Optimization of memory management on distributed machine

Viet Hai Ha

THESE DE DOCTORAT CONJOINT TELECOM SUDPARIS
et L'UNIVERSITE PIERRE ET MARIE CURIE

Spécialité : Informatique

Ecole doctorale : Informatique, Télécommunications et Electronique de Paris

Présentée par

Viet Hai HA

Pour obtenir le grade de
DOCTEUR DE TELECOM SUDPARIS

Optimisation de la gestion mémoire sur machines distribuées

Soutenue le 05 Octobre 2012

devant le jury composé de :

Directeur de thèse :	
Éric RENAULT	Maître de conférence (HDR) Institut Mines – Télécom / Télécom SudParis
Rapporteurs	
M. Pierre BOULET	Professeur Université de Lille 1
M. Sébastien LIMET	Professeur Université d'Orléans
Examinateurs	
M. Jean-Luc LAMOTTE	Professeur Unversité Pierre et Marie Curie
M. Christophe CÉRIN	Professeur Unversité Paris 13
Mme. Selma BOUMERDASSI	Maître de conférence (HDR) CNAM

Thèse n° 2012TELE0042

Acknowledgements

I would like to express my gratitude, appreciation and sincere thanks to my supervisor Dr. Éric Renault for his excellent guide, useful discussions, theoretical and technical help, and for his continuous encouragement during the whole period of my work.

I am grateful to Mmes Xayplathi Lyfoung and Valérie Mateus and the other members of TMSP for their helps for administrative procedures.

I am deeply indebted to the Vietnamese government for the scholarship to study abroad. I also thank TMSP for the financial support of and the working conditions.

I would like to thank all my friends at TMSP for their support and the sharing of life's experiments.

Finally, I deeply thank the constant encouragement and support of my whole family including my parents, my sisters and my brothers, my wife and two my children. I dedicate this thesis to all the members of my family.

Abstract

In order to explore further the capabilities of parallel computing architectures such as grids, clusters, multi-processors and more recently, clouds and multi-cores, an easy-to-use parallel language is an important challenging issue. From the programmer's point of view, OpenMP is very easy to use with its ability to support incremental parallelization, features for dynamically setting the number of threads and scheduling strategies. However, as initially designed for shared memory systems, OpenMP is usually limited on distributed memory systems to intra-nodes' computations.

Many attempts have tried to port OpenMP on distributed systems. The most emerged approaches mainly focus on exploiting the capabilities of a special network architecture and therefore cannot provide an open solution. Others are based on an already available software solution such as DMS, MPI or Global Array and, as a consequence, they meet difficulties to become a fully-compliant and high-performance implementation of OpenMP.

As yet another attempt to built an OpenMP compliant implementation for distributed memory systems, CAPE − which stands for Checkpointing Aide Parallel Execution − has been developed which with the following idea: when reaching a parallel section, the master thread is dumped and its image is sent to slaves; then, each slave executes a different thread; at the end of the parallel section, slave threads extract and return to the master thread the list of all modifications that has been locally performed; the master includes these modifications and resumes its execution.

In order to prove the feasibility of this paradigm, the first version of CAPE was implemented using complete checkpoints. However, preliminary analysis showed that the large amount of data transferred between threads and the extraction of the list of modifications from complete checkpoints lead to weak performance. Furthermore, this version was restricted to parallel problems satisfying the Bernstein's conditions, i.e. it did not solve the requirements of shared data.

This thesis aims at presenting the approaches we proposed to improve CAPE' performance and to overcome the restrictions on shared data. First, we developed DICKPT which stands for Discontinuous Incremental Checkpointing, an incremental checkpointing technique that supports the ability to save incremental checkpoints discontinuously during the execution of a process. Based on the DICKPT, the execution speed of the new version of CAPE was significantly increased. For example, the time to compute a large matrix-matrix product on a desktop cluster has become very similar to the execution time of the same optimized MPI program. Moreover, the speedup associated with this new version for various number of threads is quite linear for different problem sizes.

In the side of shared data, we proposed UHLRC, which stands for Updated Home-based Lazy Release Consistency, a modified version of the Home-based Lazy Release Consistency (HLRC) memory model, to make it more appropriate to the characteristics of CAPE. Prototypes and algorithms to implement the synchronization and OpenMP data-sharing clauses and directives are also specified. These two works ensures the ability for CAPE to respect shared-data behavior.

Publications relevant to the thesis

1. Viet Hai Ha and Éric Renault. Discontinuous Incremental: *A New Approach Towards Extremely Lightweight Checkpoints*. Proceedings of the IEEE International Symposium on Computer Networks and Distributed Systems 2011 (CNDS 2011), Tehran, Iran, February 2011.

2. Viet Hai Ha and Éric Renault. *Design and Performance Analysis of CAPE based on Discontinuous Incremental Checkpoints*. Proceedings of the IEEE Conference on Communications, Computers and Signal Processing (PacRim 2011). Victoria, Canada, August 2011.

3. Viet Hai Ha and Éric Renault. *Improving Performance of CAPE using Discontinuous Incremental Checkpointing*. Proceedings of the IEEE International Conference on High Performance and Communications 2011 (HPCC-2011). Banff, Canada, September 2011.

4. Viet Hai Ha and Éric Renault. *Towards an efficient implementation of OpenMP on Distributed Memory Systems using CAPE with Discontinuous Incremental Checkpointing*. Poster in the Early Adopters Ph.D. Workshop: Building the Next Generation of Application Scientists, a part of the IEEE and ACM SuperComputing Conference 2011 (SC11). Seattle, USA, November 2011.

5. Viet Hai Ha and Éric Renault. *Design of a shared-memory model for CAPE*. Proceeding of the 8th International Workshop on OpenMP (IWOMP 2012), Rome, Italy, Springer LNCS 7321, pp. 262-266, June 2012.

Résumé

Afin d'exploiter les capacités des architectures parallèles telles que les grappes, les grilles, les systèmes multi-processeurs, et plus récemment les nuages et les systèmes multi-cœurs, un langage de programmation universel et facile à utiliser reste à développer. Du point de vue du programmeur, OpenMP [1] est très facile à utiliser en grande partie grâce à sa capacité à supporter une parallélisation incrémentale, la possibilité de définir dynamiquement le nombre de fils d'exécution, et aussi grâce à ses stratégies d'ordonnancement. Cependant, comme il a été initialement conçu pour des systèmes à mémoire partagée, OpenMP est généralement très limité pour effectuer des calculs sur des systèmes à mémoire distribuée.

De nombreuses solutions ont été essayées pour faire tourner OpenMP sur des systèmes à mémoire distribuée. Les approches les plus abouties se concentrent sur l'exploitation d'une architecture réseau spéciale et donc ne peuvent fournir une solution ouverte. D'autres sont basées sur une solution logicielle déjà disponible telle que DMS [6][7], MPI [8][9] ou Global Array [10], et par conséquent rencontrent des difficultés pour fournir une implémentation d'OpenMP complètement conforme et à haute performance.

CAPE — pour *Checkpointing Aided Parallel Execution* — est une solution alternative permettant de développer une implémentation conforme d'OpenMP pour les systèmes à mémoire distribuée. L'idée est la suivante : en arrivant à une section parallèle, l'image du *thread* maître est sauvegardé et est envoyée aux esclaves ; puis, chaque esclave exécute l'un des *threads* ; à la fin de la section parallèle, chaque *threads* esclaves extraient une liste de toutes modifications ayant été effectuées localement et la renvoie au *thread* maître ; le *thread* maître intègre ces modifications et reprend son exécution.

Afin de prouver la faisabilité de cette approche, la première version de CAPE (CAPE-1) [12][13][14] a été implémentée en utilisant des points de reprise complets. Cependant, une analyse préliminaire a montré que la grande quantité de données transmises entre les *threads* et l'extraction de la liste des modifications depuis les points de reprise complets conduit à de faibles performances. De plus, cette version est limitée à des problèmes parallèles satisfaisant les conditions de Bernstein, autrement dit, il ne permet pas de prendre en compte les données partagées.

L'objectif de cette thèse est de proposer de nouvelles approches pour améliorer les performances de CAPE et dépasser les restrictions sur les données partagées.

2. CAPE basé sur des points de reprise complets (CAPE-1)

2.1. Principe

L'idée initiale derrière CAPE est l'utilisation de points de reprise pour implémenter le modèle d'exécution *fork-join* d'OpenMP.

Dans la première version, CAPE est limité à des programmes contenant des régions parallèles satisfaisant les conditions de Bernstein, et les *threads* sont remplacés par des processus afin d'être distribués sur des machines différentes. Le principe de CAPE est le suivant :

Au début, le programme est exécuté dans un seul processus --- le processus maître. Lorsque ce processus rencontre une région parallèle :

- phase *fork* : le travail est divisé en parties plus petites par le processus maître. Au début de chaque morceau, le processus maître crée un point de reprise complet et l'envoie à une machine distante pour créer un processus esclave. Ensuite, le processus maître continue avec la partie suivante s'il y a lieu. Après le dernier morceau, le processus maître attend les résultats des processus esclaves. Chaque machine distante utilise le point de reprise qu'elle a reçu pour reprendre le programme au début de sa partie. Une fois la partie terminée, le résultat est extrait en utilisant un autre point de reprise et la différence d'avec le point de reprise initial est retourné au processus maître. Enfin, le processus esclave termine son exécution ;
- phase *join* : après que tous les processus esclaves ont terminé leur exécution et que le processus maître a reçu tous les résultats des processus esclaves, le processus maître reprend son exécution à la région suivante du programme. À ce moment-là, il n'y a qu'un seul processus actif dans le système.

Le modèle CAPE offre deux avantages principaux :

- il est compatible avec le modèle de mémoire partagée *relaxed consistency* d'OpenMP : les mémoires du processus maître et des processus esclaves correspondent respectivement aux mémoires et aux vues temporaires du *thread* maître et des *threads* esclaves du modèle d'OpenMP. Lors de l'exécution de chaque partie, les processus esclaves n'utilisent que leur espace mémoire propre. En conséquence, ce modèle permet de réduire le temps d'accès à la mémoire commune et donc d'augmenter la vitesse d'exécution du programme. À l'exécution, la synchronisation des mémoires des processus esclaves n'est pas nécessaire puisque les conditions de Bernstein doivent être satisfaites. La solution permettant de surmonter cette restriction est présentée en Sec. 5 ;
- la possibilité de distribuer automatiquement les régions parallèles et de recueillir automatiquement le résultat des processus esclaves (à partir des points de reprise) permet de dépasser les inconvénients des autres approches, comme Cluster OpenMP d'Intel ou les solutions basées sur MPI ou Global Array. En spécifiant les zones mémoires modifiées lors de l'exécution d'une section du programme, deux éléments importants du modèle *fork-join* peuvent être réalisés automatiquement, et ce sans avoir besoin d'utiliser des directives supplémentaires. Le premier est la distribution du travail aux processus esclaves ; le second est la récupération des résultats d'exécution dans les processus esclaves. Ces deux éléments sont très difficiles à réaliser avec les autres approches visant à l'exécution de programmes OpenMP sur des systèmes à mémoire partagée, qui de fait ne peuvent être considérées comme des solutions entièrement conforme et à haute performance.

