for data ?ow applications. *Progress report, SCALOPES Project*, page 17, 2011.

[32] The Portland Group. Pgi accelerator programming model for fortran c.

[33] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys (CSUR)*, pages 123–169, 1998.

[34] M. Cole. Algorithmic skeletons : Structured management of parallel computation. *MIT Press*, 1989.

[35] M. Cole. Bringing skeletons out of the closet : A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, page 389 ?406, 2004.

[36] W. Dosch. A pair-based functional skeleton and its list homomorphic implementation. *Tenth International Conference on Parallel and Distributed Computing and Systems (PDCS ?98)*, page 561 ?567, 1998.

[37] J. Sérot and D. Ginhac. Skeletons for parallel image processing : an overview of the skipper project. *Parallel Computing*, page 1785 ?1808, 2002.

[38] B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto, S. Pelagatti, and M. Vanneschi. An anvironment for structured parallel programming. *Advances in High Performance Computing*, page 219 ?234, 1997.

[39] S. Pelagatti. A methodology for the development and the support of massively paral lel programs. *PhD thesis, Dipartimento di Informatica*, 1993.

[40] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3l : A structured high level programming language and its structured support. *Concurrency : Practice and Experience*, page 225 ?255, 1995.

[41] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. *ACM,Proceedings of the 1st international conference on Scalable information systems*, page 13, 2006.

[42] Z. Hu K. Matsuzaki and M. Takeichi. Parallelization with tree skeletons. *In Euro-Par, volume 2790 of Lecture Notes in Computer Science*, page 789 ?798, 2003.

[43] K. Emoto, Z. Hu, K. Kakehi, , and M. Takeichi. A compositional framework for developing parallel programs on two dimensional arrays. *Technical report, Department of Mathematical Informatics, University of Tokyo*, 2005.

[44] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi. A fusion-embedded skeleton library. *In Euro-Par, volume 3149 of Lecture Notes in Computer Science*, page 644 ?653, 2004.

[45] Steven Fortune and James Wyllie. Parallelism in random access machines. *In STOC ?78 : Proceedings of the tenth annual ACM symposium on Theory of computing*, page 114 ?118, 1978.

[46] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of prams. *Theor. Comput. Sci*, page 3 ?28, 1990.

[47] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in pram computations. *In SPAA ?89 : Proceedings of the ?rst annual ACM symposium on Parallel algorithms and architectures*, page 11 ?21, 1989.

[48] Phillip B. Gibbons. A more practical pram model. *In SPAA ?89 : Proceedings of the ?rst annual ACM symposium on Parallel algorithms and architectures*, page 158 ?168, 1989.

[49] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. *In SCG ?93 : Proceedings of the ninth annual symposium on Computational geometry*, page 298 ?307, 1993.

[50] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. *International Journal of Computational Geometry and Applications*, page 379 ?400, 1996.

[51] D. E. Culler, R. M. Karp, D. A. Patterson ans A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp : Towards a realistic model of parallel computation. *In Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[52] A. Alexandrov, M. F. Ionescu, and K. E. Schauser C. Scheiman. Loggp : Incorporating long messages into the logp model - one step closer towards a realistic model for parallel computation. *Technical report : University of California at Santa Barbara Santa Barbara*, pages 1–21, 1995.

[53] L. W. Tucker and A. Mainwaring. Cmmd : Active messages on the cm-5. *Parallel Computing*, page 20, 1994.

[54] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8) :103–111, 1990.

[55] Martin Beran. Decomposable Bulk Synchronous Parallel Computers. In *SOFSEM 99 : Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*, page 349359, 1999.

[56] Jesus A Gonzalez, Coromoto Leon, Fabiana Piccoli, Marcela Printista, José Luis Roda, Casiano Rodríguez, and Francisco de Sande. Oblivious bsp. In *EuroPar 00 : Proceedings of the 6th International EURO-PAR Conference*, volume 1900, page 682685, 2000.

[57] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Marco Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, page 943 ?958, 1997.

[58] Ben H. H. Juurlink and Harry A. G. Wijshoff. The e-bsp model : Incorporating general locality and unbalanced communication into the bsp model. *Euro-Par 96 : Proceedings of the Second International Euro-Par Conference on Parallel Processing*, II, :339347, 1996.