2.2. Prototype CAPE pour les boucles `parallel for`

La Fig. 1 présente la traduction effectuée sur un code incluant une construction OpenMP `parallel for`, avec toutes les itérations de la boucle `D` satisfaisant les conditions de Bernstein. Pour faciliter la présentation, supposons que le nombre d'itérations de la boucle soit égal au nombre de processus esclaves. Le parent, c'est-à-dire le nœud maître, a la charge de la gestion des esclaves et n'exécute aucune itération de la boucle dans la partie parallèle. Toutefois, ceci n'est pas obligatoire et le nœud maître peut prendre part à l'exécution d'une ou plusieurs itérations de

boucle. La traduction est basée sur les fonctions suivantes :

- `create ( file )` crée un point de reprise complet et l'enregistre dans le fichier *`file`*. La valeur retournée par la fonction est utilisée pour déterminer si la fonction vient de créer le point de reprise ou si elle retourne après la reprise de l'exécution à partir du point de reprise. Cette fonction est très similaire à l'appel système `fork`, sauf que `create` renvoie `TRUE` après avoir généré le point de reprise et `FALSE` après la reprise de l'exécution du point de reprise ;

```
# pragma omp parallel for
    for ( A ; B ; C )
        D
```

↓ automatically translated into ↓

```
parent = create ( original )
if ( ! parent )
    exit
copy ( original, target )
for ( A ; B ; C )
    parent = create ( before_i )
    if ( parent )
        ssh host_x restart ( before_i )
    else
        D
        parent = create ( after_i )
        if ( ! parent )
            exit
        diff ( before_i, after_i, delta_i )
        merge ( target, delta_i )
        exit
parent = create ( final )
if ( parent )
    diff ( original, final, delta )
    wait_for ( target )
    merge ( target, delta )
    restart ( target )
```

Figure 1. Modèle pour les boucles `parallel for` d'OpenMP avec des points de reprise complets.

- `copy ( file_1 , file_2 )` copie le contenu de *`file_1`* dans *`file_2`* ;
- `diff ( file_1 , file_2 , file_3 )` enregistre dans *`file_3`* la liste des modifications qui devraient être appliquées sur *`file_1`* pour obtenir

file_2 ;

- `merge ( file_1 , file_2 )` applique la liste des modifications enregistrées dans *file_2* au point de reprise contenu dans *file_1* ;
- `wait_for ( file )` retourne après que toutes les modifications à effectuer sur le fichier *file* sont faites ;
- `restart ( file )` reprend l'exécution du processus en cours à partir du point de reprise contenu dans *file*.

L'opération qui consiste à reprendre l'exécution des points de reprise générés pour chaque itération de boucle (la ligne en italique dans la Fig. 1) est exécutée sur le nœud maître, mais déléguée à un processus externe en charge de la gestion de la distribution des processus sur un ensemble de ressources distantes. BOINC, utilisé dans le cadre du projet Seti@Home, est probablement l'un des outils les plus célèbres visant à distribuer des travaux sur un ensemble de ressources. Cependant, de nombreuses autres solutions comme AC2, Nimbus, OpenNebula et KOALA, sont également disponibles.

2.3. Certaines améliorations possibles pour CAPE

Bien que la faisabilité et la cohérence de CAPE aient été prouvées à la fois de manière théorique et par une implémentation, ce prototype n'est pas optimal à cause de certains éléments réduisant significativement la performance globale et limitant de fait ses applications. Les problèmes les plus importants sont les suivants :

1. *une trop grande quantité de données transmises sur le réseau*. Le processus maître a besoin d'envoyer son image sous la forme d'un point de reprise complet à chaque nœud esclave. Cette transmission conduit à deux éléments importants qui augmentent la quantité de données envoyées sur le réseau : le premier est que ces instantanés sont grands parce qu'ils contiennent toutes les informations liées au processus, dont les données du programme, mais surtout le code du programme, celui des bibliothèques qu'elles soient partagées ou non, etc. ces derniers éléments étant invariants pendant toute la durée de vie du programme ; le second est qu'ils sont différents pour chaque itération de boucle, et par conséquent que le processus maître doit les créer et les envoyer séquentiellement aux processus esclaves ;
2. *le trop grand nombre de comparaisons sur les points de reprise complets afin d'extraire le résultat d'exécution sur les nœuds esclaves*. Ceci devient un point critique lorsque les points de reprise sont grands alors que la taille du résultat final est petite. Dans ce cas, un grand nombre de comparaisons sont effectuées ce qui augmente significativement le temps d'exécution sur les nœuds esclaves et donc réduit la performance globale ;

- *une trop grande quantité de mémoire nécessaire au stockage des données temporaires*. Dans ce prototype, plusieurs points de reprise ont besoin d'être sauvegardés (`original`, `target`, `before`). Lorsque la taille de ces points de reprise est grande, ceci affecte significativement l'utilisation de la mémoire ;

3. *un délai pour démarrer/reprendre les processus trop important.* Les processus esclaves et le processus maître doivent démarrer/reprendre leur exécution pour chaque région parallèle. Les premiers ont lieu lors du lancement des processus esclaves à partir des points de reprise. Le second a lieu sur le processus maître après avoir reçu les résultats des processus

esclaves, afin d'inclure tous ces résultats dans son espace d'adressage virtuel. La possibilité de mettre à jour directement la mémoire d'un processus à partir des informations contenues dans les points de reprise permettrait d'éviter cet inconvénient ;

- *l'obligation de satisfaire les conditions de Bernstein*. OpenMP autorise la synchronisation de la mémoire entre les différents *threads* au sein des régions parallèles. Avec CAPE-1, aucun mécanisme ne permet une telle implémentation. De plus, les directives et clauses d'OpenMP associées aux variables partagées, telles que `private`, `threadprivate`, `firtsprivate`, `lastprivate`, `réduction`, etc. ne sont pas considérées dans ce prototype. Ainsi, ces deux restrictions limitent de manière importante les programmes OpenMP pouvant être exécutés avec CAPE.

3. Points de reprise incrémentaux discontinus (*Discontinuous Incremental ChecKPoinTing* — DICKPT)

3.1. Principe

La technique des points de reprise incrémentaux consiste à stocker dans un fichier les parties de la mémoire qui ont été mises à jour depuis le début de l'exécution du processus ou depuis le dernier point de prise. D'un point de vue pratique, ceci peut être réalisé en rendant les pages mémoires accessibles en lecture seulement, afin de forcer le système à délivrer un signal SIGSEGV au processus chaque fois qu'une page est accédée en écriture. À la réception de ce signal, une copie de la page est sauvegardée dans un tampon et les droits d'accès à cette page sont restaurés à leur valeur initiale. Lorsqu'un point de prise est sauvegardé, le contenu de toutes les pages modifiées sont comparées à celles sauvegardées dans le tampon et les différences sont enregistrées dans le fichier de point de reprise.

L'idée principale associée à la technique des points de reprise incrémentaux discontinus (DICKPT) est la spécification des régions du programme devant être prises en compte dans le point de reprise. Ces informations sont fournies en utilisant des directives pragma et peuvent être implémentées en utilisant différents mécanismes tels que les signaux, les points d'arrêt, etc. Trois directives pragma ont été définies :

- `pragma dickpt start` : démarrer un point de reprise, c'est-à-dire initialiser le tampon des points de reprise ;
- `pragma dickpt stop` : suspendre le point de reprise, c'est-à-dire effacer le tampon des points de reprise ;
- `pragma dickpt save` *`filename`* : sauvegarder le point de reprise dans le fichier *`filename`* et réinitialiser le tampon des points de reprise.

3.2. Évaluation des performances

Pour comparer les performances de cette nouvelle approche avec celle des points de reprise incrémentaux normaux, nous avons mesuré leur effet sur un programme de calcul des éléments successifs d'une chaîne de Markov, Cf Fig. 2. Deux cas ont été testés : avec le premier, utilisant la technique des points de reprise incrémentaux normaux, un premier point de reprise est effectué au point 0 (ligne 8), puis un autre au point 1 (ligne 21) ; le second, lié à la technique DICKPT, évite le point de reprise au point 0 et donc le premier est effectué au point 1. Dans les deux cas, une centaine

de vecteurs d'état sont calculés au point 2 (ligne 28) et un point de reprise est généré après chaque calcul. Notre plate-forme de test est composée d'un processeur Intel Core2 Duo E8400 cadencé à 3 GHz avec 3 Go de RAM et opéré par Ubuntu 9.10 sur la base du noyau Linux 2.6.31-21-generic. Le tableau II présente les mesures de performance pour les quatre tailles de vecteurs (N égal à 3320, 6640, 9960 et 13280 éléments respectivement). Pour chaque taille du vecteur, les performances sont mesurées 30 fois (les valeurs moyennes sont fournies dans le tableau) et un intervalle de confiance d'au moins 99 % a toujours été obtenu pour les mesures. Afin d'éviter la pollution due aux accès disque sur les mesures, toutes les données (l'espace d'adressage virtuel des processus, les points de reprise, etc.) sont résidents en mémoire vive.

```
1   # include <stdio.h>
2   # include <stdlib.h>
3   # define LOOP 100
4   # define N 9960
5   # define RAND 10000
6   float M [ N ] [ N ], V [ 2 ] [ N ] ;
7   int main ( int argc, char * argv [ ] ) {
8         //location 0
9         float sum, val ;
10        int i, j, k, l ;
11        for ( i = 0 ; i < N ; i ++ ) {
12              for ( sum = j = 0 ; j < N ; j ++, sum += val )
13                    M [ i ] [ j ] = val = rand ( ) % RAND ;
14              for ( j = 0 ; j < N ; j ++ )
15                    M [ i ] [ j ] /= sum ;
16        }
17        for ( sum = i = 0 ; i < N ; i ++, sum += val )
18              V [ 0 ] [ i ] = val = rand ( ) % RAND ;
19        for ( i = 0 ; i < N ; i ++ )
20              V [ 0 ] [ i ] /= sum ;
21        //location 1
22        for ( k = l = 0 ; l < LOOP ; l ++, k = 1 - k ) {
23              for ( i = 0 ; i < N ; i ++ ) {
24                    V [ 1 - k ] [ i ] = 0.  ;
25                    for ( j = 0 ; j < N ; j ++ )
26                          V [ 1 - k ] [ i ] += ( V [ k ] [ j ] * M [ j ] [ i ] ) ;
27              }
28              //location 2
29        }
30        return 0 ;
31  }
```

Figure 2. Programme de calcul des éléments successifs d'une chaîne de Markov.

La première section du tableau II présente la taille des points de reprise au point 1 (c'est-à-dire juste après l'initialisation — comme dans le cas du checkpointeur incrémental normale) et au point 2 (soit juste après le calcul d'un nouveau vecteur). La différence entre les deux correspond à la taille de la matrice de transition qui est initialisée au début du programme. Ces données montrent la quantité d'espace mémoire pouvant être économisée en abandonnant le point de reprise au point 1 et, de la même manière, comment le point de reprise peut être plus rapidement transféré sur le réseau si nécessaire.

La deuxième section de ce même tableau indique le temps requis pour exécuter le programme sans enregistrer de point de reprise, en sauvegardant tous les

points de reprise, et en sauvegardant les points de reprise au point 2 uniquement. Elle met en évidence que le surcoût impliqué par la génération les points de reprise au point 2 est très faible (entre 1,5 % et 3,3 % pour les 100 points de reprise, soit entre 0,01 % et 0,03 % par point de reprise) par rapport au surcoût induit par la génération à la fois les points de reprise aux points 1 et 2 (le temps d'exécution total étant multiplié par 8,5).

La troisième section de ce tableau fournit le temps d'exécution pour reprendre l'exécution du processus à l'itération 50 de la boucle. Deux cas sont envisagés : le premier utilise tous les points de reprise, c'est-à-dire que le programme est redémarré, suspendu au début de la fonction principale (main), tous les points de reprise sont injectés dans le processus et l'exécution reprend à l'itération 50 de la boucle ; le second n'utilise que les points de reprise au point 2, c'est-à-dire que le programme est redémarré, suspendu après l'étape d'initialisation (point 1), tous les points de reprise au point 2 sont injectés dans le processus et l'exécution reprend à l'itération 50 de la boucle.

Les mesures de performances montrent qu'éviter le point de reprise au point 1 est toujours bénéfique.

	Taille des matrices			
	3320	6640	9960	13280
Taille des points de reprise (en MB)				
…. au point 1	42.192	168.741	379.648	674.912
…. au point 2	0.013	0.026	0.038	0.051
Temps d'exécution total (en secondes)				
…. sans générer de points de reprise	11.34	41.30	108.27	168.69
…. en générant tous les points de reprise	13.10	58.14	393.60	1433.72
…. en générant seulement les points de reprise au point 2	11.71	42.16	109.96	171.70
Temps d'exécution après redémarage à l'itération 50 (en secondes)				
…. en utilisant les points de reprise aux points 1 et 2	6.41	23.14	59.56	93.91
….en utilisant seulement les points de reprise au point 2	6.09	21.84	56.64	88.71

Tableau 1. Évaluation des performances de DICKPT.

3.3. Avantages et inconvénients

Par rapport à la technique des points de reprise incrémentaux normaux, cette nouvelle approche a les points forts et les points faibles suivant :

- *augmentation des performances*. DICKPT augmente la vitesse d'exécution du programme dans les deux périodes où des points de reprise (*checkpoints*) sont pris ou redémarrés (*recovering*). De plus, il diminue fortement la taille des points de reprise ;
- *augmentation de la flexibilité d'utilisation*. DICKPT permet de sélectionner dans le programme les régions à prendre en compte pour les points de reprise. Le cas d'utilisation des points de reprise incrémentaux normaux (et non discontinus) est obtenu en ajoutant les directives DICKPT `start` et

`stop` en première et dernière instruction respectivement du programme ;

- *ajout de deux nouvelles propriétés intéressantes dans CAPE*. La première est l'extraction des points de reprise dans les régions délimitées, pouvant être disjointes. La seconde permet d'injecter directement un point de reprise dans l'espace d'adressage du processus cible. Ces propriétés facilitent l'extraction des résultats de l'exécution sur les nœuds esclaves et l'intégration de ces résultats dans l'espace d'adressage des *threads*, sans avoir à les redémarrer. Ces deux propriétés participe à l'amélioration des performances de CAPE ;
- *modification du code source* : c'est l'inconvénient le plus important induit par l'utilisation d'un *checkpointer*. L'utilisateur doit insérer des directives dans le programme pour indiquer les régions faisant l'objet de points de reprise. Toutefois, lorsqu'un *checkpointer* est utilisé dans CAPE, cette insertion se fait de manière automatique par l'intermédiaire du compilateur de CAPE. Par conséquent, ceci est complètement transparent pour les utilisateurs de CAPE.
- *fragmentation des points de reprise* : les points de reprise n'étant pas pris d'un seul bloc (ils sont entourés par une paire `pragma dickpt start` et `pragma dickpt stop`), ils ne peuvent pas être fusionnés en un point de reprise unique. Ainsi, plusieurs fichiers sont nécessaires pour contenir les points de reprise de blocs différents. Cet inconvénient peut devenir gênant lorsque les points de reprise référencent une même zone de mémoire.

4. CAPE basé sur des points de reprise incrémentaux (CAPE-2)

À ce jour, deux versions de CAPE ont été développées. La première (CAPE-1) utilise des points de reprise complets. La seconde (CAPE-2) utilise des points de reprise incrémentaux. Pour des raisons de simplification et sauf indication contraire, dans la suite du document, CAPE fera référence à CAPE-2.

4.1. Primitives de la traduction

CAPE est basé sur un ensemble de primitives, dont certaines sont associées aux directives DICKPT présentées à la Sec. 3.1, les autres ayant été définies plus spécifiquement pour CAPE :

- `start ( )` démarre ou redémarre un point de reprise en initialisant le tampon des points de reprise. Cette directive est associée à la directive `start` de DICKPT ;
- `stop ( )` suspend un point de reprise en effaçant le tampon des points de reprise. Cette directive est associée à la directive `stop` de DICKPT ;
- `create ( file )` enregistre le contenu du tampon des points de reprise dans le fichier *`file`*, puis réinitialise le tampon des points de reprise. Cette directive est associée à la directive `create` de DICKPT ;
- `inject ( file )` met à jour le processus courant avec les informations contenues dans le point de prise *`file`*. Cette primitive diffère de la primitive originale `inject` de DICKPT, dans le sens où cette primitive ne met pas à jour le pointeur d'instruction, c'est-à-dire qu'après avoir été mis à jour, le processus continue son exécution avec l'instruction suivant l'appel la fonction ;
- `master ( )` retourne `TRUE` lorsque l'exécution s'effectue sur le processus

maître, et `FALSE` dans le cas contraire. Cette primitive peut être implémentée à partir de la fonction `get_process_num` ;

- `send ( file , node )` envoie le contenu du fichier *`file`* au nœud *`node`* ;
- `wait_for ( file )` attend que tous les éléments du fichier *`file`* aient fusionnés ;
- `last_parallel ( )` retourne `TRUE` lorsque le bloc parallèle courant est le dernier bloc parallèle du programme et `FALSE` dans le cas contraire ;
- `merge ( file_1 , file_2 )` applique la liste des modifications enregistrées dans le fichier *`file_2`* au fichier *`file_1`* ;
- `broadcast ( file )` envoie le point de reprise *`file`* à tous les nœuds esclaves. Cette fonction ne peut être exécutée que sur le nœud maître ;
- `receive ( file )` attend que le fichier *`file`* soit disponible ;
- `get_process_num ( )` retourne le numéro du processus courant ;
- `get_num_processes ( )` retourne le nombre total de processus.

4.2. Modèle pour la construction `parallel for`

La construction `parallel for` est la construction de répartition des tâches la plus complexe, mais aussi la plus utilisée. La transformation de cette construction est très générale et peut servir comme base pour la transformation des autres constructions dédiées à la répartition des tâches. Le modèle permettant de transformer cette construction vers la forme CAPE est présenté en Fig. 3. Initialement, le programme est exécuté sur tous les processus. Lorsque le programme arrive sur une région parallèle, son exécution suit le modèle *fork-join*, comme dans le modèle CAPE-1, à l'exception des trois points importants suivants : 1) le processus maître crée et envoie aux processus esclaves des points de reprise incrémentaux (lignes 4 et 5) au lieu de points de reprise complets ; 2) les résultats dans les processus esclaves sont extraits directement en utilisant des points de reprise incrémentaux (ligne 18), au lieu de comparer des points de reprise complets ; 3) les points de reprise sont insérés directement dans les processus pour mettre à jour les espaces mémoire (lignes 9, 15 et 23), sans avoir besoin de les redémarer.

Par rapport au prototype CAPE-1, celui-ci présente les avantages suivants :

- *réduction de la quantité des données transférées sur le réseau.* La taille des points de prise `before` est beaucoup plus faible. Dans le cas de CAPE-2, ces points de prise ne contiennent généralement qu'un très petit nombre d'octets, tandis que les points de prise complèts dans le cas de CAPE-1 sont beaucoup plus grands. Dans le cas où un `broadcast` est effectué à la fin de la construction, une liste des modifications est transmise sur le réseau. Cependant, cette liste a généralement une taille plus petite que celle d'un point de reprise complet. De plus, une bonne implémentation du broadcast peut réduire significativement le temps d'exécution.

```
# pragma omp parallel for
    for ( A ; B ; C )
        D
```

↓ automatically translated into ↓

```
1   if ( master ( ) )
2       start ( )
3       for ( A ; B ; C )
4           create ( before )
5           send ( before, slave_x )
6       create ( final )
7       stop ( )
8       wait_for ( after )
9       inject ( after )
10      if ( ! last_parallel ( ) )
11          merge ( final, after )
12          broadcast ( final )
13  else
14      receive ( before )
15      inject ( before )
16      start ( )
17      D
18      create ( after_i )
19      stop ( )
20      send ( after_i, master )
21      if ( ! last_parallel ( ) )
22          receive ( final )
23          inject ( final )
24      else
25          exit
```

Figure 3. Modèle pour les boucles `parallel for` utilisant des points de reprise incrémentaux.

- *réduction du nombre de comparaisons pour extraire des résultats sur les nœuds esclaves :* si la taille des régions mémoire modifiées sur les nœuds esclaves est plus petite que celle des points de reprise complets, ce nombre est réduit d'autant ;
- *réduction de la quantité d'espace mémoire nécessaire pour stocker les données temporaires*. Les points de reprise `original, target, before` dans le prototype CAPE-1 ne sont plus utilisés dans CAPE-2. Un nouveau tampon pour les points de prise incrémentaux `after` est ajouté, mais sa taille est plus petite que celle des points de reprise complets ;
- *faible latence au démarrage / à la reprise de l'exécution des processus :* grâce à la primitive `inject` du *checkpointer* DICKPT, les points de prise `before, after,` et `final` peuvent être intégrés directement dans l'espace d'adressage des processus. Ainsi, les processus ne doivent plus être démarrés ou reprendre leur exécution à partir d'un point de reprise et, par conséquent, le temps d'exécution global peut être réduit.

4.3. Évaluation des performances

Afin de valider notre approche, nous avons réalisé des mesures de performance sur un *desktop cluster* composé de nœuds à base de processeurs Intel® Core™ 2 Duo E8400 cadencés à 3 GHz, intégrant 2 Go de RAM, exploités par le noyau Linux 2.6.35 avec Ubuntu version 10.10 et interconnectés par un réseau Ethernet standard à 100 Mb/s. Afin d'éviter autant que possible les influences extérieures, le système était consacré aux essais pendant toute la durée des mesures.

Le programme utilisé pour les tests est un produit de matrices dont la taille varie de 3000×3000 à 12000×12000. Les matrices sont denses et aucun algorithme spécifique n'a été mis en œuvre pour prendre en compte les matrices creuses. Chaque expérience a été réalisée au moins 10 fois et un intervalle de confiance d'au moins 90 % a toujours été obtenu pour les mesures. Les données rapportées ici sont les moyennes des 10 mesures.

Les Fig. 4 et 5 présentent le temps d'exécution en seconde du produit de matrices pour différents nombres de nœuds et tailles de matrices. En dépit du fait que les processeurs soient double-cœurs, un seul cœur a été utilisé lors des mesures. Trois mesures sont représentées à chaque fois : à gauche, celle de CAPE-1 ; au milieu, celle CAPE-2 ; et à droite, celle de MPI. Le programme MPI utilisé a été développé par nos soins et est optimisé afin de limiter au maximum la quatité de données échangées.

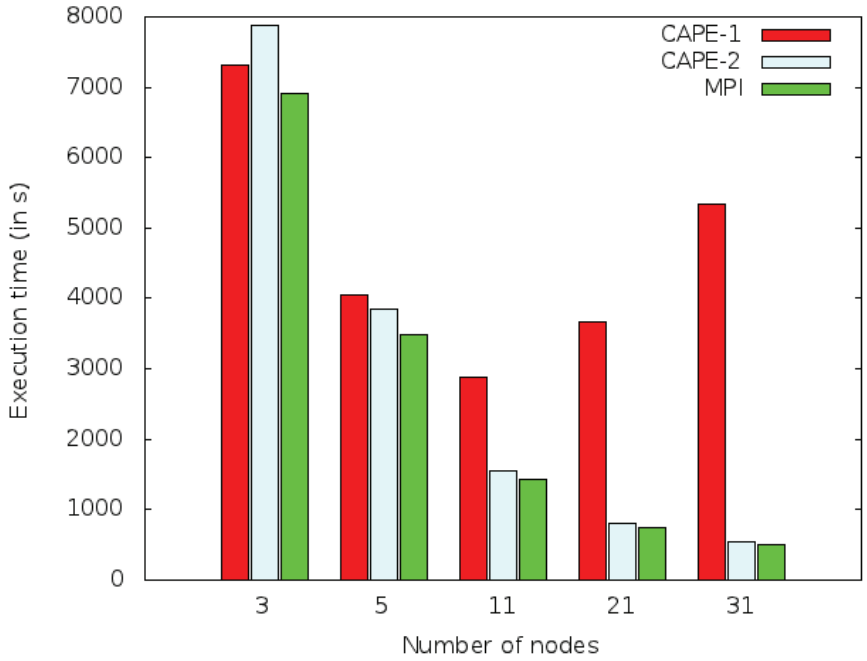


Figure 4. Temps d'exécution (en secondes) en fonction du nombre de nœuds.