[59] Michael T. Goodrich. Communication-ef ?cient parallel sorting. *SIAM Journal on Computing*, page 416 ?432, 2000.

[60] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib : The BSP Programming Library. *Parallel Computing*, 24(14) :1947–1980, 1998.

[61] David B. Skillicorn, Jonathan M. D. Hill, and William F. McColl. Questions and answers about bsp. *Technical Report TR-15-96, Oxford University Computing Laboratory*, 1996.

[62] Wijnand J. Suijlen. Bsponmpi. http ://bsponmpi.sourceforge.net, January 2007.

[63] Rob H. Bisseling. *Parallel Scientific Computation : A Structured Approach using BSP and MPI*. Oxford University Press, oxford university press edition, 2004.

[64] Amr Fahmy and Abdelsalam Heddaya. Communicable memory and lazy barriers for bulk synchronous parallelism in bspk. *Technical Report BUCS-TR-1996-01, Computer Science Department, Boston University*, page 10, 1996.

[65] Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, and Thanasis Tsantilas. Portable and efficient parallel computing using the bsp model. *IEEE Transactions on Computers*, page 670 ?689, 1999.

[66] Olaf Bonorden, Ben H. H. Juurlink, Ingo von Otte, , and Ingo Rieping. The Paderborn University BSP (PUB) Library. *Parallel Computing*, 29 :187–207, 2003.

[67] F. Loulergue. Implementation of a functional bulk synchronous parallel programming library. In 14*th IASTED International Conference on Parallel and Distributed Computing Systems*, pages 452–457. ACTA Press, 2002.

[68] Frédéric Loulergue Louis Gesbert, Frédéric Gava and Frédéric Dabrowski. Bulk Synchronous Parallel ML with Exceptions. *Future Generation Computer Systems*, 26 :486–490, 2010.

[69] D. Rémy. Using, understanding, and unravellling the ocaml language. *Applied Semantics, number 2395 in LNCS*, page 413 ?536, 2002.

[70] G. Hains and F. Loulergue. Functional bulk synchronous parallel programming using the bsmllib library. *In S. Gorlatch and C. Lengauer, editors, Constructive Methods for Parallel Programming, Advances in Computation : Theory and Practice*, page 165 ?178, 2002.

[71] F. Loulergue. Distributed evaluation of functional bsp programs. *Parallel Processing Letters*, page 423 ?437, 2001.

[72] K. Hamidouche, J. Falcou, and D. Etiemble. Hybrid bulk synchronous parallelism library for clustered smp architectures. In *Proceedings of the 4th Int. Workshop on High-level Parallel Programming and Applications (HLPP 2010)*, 2010.

[73] Dan Marsden and Joel de Guzman. The BOOST Phoenix Library. http ://spirit.sourceforge.net/, 2006.

[74] Jaakko Järvi and John Freeman. Lambda Functions for C++0x. In *SAC '08 : Proceedings of the 2008 ACM symposium on Applied computing*, pages 178–183, New York, NY, USA, 2008. ACM. Best paper award (Software theme).

[75] C. Tadonki, L. Lacassagne, T. Saidani, J. Falcou, and K. Hamidouche. The harris algorithm revisited on the cell processor. *International workshop on highly-Efficient Accelerators and Reconfigurable Technologies, HEART 2010*, page 8, 2010.

[76] Cappello Franck, Desprez Frederic, and Margery David. Grid5000. https ://www.grid5000.fr/mediawiki/index.php/Grid5000 :Home, January 2010.

[77] Steve W Bova, Clay Breshears, H Gabb, B Kuhn, B Magro, Rudolf Eigenmann, Greg Gaertner, Stefano Salvini, and H Scott. Parallel programming with message passing and directives. *Computing in science and engineering.*, pages 22–37, 2001.

[78] C. J. Noble I.J. Bush and R. J. Allan. Mixed OpenMP and for MPI parallel fortran applications. *In European workshop on OpenMP (EWOMP 2000)*, Edinburgh, UK, 2000.

[79] E Chow and Hysom. Assessing performance of hybrid mpi/openmp programs on smp cluster. *technical report UCRL-JC*, 2001.