La Fig. 4 présente le temps d'exécution pour différents nombres de nœuds. La taille des matrices est constante et égale à 12000×12000. Cependant, des remarques similaires pourraient faits pour les autres tailles de matrices. On peut remarquer que, sauf dans le cas de trois nœuds, le temps d'exécution en utilisant des points de reprise incrémentaux est toujours inférieur au temps d'exécution en utilisant des points de reprise complets. Plus le nombre de nœuds est grand, plus faible est le temps d'exécution pour CAPE-2 et MPI. De plus, le temps d'exécution pour CAPE-2 se rapproche de plus en plus de celui de MPI à mesure que le nombre de nœuds augmente. Le cas de CAPE-1 est différent. Lorsque peu de nœuds sont

utilisés pour le calcul (jusqu'à 11), le temps d'exécution baisse quand le nombre de nœuds augmente, et la valeur est assez similaire aux deux autres cas (CAPE-2 et MPI). Toutefois, pour de plus grands nombres de nœuds, le temps d'exécution pour CAPE-1 devient directement proportionnel au nombre de nœuds. Ceci est dû à la quantité de données qui sont transmises sur le réseau qui devient très important (il y a au moins un point de reprise complet pour chaque nœud esclave), même si la quantité de données qui sont effectivement intéressantes pour chaque nœud esclave est réduite. Ceci a clairement justifié l'utilisation de points de reprise incrémentaux pour CAPE.

Ensuite, les performances pour trois nœuds peuvent paraître étrange avec un temps d'exécution pour CAPE-1 plus faible que pour CAPE-2. En fait, avec un petit nombre de nœuds, la quantité de données transmises des nœuds esclaves vers le nœud maître est presque la même que l'on utilise des points de reprise complets ou incrémentaux, puisque dans les deux cas une grande partie des matrices est mise à jour par chacun des esclaves. De plus, le nœud maître ne doit envoyer qu'un petit nombre de points de reprise, ce qui fait que le temps pour cette phase devient négligeable dans le temps d'exécution total. Toutefois, dans le cas des points de reprise incrémentaux, les processus sont surveillés afin de capturer les pages mémoires accédée en écriture. Le surcoût induit par cette surveillance est proportionnelle à la quantité de calcul et inversement proportionnelle au nombre de nœuds ; elle est donc très élevée dans le cadre de nos mesures lorsque peu de nœuds sont impliqués. Heureusement, ceci n'est pas un problème pour CAPE. Des processeurs à quatre, voire huit cœurs sont déjà disponibles sur le marché et, par conséquent, CAPE vise des architectures ayant un nombre de nœuds/cœurs beaucoup plus élevé.

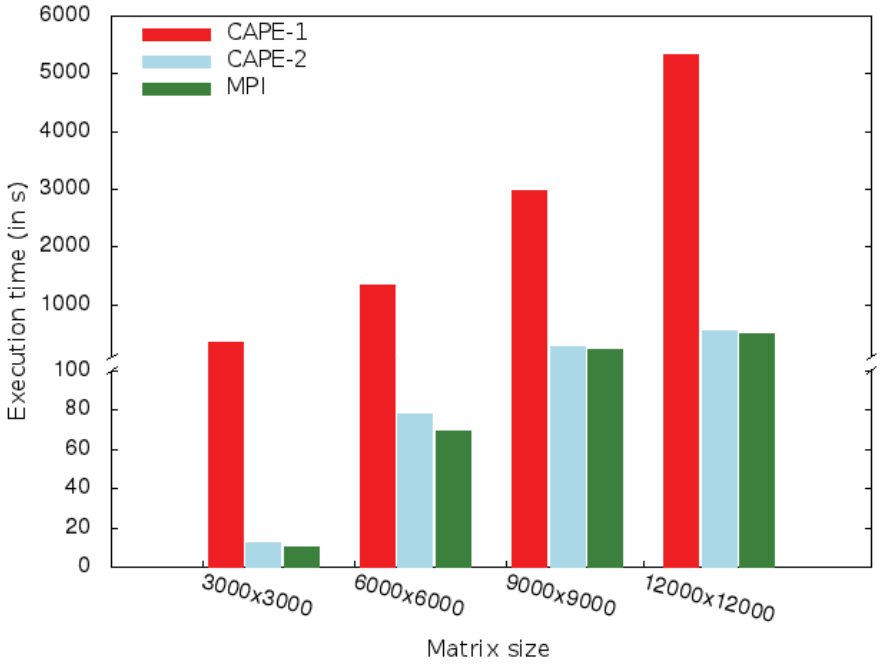


Figure 5. Temps d'exécution (en secondes) par rapport à la taille du problème.

La Fig. 5 présente le temps d'exécution pour des tailles de matrices différentes. Le nombre de nœuds utilisé ici est 31. Cependant, des remarques similaires pourraient être faites pour les autres nombres de nœuds étudiés. La figure montre clairement

que le temps d'exécution de CAPE-1 est directement proportionnel au carré de la taille des matrices, tandis que le temps d'exécution pour CAPE-2 et MPI est directement proportionnel à la taille de la matrice. Ceci est dû au fait que la mémoire des processus se compose principalement des matrices, et que presque toute la mémoire virtuelle des processus est transmise sur le réseau lors de l'utilisation de points de reprise complets. Avec CAPE-2 et MPI, les mémoires virtuelles ne sont pas transmises sur le réseau et seules les données qui ont été mises à jour lors du calcul du produit matrice-matrice sont prises en compte. De plus, on peut remarquer que les temps d'exécution pour CAPE-2 et MPI sont très proches. Une analyse en profondeur des performances montre que le temps d'exécution de CAPE-2 n'est que 10 % plus important que le temps d'exécution de MPI, sauf pour les matrices 3000×3000 où le ratio est de 1,3.

La Fig. 6 montre l'accélération de CAPE-2 pour différents nombres de nœuds et tailles de matrice. La ligne en pointillée représente l'accélération théorique maximale. Cette figure montre clairement que la solution est efficace avec un rendement (rapport de l'accélération sur le nombre de nœuds) de l'ordre de 75 % à 90 %. Elle met aussi en évidence que plus la taille des matrices est grande, plus le gain en vitesse est important, ce qui n'était pas le cas avec CAPE-1.

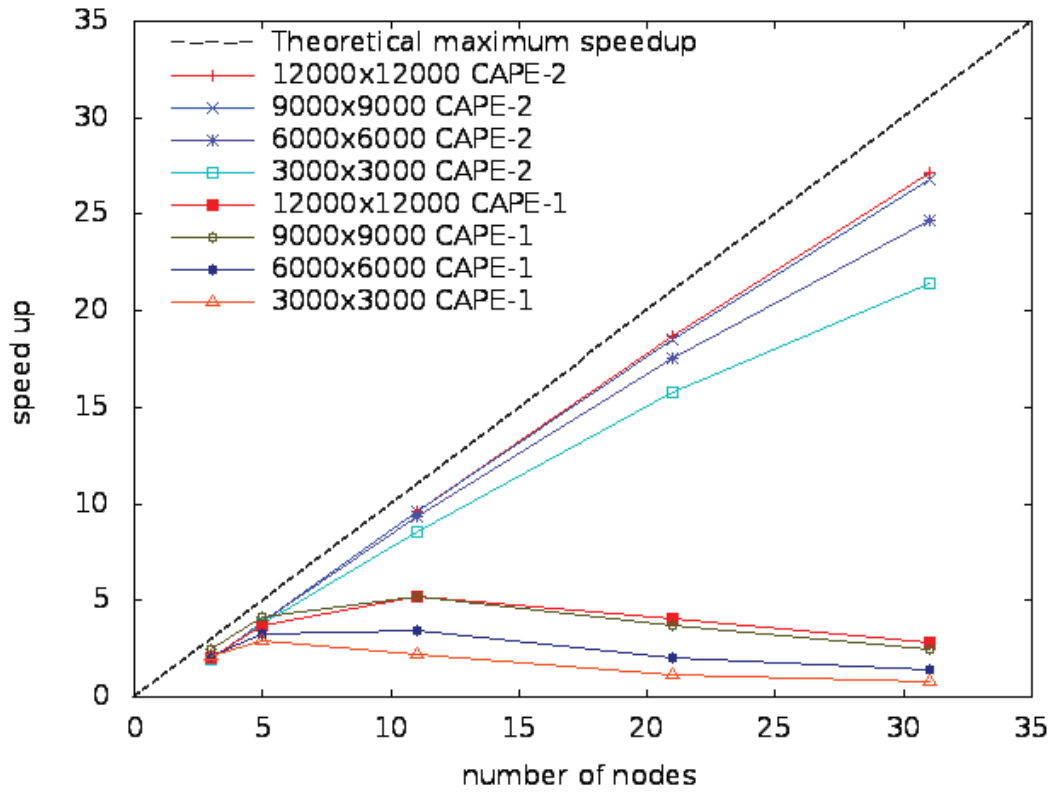


Figure 6 : Accélération en fonction du nombre de nœuds.

4.4. Conclusion

Cette section a présenté CAPE-2, la nouvelle version de CAPE basée sur des points de reprise incrémentaux. Avec les nouvelles propriétés fournies par DICKPT, cette version a permis de surmonter les quatre premiers points faibles de CAPE-1 (Cf Sec. 2.3) : 1) la réduction de la quantité de données transmises sur le réseau ; 2) la réduction du nombre de comparaisons pour extraire les résultats d'exécution sur les nœuds esclaves ; 3) la réduction de la quantité d'espace mémoire pour stocker des données temporaires ; et 4) l'élimination des phases démarrer/reprendre des

processus pour intégrer les points de reprise dans leurs espaces respectifs. Toutes ces améliorations ont permis d'accroître significativement les performances globales de CAPE.

5. Prise en compte des données partagées

5.1. Le modèle *Updated Home-based Lazy Release Consistency* (UHLRC)

OpenMP est basé sur le modèle de mémoire partagée *shared-relaxed consistency*. Dans ce modèle, tous les *threads* partagent une mémoire commune. Toutefois, chaque *thread* peut également exécuter sur sa propre mémoire locale, qui est une vue temporaire de la mémoire commune. La cohérence entre la mémoire locale et la mémoire commune est réalisée par l'utilisation de la directive `flush`. Un appel à la directive `flush` permet de rendre cohérente la vue temporaire d'un *thread* et la mémoire commune. Par contre, elle n'affecte pas les autres *threads*.

Il existe deux types de `flush` dans OpenMP : le premier (*flush-set* ou *selective flush*) nécessite de préciser les variables concernées et le second (*global flush*) ne requiert aucun paramètre. Pour la *selective flush*, la cohérence est appliquée uniquement sur les variables précisées, tandis que pour le *global flush*, la cohérence est appliquée sur l'espace mémoire tout entier.

Le modèle d'exécution de base de CAPE sur des systèmes homogènes assure la cohérence entre les mémoires des processus maître et esclaves dans les régions séquentielles et les points de début et de fin de régions parallèles. À l'initialisation du processus et dans les régions séquentielles, tous les *threads* exécutent les mêmes instructions, ils ont donc les mêmes espaces mémoires. Il en est de même pour le début des régions parallèles (avant la division du travail par les constructions parallèles). À la fin des constructions parallèles, tous les *threads* injectent le même ensemble d'éléments modifiés, ce qui rend leurs espaces mémoires cohérents. Ainsi, le seul problème restant à assurer est la cohérence entre les *threads* au sein des sections parallèles est l'implémentation d'un mécanisme pour la directive `flush`.

Pour la plupart des approches utilisant le modèle *Home-based Lazy Release Consistency* (HLRC), un flush sur une copie d'une page sur un nœud distant comporte trois phases principales :

1. sur le nœud distant : calcul des différences entre les pages (aussi nommé liste des *diffs*) en utilisant la fonction diff, puis envoyer ces différences au nœud *home* ;
2. sur le nœud *home :* application des différences reçues à la page *home* ; puis, calcul des différences entre les pages et envoi des différences au nœud distant ;
3. sur le nœud distant : application des différences reçues à la page.

Pour le *global flush*, les phases ci-dessus sont appliquées pour chaque page partagée du *thread*.

Dans le cas de CAPE, l'algorithme ci-dessus peut être utilisé directement en considérant que le nœud maître est le nœud *home* et que ses pages mémoires sont les pages *homes*. Cependant, à la suite de l'utilisation de points de reprise, le coût de l'exécution du *flush* peut être réduit en deux manières. Tout d'abord, sur les nœuds esclaves, la fonction `create` du *checkpointer* incrémental peut remplacer la fonction `diff` puisqu'elles font exactement le même travail. Deuxièmement, sur le nœud maître, le nombre de comparaisons est considérablement réduit si une liste

d'éléments mémoires modifiées de toutes les pages partagées est maintenue et la fonction `diff` est appliquée sur cette liste au lieu des pages *homes*. D'un autre point de vue, cette liste contient les résultats de l'exécution des nœuds esclaves qui ont appelé la fonction *global flush*. Donc, ces résultats peuvent être fusionnés dans le dernier résultat à la fin de la région parallèle. En remplacement de la liste modifiée par l'ensemble des pages *homes*, ce modèle est appelé modèle *Updated Home-Based Lazy Release Consistency* (UHLRC).

5.2. L'implémentation des directives et clauses associées aux données partagées d'OpenMP

Le dernier problème associé aux données partagées concerne l'implémentation des directives et des clauses OpenMP. Pour les directives et clauses `shared`, `threadprivate`, `private`, `firstprivate`, `lastprivate`, `copyin`, `copyprivate` et reduction, la solution consiste à déclarer et utiliser des variables locales auxiliaires dans chaque processus, et en insérant les instructions nécessaires pour initialiser ces variables et/ou mettre à jour la valeur des variables d'origine à la fin des constructions. Le cas de `reduction` est un peu plus complexe et nécessite un traitement supplémentaire. Une solution possible consiste à utiliser une combinaison des valeurs partielles dans les points de reprise sur les nœuds esclaves et de les re-collecter sur le nœud maître pour enfin calculer la valeur de la réduction. Une autre solution consiste à utliser des fonctions spéciales sur les nœuds esclaves et maître pour envoyer ou recevoire des valeurs partielles de la variable reduite.

La transformation d'un programme OpenMP contenant des directives et des clauses associées aux données partagées est divisée en trois étapes lors de l'utilisation CAPE : (1) les directives et clauses contenant les mêmes variables sont fusionnées ; (2) les directives et clauses sont remplacés par un ensemble d'instructions équivalentes ; (3) les modèles de CAPE pour des constructions OpenMP sont appliqués pour transformer le programme vers le language de base.

5.2.1. Fusion des directives et clauses

Il peut arriver qu'une variable soit référencée dans plusieurs directives et/ou clauses. Dans ces cas, il est possible de les ramener à une forme unique. Par exemple, dans le code suivant :

```
# pragma omp threadprivate ( x )
# pragma omp parallel for copyin ( x )
for ( A ; B ; C ) D ;
```

la directive de la première ligne peut être supprimée et fusionnée avec la clause `copyin (x)` de la deuxième ligne. Cette règle peut également être appliquée pour fusionner cette directive et la clause `copyprivate`. Ceci est généralisable à bien d'autres cas.

5.2.2. Modèle général

La Fig. 7 présente le modèle général permettant de transformer des boucles `parallel for`.

À l'initialisation, tous les blocs sont transformés, y compris le `before_block`, `enter_block`, `exit_block` et `after_block` qui sont vides. Puis, le modèle est appliqué de manière itérative pour chaque directive ou clause en fusionnant les parties transformées de la directive ou clause correspondant à chaque étape.

```
# pragma omp parallel for clause_U
    for ( A ; B ; C )
         D
```

↓ automatically translated into ↓

```
before_block
{
     enter_block
     # pragma omp parallel for
     for ( A ; B ; C )
     {
          D
          exit_block
     }
}
after_block
```

Figure 7 : Modèle pour les primitives et clauses associées aux données partagées dans les boucles `parallel for`.

5.2.3. Blocs de la transformation

Le tableau 2 montre les blocs de transformation pour les directives et les clauses associées aux données partagées d'OpenMP. Le cas des clauses `copyprivate` et `reduction` ne sont pas complètement fournis dans ce tableau (mais intégrés dans la thèse). À ce jour, l'hypothèse qui est faite est que toutes les variables sont partagées par défaut et CAPE ne considère pas l'option `none` pour le cas de la clause `defaut`. Ainsi, cette clause n'est pas prise en compte dans le tableau.

5.3. Evaluation des performances

Actuellement, l'intégration dans CAPE du modèle UHLRC et le traitement des directives et clauses associées aux données partagées est en cours de développement. Cependant, à partir de la description ci-dessus, il apparaît que cette intégration ne devrait pas diminuer significativement les performances globales en considérant que :

- d'une maniére générale, le modèle UHLRC ne modifie pas significativement le modèle mémoire actuel de CAPE. Le seul ajout fait est le mécanisme utilisé pour implémenter la directive `flush`, et une seule tâche auxiliaire est ajoutée aux moniteurs. Il n'y a donc pas beaucoup de changements dans son exécution. Sur les nœuds esclaves, le mécanisme implémentant cette tâche est le même que celui de la tâche principale du *checkpointer* incrémental. En fait, en fonctionnant comme un *checkpointer* incrémental, le moniteur doit toujours écouter les signaux du processus monitoré. Ainsi, le travail consiste simplement à inclure un nouveau cas pour le signal ajouté pour la demande de traitement de la directive `flush`. En outre, le nœud maître, lors de l'exécution des régions parallèles, vise à répartir des travaux aux nœuds

	before_block	enter_block	exit_block	after_block
shared (x)				
threadprivate (x)	typeof (x) __x__ ;	typeof (__x__) x ;		
private (x)	typeof (x) __x__ ;	typeof (__x__) x ;		
firstprivate (x)	typeof (x) __x__ ; __x__ = x;	typeof (__x__) x ; x = __x__		
lastprivate (x)	typeof (x) __x__ ;	typeof (__x__) x ;	if (thread_num == (num_threads - 1)) __x__ = x ;	x = __x__ ;
reduction (x)	typeof (x) __x__ ;	typeof (__x__) x; init_value (x) ;	send (x, master) ;	*continue in subsec. 5.5.3.5*

copyin (x)	typeof (x) __x__ ;
	if (master ( ))
	__x__ = x ;
		typeof (__x__) x ;
		if (master ( )) {
		x = __x__ ;
		broadcast (x) ; }
		else {
		receive (x) ;
		inject (x) ; }		

Tableau 2 : Blocs de la transformation.

esclaves, puis à attendre les résultats. Cela signifie que généralement, dans les régions parallèles, le nœud maître est inactif. En conséquence, cette nouvelle tâche demande simplement que le nœud écoute et traite les demandes de flush venant des nœuds esclaves ;

- avec le modèle UHLRC, le nombre de comparaisons — le principal opérateur pour implémenter la directive `flush` — est diminué. Il devrait donc théoriquement offrir de meilleures performances ;
- le traitement des directives et clauses associées aux données partagées est, dans la plupart des cas, effectué au moment de la compilation. L'exécution des codes supplémentaires ne consomme pas beaucoup de temps, tant que ceux-ci sont limités à la création de variables locales et à l'assignation de leur valeur initiale. Le cas particulier des directives `reduction` et `copyprivate` exige la transfert d'une petite quantité de données entre les nœuds, ce qui n'affecte pas les performances d'une manière générale.

5.4. Conclusion

Cette section a présenté des solutions pour deux aspects liés aux données partagées d'OpenMP : l'implémentation d'un modèle de mémoire partagée et l'implémentation des directives et clauses associées aux données partagées.

Concernant le modèle de mémoire partagé, UHLRC est un nouveau modèle de mémoire partagée répartie basé sur le modèle HLRC, est présenté dans ce document. Les algorithmes permettant la mise en œuvre des opérations `flush` dans ce modèle sont également présentés. Grâce aux améliorations qui réduisent le nombre d'opérations de comparaisons — la plus importante opération pour garantir la cohérence de la mémoire des processus — ce nouveau modèle promet d'atteindre de bonnes performances.

Pour l'implémentation des directives et clauses associées aux données partagées, un modèle global et des blocs élémentaires pour chaque directive et clause a été présenté. Cette implémentation permet de transformer toutes les directives et clauses associées aux données partagées d'OpenMP sans affecter significativement les performances globales.

6. Conclusion et perspectives

Ce document a présenté les travaux de la thèse sur les trois aspects principaux visant à améliorer les performances et à surmonter les restrictions de CAPE sur les problèmes liés aux données partagées.

1. DICKPT — une nouvelle technique de point de reprise. DICKPT est une technique basée des points de reprise incrémentaux. En introduisant la possibilité de prendre des points de reprise incrémentaux indépendants sur des sections discontinues du programme, les performances des points de reprise tant sur le temps d'exécution que sur la taille des points de reprise, est significativement réduite dans de nombreux cas. Ceci est très utile pour CAPE afin de prendre des points de reprise sur les parties nécessaires des programmes OpenMP transformés, ce qui contribue à accroître les performances globales. En Exploitant la structure particulière de points de reprise incrémentaux, certaines méthodes permettant de les compresser sont aussi présentées dans le document ;
2. CAPE-2 — une nouvelle version de CAPE — utilise les points de reprise incrémentaux. En exploitant les nouvelles propriétés de DICKPT, un nouveau modèle d'exécution et de nouveaux prototypes de transformation des constructions dédiées au partage du travail

(*work-sharing*) d'OpenMP ont été proposé. Dans cette version, les inconvénients de CAPE-1 qui affectaient les performances globales ont été surmontés : la quantité de données transmises sur le réseau, le nombre d'opérations pour extraire les résultats d'exécution, la quantité d'espace mémoire pour les données temporaires, ont été considérablement réduits. De plus, grâce à la possibilité d'injecter directement des points de reprise dans l'espace d'adressage des processus, la nécessité de redémarrer après l'intégration des points de reprise dans CAPE-1 a été supprimé. Tous ces éléments conduisent à une forte amélioration des performances de CAPE-2 par rapport à CAPE-1 ;

3. Les solutions pour les problèmes liés aux données partagées d'OpenMP. Deux aspects ont été présentés et résolus : 1) l'implémentation d'un nouveau modèle de mémoire partagée, et 2) l'implémentation des directives et clauses associées aux données partagées.

   Le premier aspect présenté est UHLRC, un modèle de mémoire partagée, basé sur le modèle HLRC. Ce modèle possède deux inovations importantes. La première est l'utilisation de la directive `create` du *checkpointer* DICKPT au lieu de l'utilisation de la fonction `diff` pour calculer les différences entre le contenu actuel de la mémoire du processus et celle de la mémoire tampon. La seconde est liée au remplacement des pages *home* dans le *thread* maître par une liste d'éléments mémoire modifiés. D'un côté, cela permet de réduire le nombre de comparaisons dans la fonction `flush`, fonction qui est utilisée pour garantir la cohérence entre la mémoire des processus maître et esclaves. D'un autre côté, grâce à la connaissance de la liste des régions modifiées dans l'espace mémoire des moniteurs de CAPE, la quantité de données transmises entre les processus est réduite, et cela contribue à son tour à accroître les performances globales. Les algorithmes permettant d'implémenter le modèle UHLRC dans CAPE a aussi été présenté.