[80] K. Hamidouche, J. Falcou, and D. Etiemble. A framework for an automatic hybrid mpi+openmp code generation. *ACM High Performance Computing Symposium , HPC-11*, page 10, 2011.

[81] Nicholas Nethercote and Julian Seward. Valgrind : a framework for heavyweight dynamic binary instrumentation. *PLDI '07 Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007.

[82] Bernd Mohr, Allen D. Malony, Janice E. Cuny, and Bernd Mohr Allen D. Malony. Tau - tuning and analysis utilities for portable parallel programming. 1995.

[83] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi : A portable interface to hardware performance counters. *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999.

[84] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The vampir performance analysis tool-set. *Book : Tools for High Performance Computing*, 2008.

[85] Jean-Christophe Beyler and Philippe Clauss. Performance driven data cache prefetching in a dynamic software optimization system. *ACM International Conference on Supercomputing, ICS'07*, page 8, 2007.

[86] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model : Part I, one-dimensional time. *ACM International Conference on Code Generation and Optimization (CGO'07)*, pages 144–156, 2007.

[87] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. *In Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, pages 185–201, 2006.

[88] Ph. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. *IEEE Int. Conf. on Application Specific Array Processors, (ASAP'96)*, 1996.

[89] C Cascaval, L DeRose, D Padua, and A Reed. Compile-time based performance prediction. *International Workshop on Languages and Compilers for Parallel Computing*, 1999.

[90] A Snavely, L Carrington, N Wolter, Badia Labartaand, and A Purkayastha. A framework for performance modeling and prediction. *ACM/IEEE conference on Supercomputing*, 2002.

[91] Ler Adhianto and Barbara Chapman. Performance modeling of communication and computation in hybrid mpi and openmp application. *Parallel and Distributed systems ICPADS*, page 6, 2006.

[92] Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen, and Weimin Zheng. Openuh : an optimizing, portable openmp compiler. *Concurrency and Computation : Practice Experience - Current Trends in Compilers for Parallel Computers*, pages 1–15, 2007.

[93] C Liao and Barbara Chapman. Invited paper : A compile-time cost model for openmp. *Parallel and distributed Processing symposium*, 2007.

[94] R Reyes, A.J Dorta, F Almeida, and F Sande. Automatic hybrid mpi+openmp code generation with llc. *European PVM/MPI*, pages 185–195, 2009.

[95] Clang. *http ://clang.llvm.org/.*

[96] Chris Lattner and Vikram Adve. Llvm : A compilation framework for lifelong program analysis transformation. *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[97] Llvm. *http ://llvm.org/docs/WritingAnLLVMPass.htmlFunctionPass*.

[98] Vincent Loechner, Benoît Meister, and Philippe Clauss. Data sequence locality : A generalization of temporal locality. *Proceedings of the 7th International Euro-Par, Euro-Par '01*, page 8, 2001.

[99] Sphinx. *http ://www.llnl.gov/casc/sphinx/sphinx.html.*

[100] Bronis R. de Supinski. The asci pse milepost : Run-time systems performance tests. *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2001.

[101] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proceedings of the 5th Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 73–84, 2004.

[102] Richard Lassaigne and Sylvain Peyronnet. Probabilistic verification and approximation. *Annals of Pure and Applied Logic*, 152(1-3) :122 – 131, 2008.

[103] Håkan L. S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *CAV*, pages 223–235, 2002.

[104] Radu Grosu and Scott A. Smolka. Monte Carlo model checking. In *TACAS*, pages 271–286, 2005.

[105] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *CAV*, pages 202–215, 2004.

[106] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM (JACM)*, 42(4) :857–907, 1995.

[107] R. M. Karp and M. Luby. Monte-Carlo algorithms for enumeration and reliability problems. In *FOCS*, pages 56–64, 1983.

[108] Thomas Hérault, Richard Lassaigne, and Sylvain Peyronnet. APMC 3.0 : Approximate verification of discrete and continuous time Markov chains. In *QEST*, pages 129–130, 2006.

[109] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM : A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[110] Amir Pnueli and Lenore D. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1) :53–72, 1986.

[111] Akim Demaille, Sylvain Peyronnet, and Benoît Sigoure. Modeling of sensor networks using XRM. In *ISoLA*, pages 271–276, 2006.

[112] K. Hamidouche, A. Borghi, P. Esterie, J. Falcou, and S. Peyronnet. Three high performance architectures in the parallel apmc boat. *IEEE International Workshop on Parallel and Distributed Methods in Verification, PDMC 2010*, page 8, 2010.

[113] David Mount. *Bioinformatics : Sequences and Genome Alignment*. Cold Spring Harbor Laboratory Press, New York, 2004.

[114] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal Mol. Biol.*, 147 :195–197, 1981.

[115] Gregory Pfister. *In Search of Clusters : the Coming Battle for Lowly Parallel Computing*. Prentice Hall, 1997.

[116] K. Hamidouche, F. M. Mendonca, J. Falcou, and D. Etiemble. Parallel biological sequence comparison on heterogeneous high performance computing platforms with bsp++. *23rd IEEE International Symposium on Computer Architecture and High Performance Computing - SBAC-PAD'201*, page 8, 2011.

[117] A. Aji and W. Feng. Optimizing performance, cost, and sensitivity in pairwise sequence search on a cluster of playstation. In *Proceedings of the 8th IEEE International Conference on BioInformatics and BioEngineering*, 2008.

[118] M. Noorian, H. Pooshfam, Z. Noorian, and R. Abdulla. Performance enhancement of smith-waterman algorithm using hybrid model : Comparing the mpi and hybrid programming paradigm on smp clusters. In *Proceedings of the 2009 IEEE Int. Conf. on Systems, Man, and Cybernetics*, 2009.

[119] A. Aji, W. Feng, F. Blagojevic, and D. Nikolopoulos. Cell-swat : Modeling and scheduling wavefront computations on the cell broadband engine. In *Proceedings of the Computing Frontiers Conference*, 2008.

[120] A. Szalkowski, C. Ledergerber, P. Krahenbuhl, and C. Dessimoz. Swps3 - fast multi-threaded vectorized smith-waterman for ibm cellb.e. and x86sse2. *BMC Research Notes*, 1 :107–110, October 2008.

[121] V. Sachdeva, M. Kistler, E. Speight, and T. Tzeng. Exploring the viability of the cell broadband engine for bioinformatics applications. *Parallel Computing*, 34(11) :616–626, November 2008.

[122] A. Wirawan, B. Schmidt, H. Zhang, and C. Kwoh. High performance protein sequence database scanning on the cell broadband engine. *Scientific Computing*, 17 :97–111, 2009.

[123] Y. Song, G. Striemer, and A. Akoglu. Performance analysis of ibm cell broadband engine on sequence alignment. In *Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 439–446, 2009.

[124] A. Sarje and S. Aluru. Parallel genomic alignments on the cell broadband engine. *IEEE Transactions on Parallel and Distributed Systems*, 20(11) :1600–1610, November 2009.

[125] A. Boukerche, R. B. Batista, and A. C. M. A. Melo. Exact pairwise alignment of megabase genome biological sequences using a novel z-align pa-

rallel strategy. In *Proceedings of the 2009 IPDPS Workshop on Nature-Inspired Distributed Systems*, 2009.

[126] C. Chen and B. Schmidt. Computing large-scale alignments on a multi-cluster. In *Proceedings of the IEEE Int. Conference on Cluster Computing*, pages 38–45, 2003.

[127] S. Rajko and S. Aluru. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, 15(12) :1070–1081, December 2004.

[128] C. Harris and M. Stephens. A combined corner and edge detector. *4th ALVEY Vision Conference*, 1998.

[129] Hans Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. *In tech. report CMU-RI-TR-80-03, Robotics Institute, Carnegie Mellon University doctoral dissertation, Stanford University*, 1980.

[130] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. Stapl : An Adaptive, Generic Parallel C++ Library,. In Springer Verlag, editor, *Proceedings of Workshop on Language and Compiler for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science*, pages 195–210, 2003.

[131] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp : A hybrid multi-core parallel programming environment. *First Workshop on General Purpose Processing on Graphics Processing Units (2007)*, pages 1–5, 2007.