   Pour le deuxième aspect, les algorithmes pour implémenter les directives et clauses liées aux données partagées d'OpenMP sont fournis.

   Bien qu'actuellement, nous n'ayons pas effectué les expérimentations nécessaires à l'évaluation des performances dans ce cas, nous pouvons estimer que l'intégration de ce nouveau modèle mémoire dans CAPE n'affecte pas significativement les performances globales. Cela est dû à deux principaux arguments. Le premier est est le fait que le nouveau modèle mémoire est approprié au modèle d'exécution de CAPE qui ne nécessite alors que quelques légers changements dans les prototypes de CAPE. Le deuxième est lié à l'implémentation des directives et des clauses d'OpenMP à l'aide de seulement quelques instructions simples ne consommant pas beaucoup de temps d'exécution.

Globalement, les performances de CAPE ont été augmentées de manière significative dans cette nouvelle version. Tous les problèmes liés aux données partagées ont été analysés et résolus par l'utilisation du nouveau modèle mémoire et les algorithmes présentés.

Dans un proche avenir, nous souhaitons développer de nouveaux composants de CAPE dans lesquels le modèle UHLRC et les directives et les clauses d'OpenMP associées aux données partagées sont complètement implémentées. Un autre travail serait de mener plus d'expériences sur l'exécution de CAPE, en particulier en le comparant aux autres implémentations d'OpenMP sur machines à mémoire distribuée. Cela permettra de fournir une meilleure évaluation et ainsi de mieux démontrer les avantages de cette implémentation de CAPE totalement conforme à OpenMP et à hautes performances sur des systèmes à mémoire distribuée.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Introduction

Despite the quick development of hardware solutions, there are always requirements that go far beyond the capabilities of sequential programs. Parallel programing has been the unique solution to bypass the hardware limitations. In this approach, problems are divided in smaller parts that can be executed concurrently. For example, in Fig. 1.1, the problem is divided in four parts and at each moment t_i, different instructions of these parts are executed concurrently on different CPUs. Almost all modern programming languages enable parallel execution and the most common way consists in using processes and threads. Processes have separated memory spaces and execute independently. This leads to solid systems: the break-down of a process does not lead the system to shutdown. However, inter-process data sharing and synchronization are complex and time consuming. Threads use a common virtual address space that makes data sharing much easier and lower time consuming. Nevertheless, the break-down of a thread affects the whole program. Furthermore, multi-threaded programs typically run on SMP systems, not on distributed-memory architectures. As an example, Fig.1.2 shows a matrix-matrix product program written in C based on POSIX Threads, a widely used API for threads. Programming parallel applications in this way requires many auxiliary works at the base level such as creating parallel streams, assigning jobs to streams, making them running, ensuring the synchronization, etc. that makes it becoming a heavy, boring and difficult task. Parallel programing models at a higher level are required to help programmers bypass these difficulties.

OpenMP [1] has become the standard for the development of parallel applications on shared-memory platforms. It is composed of a set of very simple and powerful directives and functions to generate parallel programs in C, C++ or Fortran. From the programmer's point of view, OpenMP is easy to use as it allows to incrementally express parallelism in sequential programs, i.e. the programmer can start with a sequential version of a program (written in C, C++ or Fortran) and step by step

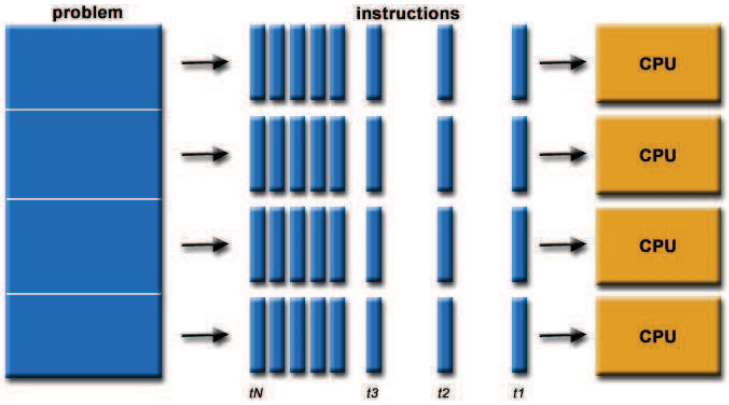


Figure 1.1: Parallel computing [2].

add OpenMP directives to change it into a parallel version. Moreover, the level of abstraction provided by OpenMP makes the expression of parallelism more implicit where the programmer specifies what is desired rather than how to do it. This has to be compared to message-passing libraries, like MPI [3], where the programmer specifies how things must be done using explicit send/receive and synchronization calls.

Figure 1.3 presents an OpenMP code computing the product of two-dimension matrices A and B, with the result stored in matrix C.

The only difference between this code and the original sequential code is the insertion of a directive to include the OpenMP header file and the directive in the form of a `pragma` indicating that the following `for` loop has to be executed in parallel. The work is more complex for programmers using MPI (an example of code is presented in Fig. 1.4). In this case, programmers are responsible for organizing the work: a master sends initial data to workers and each worker computes a part of the product and sends back its result to the master. Programmers also have to write many other auxiliary operations such at MPI initialization, specify data types, transaction types, etc. All this makes the MPI code completely different from the sequential code, and far more difficult to write, to debug and to understand.

Nevertheless, OpenMP has a major restriction for large-scale computers: it is designed for SMP architectures. If SMP machines have been available for a long time as multi-CPU systems and have become more popular with the development of multi-core architectures, this restriction remains important as it prevents the use of OpenMP on distributed-memory architectures such as clusters, grids and clouds, and therefore does not allow OpenMP to take benefits of the development of new architectures like low-cost desktop clusters, grids and more generally high-performance systems [4].

Many attempts have tried to port OpenMP on distributed-memory systems. However they all fail to give a fully-compliant high-performance OpenMP implementation. A straightforward idea to implement OpenMP on distributed systems

```
#include <pthread.h>
...
#define N 100
int num_thrd, A[N][N], B[N][N], C[N][N];
void* multiply(void* slice){
    int s = (int)slice;
    int i, j, k;
    for (i = (s * N)/num_thrd; i < ((s+1) * N)/num_thrd; i++){
         for (j = 0; j < N; j++){
         C[i][j] = 0;
              for ( k = 0; k < N; k++)
                   C[i][j] += A[i][k]*B[k][j];
         }
    }
}
int main(int argc, char* argv[]){
    pthread_t* thread;
    int i;
    num_thrd = atoi(argv[1]);
    thread = (pthread_t*) malloc(num_thrd*sizeof(pthread_t));
    for (i = 1; i < num_thrd; i++){
         if (pthread_create (&thread[i], NULL, multiply, (void*)i) != 0 ){
              perror("Cannot create thread");
              free(thread);
              exit(-1);
         }
    }
/* main thread works on slice 0 so everybody is busy */
    multiply(0);
    /* main thead waiting for te other threads to complete */
    for (i = 1; i < num_thrd; i++)
         pthread_join (thread[i], NULL);
    ...
}
```

Figure 1.2: POSIX Thread code of a matrix-matrix product computation.

```
#include <omp.h>
...
#pragma omp parallel for private(row, col, ind)
for(row = 0; row < N; row++){
    for(col = 0; col < N; col++)
          for(ind = 0; ind < N; ind++)
               C[row][col] += A[row][ind] + B[ind][col];
}
```

Figure 1.3: OpenMP code for a matrix-matrix product computation.

is to use the virtual global address space provided by a Single System Image (SSI) and/or a Distributed Shared Memory (DSM) [5] to emulate the memory shared by SMP systems. OpenMP programs can then be compiled and run without major modifications. This approach can easily provide a fully-compliant version of OpenMP. However, the global address space is located across machines and causes a strong overhead to the global performance. This is clearly showed in [5], where the larger the number of threads, the lower the performance. As a result, in order to reduce the impact of synchronization of shared-memory regions, relaxed-consistency memory models have been used [6][7]. However, this approach meets the difficulty to specify shared variables outside parallel regions as DSMs require an explicitly declaration of shared variables while OpenMP uses an implicit shared-memory model. The implementation of OpenMP on top of MPI [8][9] was promising high performance. However, it systematically requires the programmer to add non-OpenMP additional directives, e.g. to specify returned data, meaning that these kinds of implementations cannot be considered as fully OpenMP compliant. The implementation of OpenMP on top of Global Array [10] also implies many difficulties to specify how data should be distributed. It is also the problem of the Cluster OpenMP of Intel [11], that leads to the use of an additional directive to specify shared variables.

In the OpenMP fork-join execution model, an OpenMP program begins as a single thread called the master thread that is executed sequentially. When the thread encounters a parallel construct, it creates a set of slave threads to execute concurently the job of the construct. Only the master thread resumes its execution beyond the end of the parallel construct. Figure 1.5 presents the principle of this model.

The base principle of the CAPE execution model is similar to the OpenMP fork-join model, with the replacement of threads by processes. This helps distributing them on the different nodes of distributed-memory architectures. Checkpointing technique is used to dump the master process memory space, distribute it to the other nodes of the system where the process is restarted (slave processes). It is also the base tool to extract results from slave processes, merge them into the master process's space and resume the execution of the master.

```
#include<mpi.h>
...
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
numworkers = numtasks-1;
if (taskid == MASTER) { /************* master task ******************/
    ...
    /* send matrix data to the worker tasks */
    averow = N/numworkers;
    offset = 0;
    for (dest=1; dest<=numworkers; dest++) {
         rows = averow;
         MPI_Send(&offset, 1, MPI_INT, dest, FROM_MASTER, MPI_COMM_WORLD);
         MPI_Send(&rows, 1, MPI_INT, dest, FROM_MASTER, MPI_COMM_WORLD);
         MPI_Send(&a[offset][0], rows*N, MPI_LONG, dest, FROM_MASTER, MPI_COMM_WORLD);
         MPI_Send(&b, N*N, MPI_LONG, dest, FROM_MASTER, MPI_COMM_WORLD);
         offset = offset + rows;
    }
    /* wait for results from all worker tasks */
    for (i=1; i<=numworkers; i++) {
         MPI_Recv(&offset, 1, MPI_INT, i, FROM_WORKER, MPI_COMM_WORLD, &status);
         MPI_Recv(&rows, 1, MPI_INT, i, FROM_WORKER, MPI_COMM_WORLD, &status);
         MPI_Recv(&c[offset][0], rows*N, MPI_LONG, i, FROM_WORKER, MPI_COMM_WORLD,
                  &status);
    }
if (taskid > MASTER) { /********** worker task ******************/
    ...
    MPI_Recv(&offset, 1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, MASTER, FROM_MASTER, MPI_COMM_WORLD, &status);
    MPI_Recv(&a, rows*N, MPI_LONG, MASTER, FROM_MASTER, MPI_COMM_WORLD, &status);
    MPI_Recv(&b, N*, MPI_LONG, MASTER, FROM_MASTER, MPI_COMM_WORLD, &status);
    for (k=0; k<N; k++)
         for (i=0; i<rows; i++)
              for (j=0; j<N; j++)
                   c[i][k] = c[i][k] + a[i][j] * b[j][k];
    MPI_Send(&offset, 1, MPI_INT, MASTER, FROM_WORKER, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, MASTER, FROM_WORKER, MPI_COMM_WORLD);
    MPI_Send(&c, rows*N, MPI_LONG, MASTER, FROM_WORKER, MPI_COMM_WORLD);
}
```

Figure 1.4: Example of MPI code for the matrix-matrix product computation.

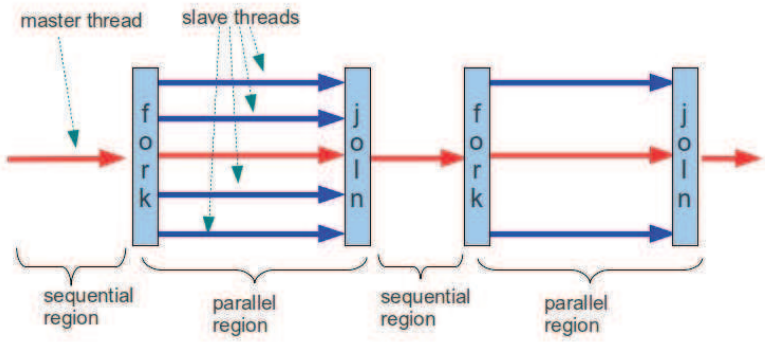


Figure 1.5: OpenMP fork-join model.

From this based idea, CAPE has a major advantage: the capability to automatically execute OpenMP programs in distributed-memory architectures. Indeed, from an initial OpenMP program, a special compiler can change it into a CAPE program. While running, a checkpointer is in charge of taking snapshots of process' memory, extract the result of program execution, include checkpoints into a process' memory and resume the execution of a process from a checkpoint. This feasibility has been proved with the first implementation [12][13] that was using complete checkpoints. However, there were two major restrictions in this version: the low performance and the inability to process shared data. The first restriction is mainly due to the use of complete checkpoints, that leads to a large amount of data transfered over the network. It also leads to the execution a large number of comparison operators to extract the result from slave processes. The second restriction is due to the fact that Bernstein's conditions must be verified in CAPE prototypes, which prevents CAPE becoming an OpenMP fully-compliant implementation.

1.2 Problem presentation

After the analysis of the restrictions of CAPE from the above discussion, this thesis focuses on the improvement of CAPE with two main objectives: increase its performance and add the ability of processing shared data.

For the first objective, we followed the CAPE execution principle and the analysis in [14]: the most important factor effects on CAPE's performance are the amount of transfered data over the network and the speed to extract execution's result from slave processes. Furthermore, the speed to include checkpoints into the memory space and to resume from an execution process at a checkpoint also have influences on the global performance. Decreasing the impact of these factors on the performance of CAPE is the first problem to solve.

The second objective relates first to the implementation of a shared-memory model for all processes of the system. As presented in [1], OpenMP uses the relaxed-

consistency (RC) [15] shared memory model. All OpenMP threads have access to a place to store and to retrieve variables, called the *memory*. In addition, each thread is allowed to have its own *temporary view* of the memory. The temporary view of the memory for each thread is not a required part of the OpenMP memory model, but can represent any kind of intervening structure. On SMP systems, this model is implicitly met as all threads use a common memory space. On distributed-memory systems, memory is physically spread over many nodes, so implementing a distributed-shared memory system (DSM) or a RC is required. Furthermore, for better performance, some derivations of the RC model have been developed, such as the Lazy Release Consistency (LRC) [16] model, the Home-Based Lazy Release Consistency (HLRC) [17] model. As a result, second problem to solve is: which is the most appropriate model for CAPE and how to implement it with high performance. The rest of this second objective's work relates to the processing of OpenMP data-sharing directives and clauses. This includes directives such as `threadprivate`, `default` and clauses such as `private`, `firstprivate`, `lastprivate`, `reduction`, `copyprivate`, `copyin`.

From the above discussion, this thesis proposes to solve the following requirements:

1. Increase the performance of CAPE by improving the works related to checkpoints and CAPE prototypes.
2. Implement a high-performance shared-memory model for CAPE.
3. Implement all OpenMP data-sharing directives and clauses.

1.3 Organization of the thesis

Chapter 2 is dedicated to the state of the art of CAPE. Section 2.1 presents an outline of OpenMP. Section 2.2 discusses about different implementations of OpenMP on distributed-memory systems: their principles, advantages and drawbacks. Section 2.3 focuses on checkpointing techniques, the base tool for CAPE. The last section of this chapter is an in-depth presentation of the first version of CAPE (CAPE-1), based on complete checkpoints.

Chapter 3 presents the Discontinuous Incremental Checkpointing (DICKPT) technique which is based on the Incremental Checkpointing technique. The presentation is composed of the design and a performance evaluation of the DICKPT.

Chapter 4 presents CAPE-2, a new version of CAPE using DICKPT technique. Section 4.1 and Sec. 4.2 show its execution model and its organization. Sections 4.3 and 4.4 aim at presenting the primitives and the prototypes to transform OpenMP constructs. The last section is used for an evaluation of performance.

Chapter 5 presents two of the main aspects to implement data-sharing for CAPE. The first one is the design of the Updated Home-based Lazy Relax Consistency, a

new memory model based on Home-based Lazy Relax Consistency with modifications to exploit the special characteristics of CAPE. The second one presents the prototypes implementing OpenMP data-sharing directives and clauses.

The last part of the thesis, in Chap. 6, concludes this document and presents an overview of the future work.

Chapter 2

State of the art

Contents

2.1 OpenMP

OpenMP is a parallel programming API that provides a standard among a variety of shared-memory architectures/platforms with four main goals [18]:

1. Establish a simple and limited set of directives to program shared-memory machines, where most expression of parallelism can be implemented using only three or four directives.

2. Provide the capability to incrementally parallelize a sequential program, unlike message-passing libraries which typically require an all or nothing approach.

3. Provide the capability to implement both coarse-grain (domain decomposition) and fine-grain (loop-level) parallelism.

4. Provide a portable product. This goal has been achieved by supporting Fortran (77, 90, and 95), C, and C++ and a public forum for API and membership.

Led by the OpenMP Architecture Review Board (ARB), with many strong members such as Compaq, Hewlett-Packard, Intel, IBM, Sun, etc. the OpenMP standard specification started in Spring 1997. The current version (2012) is 3.1. It is composed of a collection of compiler directives, library routines, environment variables and provides a model for parallel programming that is portable across shared-memory architectures from different vendors.

OpenMP is not an independent programming language [1]. OpenMP programs have to be first written in a *base language*[1], then changed into parallel form by inserting OpenMP directives. An *OpenMP compliant implementation*[2] is required to compile such a program. At present, there are many OpenMP compliant implementations provided by many vendors such as GNU, IBM, Oracle, Intel, HP, Cray on almost all common platforms such as Windows, Linux, Solaris, AIX, Mac OS, Cray XT.

OpenMP is an explicit programming model [1]. Programmers have to explicitly specify the actions to be taken by the compiler and runtime system in order to execute the program in parallel. OpenMP-compliant implementations are not required to check for data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs. In addition, compliant implementations are not required to check for code sequences that cause a program to be classified as non-conforming. Application developers are responsible for correctly using the OpenMP API to produce a conforming program. The OpenMP API does not cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization.

With the strong abilities and the ease to use, OpenMP is more and more applied on shared-memory parallel programming.

2.1.1 Execution model

OpenMP uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. OpenMP is intended to support programs that executes correctly both as parallel programs and as sequential programs (directives ignored and a simple OpenMP stubs library). However, it is possible and permitted to develop a program that executes correctly as a parallel program but not as a sequential program, or that produces different results when executed as a parallel program compared to when it is executed as a sequential program. Furthermore, using different numbers of threads may result in different numeric results because of changes in the association of numeric operations.

An OpenMP program begins as a single thread of execution, called the initial thread. The initial thread executes sequentially. When any thread encounters a

[1] A programming language that serves as the foundation of the OpenMP specification.

[2] An implementation of the OpenMP specifications that compiles and executes any conforming program as defined by the specification.

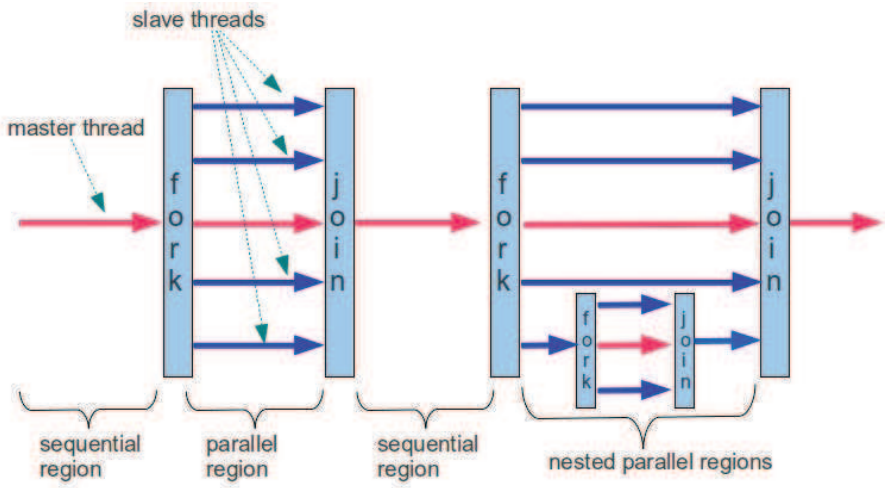


Figure 2.1: OpenMP fork-join model with nested parallel regions.

parallel construct, the thread creates a team composed of itself and zero or more additional threads and becomes the master of the new team. A set of implicit tasks, one per thread, is generated. The code for each task is defined by the code inside the parallel construct. Each task is assigned a different thread in the team and becomes tied, i.e. it is always executed by the thread to which it is initially assigned. There is an implicit barrier at the end of the parallel construct. Only the master thread resumes its execution beyond the end of the parallel construct, resuming the task region that was suspended upon encountering the parallel construct. Any number of parallel constructs can be specified in a single program.

Parallel regions may be arbitrarily nested inside each other. If nested parallelism is disabled, or is not supported by the OpenMP implementation, then the new team consists of only the encountering thread. However, if nested parallelism is supported and enabled, then the new team can consist of more than one thread.

Synchronization constructs and library routines are available in the OpenMP API to coordinate tasks and data access in parallel regions. In addition, library routines and environment variables are available to control or query the runtime environment of OpenMP programs. OpenMP makes no guarantee that input or output to the same file is synchronous when executed in parallel. In this case, the programmer is responsible for synchronizing input and output statements (or routines) using the provided synchronization constructs or library routines. For the case where each thread accesses a different file, no synchronization by the programmer is necessary.

2.1.2 Memory model

OpenMP provides a relaxed-consistency shared-memory model. All OpenMP threads have access to a place to store and to retrieve variables, called the *memory*. In ad-

dition, each thread is allowed to have its own *temporary view* of the memory. The temporary view of memory for each thread is not a required part of the OpenMP memory model. However, it can represent any kind of intervening structure, such as machine registers, cache, or other local storage, between the thread and the memory. The temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable. Each thread also has access to another type of memory that must not be accessed by other threads, called the *threadprivate memory*.

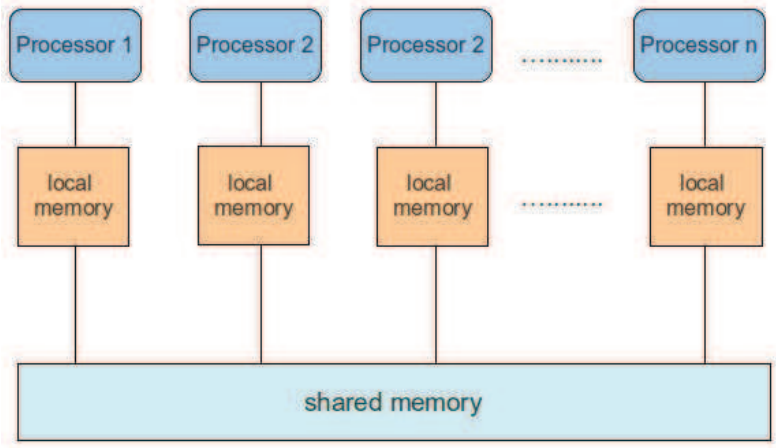


Figure 2.2: Shared-memory architecture.

A directive that accepts data-sharing attribute clauses determines two kinds of access to variables used in the directive's associated structured block: shared and private. Each variable referenced in the structured block has an original variable, which is the variable by the same name that exists in the program immediately outside the construct. Each reference to a shared variable in the structured block becomes a reference to the original variable. For each private variable referenced in the structured block, a new version of the original variable (of the same type and size) is created in memory for each task that contains code associated with the directive. Creation of the new version does not alter the value of the original variable. However, the impact of attempts to access the original variable during the region associated with the directive is unspecified. References to a private variable in the structured block refer to the current task's private version of the original variable. The relationship between the value of the original variable and the initial or final value of the private version depends on the exact clause that specifies it.

A single access to a variable may be implemented with multiple load or store instructions, and hence is not guaranteed to be atomic with respect to other accesses to the same variable. Accesses to variables smaller than the implementation defined minimum size or to C or C++ bit-fields may be implemented by reading, modifying, and rewriting a larger unit of memory, and may thus interfere with updates of variables or fields in the same unit of memory.

If multiple threads write without synchronization to the same memory unit, including cases due to atomicity considerations as described above, then a data race occurs. Similarly, if at least one thread reads from a memory unit and at least one thread writes without synchronization to that same memory unit, including cases due to atomicity considerations as described above, then a data race occurs. If a data race occurs then the result of the program is unspecified. A private variable in a task region that eventually generates an inner nested parallel region is permitted to be made shared by implicit tasks in the inner parallel region. A private variable in a task region can be shared by an explicit task region generated during its execution. However, it is the programmer's responsibility to ensure through synchronization that the lifetime of the variable does not end before completion of the explicit task region sharing it. Any other access by one task to the private variables of another task results in unspecified behavior.

2.1.3 Directive format

OpenMP directives are inserted into a sequential base program to changed into parallel form. The format of these directives are dependent on the base language. This document is restricted to the C/C++ format only. The syntax of a directive is formally as follows:

```
# pragma omp directive-name [clause[ [,] clause]...]  new-line
```

Figure 2.3: OpenMP directive format for C/C++.

Each directive starts with `#pragma omp`. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. In particular, white spaces can be used before and after the `#`, and sometimes white spaces must be used to separate words in a directive. Preprocessing tokens following the `#pragma omp` directive are subject to macro replacement. OpenMP directives are case-sensitive. An OpenMP executable directive applies to at most one succeeding statement, which must be a structured block.

The example below specifies a parallel region that is run by all threads. Variables `x` and `y` are private in each thread.

```
#pragma omp parallel private(x, y)
    C/C++ command block
```

2.1.4 "Hello World!" program

No introduction to OpenMP can be considered as complete without the traditional "Hello World!" program. Below is the source code for C/C++. Note that index

numbers on the left are line numbers, are not part of the code. The program consists in concurrently displaying on the standard output the message "Hello World!".

```
1   #include <omp.h>
2   #include <stdio.h>

3   int main(int argc, char * argv[]){
4         int nthreads, tid;
5         printf("A simple OpenMP program\n");

6         #pragma omp parallel private(tid, nthreads)
7         {
8               tid = omp_get_thread_num();
9               nthreads = omp_get_num_threads();
10              printf("Hello World from thread number %d of %d\n",
                      tid, nthreads);
11        }

12        printf("End of program\n");
13  }
```

Figure 2.4: OpenMP “Hello World!” program for C/C++.

In the above code, the directive in line number 1 includes the OpenMP header file. This file is available in OpenMP compliant compilers, and GNU `gcc` is one of them.

In the flow of the main function, the program is first run as one thread (the master thread), so the function in line 5 displays the message only once. The directive in line 6 specifies a parallel region limited to the following pair of brackets. The number of threads is not specified in the program, so it depends on environment parameters. Clause `private(tid, nthreads)` specifies that the variables `tid` and `nthreads` that contains the values of thread number and the number of threads respectively are private in each thread of the parallel region. In fact, the number of threads are identical in all threads. Therefore, `nthreads` does not need to be private.

Routines `omp_get_thread_number()` and `omp_get_num_threads()` respectively return the thread number of the current thread and the number of threads. These values are displayed in line 10.

After the close bracket at line 11, all threads join and the master thread is the only one to keep on resuming. As a result, function in line 12 is run only once.

To compile an OpenMP program, a compliant compiler with the associated option is required. With `gcc` on Linux, the option is `-fopenmp`. Note that by default, the number of threads specified by the environment equals to 1, so it has to be reset to another value before running the program. In this example, four threads

are used.

Figure 2.5 shows the phases to compile and run the above program. One can see that the displayed messages are not following thread numbers.

Figure 2.5: Compilation and execution of an OpenMP program.

2.2 OpenMP on distributed systems

As presented in the above section, OpenMP is very easy to use. Its execution and memory models are strictly appropriate with SMP architectures such as multiprocessor and multi-core systems. The multiprocessor architecture started in 1962 with the Burroughs D825, a system composed of two processors. After that, it has been implemented on many other servers and mainframes with the number of processors varied from 2 to 16 and more on the hight-end systems. The multi-core architecture appeared later but developed quickly in the last near years. In cheap systems, dual-cores and quad-cores have become popular and in the near future, octo-core systems will be available. In hight-end systems, the number of cores can be far larger, e.g. the Epiphany multicore IP architecture of Adapteva [19] contains up to 4,096 processors on a single chip.

It is important to notice that the speed-up of an OpenMP program depends not only on the number of parallel threads but also on the number of processors in the system. Using a large number of threads does not always increase the execution speed. Normally, the speed-up is the best when the number of threads equals the number of processors. Table 2.1 shows the cases of running a matrix-matrix product on a dual-core and a 16-core machine. One can see that the best case is using two threads for the dual-core and 16 threads for the 16-core.

With the above arguments, one can say that the SMP architecture supports all requirements of shared-memory parallelism: the small size of cheap popular systems; the medium size of servers and mainframes and the big size of hight-end systems. However, beside SMP architectures, distributed-memory architectures are widely

Num. threads		1	2	4	8	16	24	32
t (s)	2 cores	1852.70	1050.01	1054.53	1068.93			
	16 cores	246.18	130.71	63.56	41.40	31.13	31.66	31.17

Table 2.1: Matrix-matrix product execution time on dual-core and 16-core machines.

used and support the special abilities. The most important ones are clusters, grids and clouds. Desktop clusters that have been used since the 90s usually as a solution to establish cheap systems. Cluster is also the architecture to build large systems with capabilities far beyond SMP systems. For example, in the list ranking the 500 fastest machines [4] on November 2011, cluster systems count for 62% and are at positions 1, 4, 5, 9, 10 in the top 10. Grid is an architecture that combines many systems that are more loosely coupled, usually heterogeneous, and geographically dispersed. Finally, although they appeared the latest, clouds have quickly emerged as the solution to build dynamic, easy-to-use and low-cost (at the user's view point) systems.

Thus, porting OpenMP on distributed-memory systems is clearly a good trend to extend the availability of OpenMP. From OpenMP specifications, these main requirements have to be solved for an OpenMP compliance are:

- Implementing the fork-join execution model.
- Implementing the shared-memory model and processing data-sharing directives and clauses.
- Implementing the synchronization mechanism.

Many approaches have been tried, with different advantages and drawbacks. The most important attempts are presented below.

2.2.1 OpenMP on SSI or DSM

2.2.1.1 Use a SSI as the common memory of all threads

The most straightforward approach to port OpenMP on distributed systems is using a Single System Image (SSI). SSI provides an abstract shared-memory layer over the physical distributed-memory architecture. The use of a SSI is not enough to run directly OpenMP programs on different nodes as the original threads are created and run on a single machine. However, this problem can be solved using distributed threads. For example, using the SSI of Kerrighed [20] and the *gthreads*, an implementation of POSIX threads on this architecture, Morin *et al.* have successfully run OpenMP programs without any modification on a commodity cluster [5]. The main advantage of this approach is the ability to easily provide a fully-compliant version of OpenMP. All execution model, memory model and synchronization mechanism

keep the original forms and can be easily implemented. Only a special compiler is needed to link target programs with the *gthreads* library. However, the physical expansion of memory on different machines delays access time [21] and requires a synchronization mechanism. Both of them strongly reduce the global performance. Table 2.2 presents the result of an experiment in [5] that shows that the larger the number of threads, the lower the performance.

Number of OpenMP threads	Execution time in seconds
1	19,78
2	31,57
4	42,71

Table 2.2: Execution times for running the HRM1D code on Kerrighed.

2.2.1.2 Map only a part of thread's memory space on the shared memory area of DSMs

To reduce the impact of the mapping of the whole memory space of threads on a single abstract shared memory, another approach consists in mapping only a part of it on the shared-memory area given by DSMs. This way, the synchronization operations only affect on the considered area and data traffic can thus be significantly reduced. The remaining problem then is how to specify variables that have to be located in shared area.

Extend the OpenMP API

The first approach consists in extending the OpenMP API. For example, the Cluster OpenMP solution [11] of Intel adds an additional directive — the `sharable` directive. This directive identifies variables that are referenced by more than one thread and are then managed by the DSM. Some variables are automatically made `sharable` by the compiler like those that are allocated on the stack and the pass-by-value formal parameters. Other variables must be explicitly declared as `sharable`, e.g. file-scope variables in C and C++.

Use of the *shmemory* model

The *shmem* model provided by SCASH [22] is the approach used in *Omni* compiler system [23]. With this model, variables declared in the global scope are private. The shared address space must be explicitly allocated by the shared memory allocation primitive at run time. To compile an OpenMP program into the shmem memory model, the compiler translates the code to allocate global variables in the shared address space at run time. The compiler processes OpenMP programs by means of the following three steps:

1. All declarations of global variables are converted into pointers containing the address of the data in the shared address space.

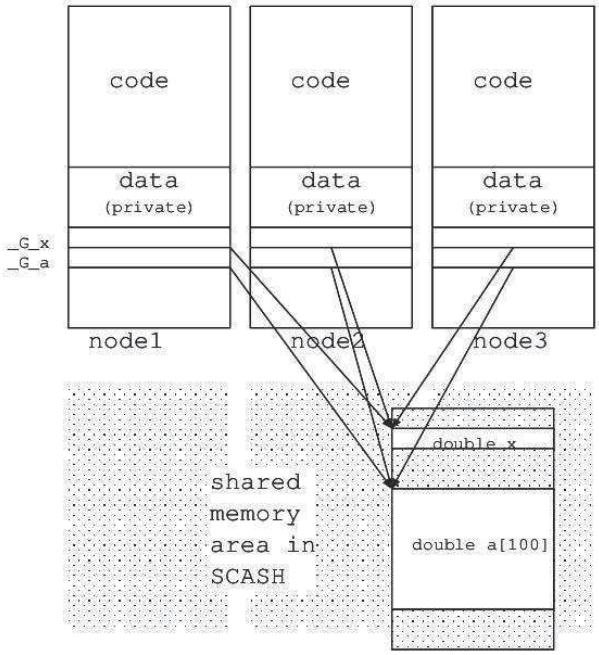


Figure 2.6: shmem memory model.

2. The compiler rewrites all references to global variables to the indirect references through the corresponding pointers.

3. The compiler generates the global data initialization function for each compilation unit. This function allocates objects in the shared address space and stores these addresses in the corresponding indirect pointers.

With the shmemory model and the transformation of shared variables as presented above, Omni enables OpenMP programs to run transparently on cluster environments. However, as shared variables are located in a common memory and used directly by all threads located anywhere across the system, the performance is reduced by the relay access time and the management of concurrent accesses. SCASH can deliver high performance for an OpenMP program if the placement of data and the computation are such that data needed by each thread is local to the processor on which thread is running. To use this mechanism, Omni has to extend OpenMP with a set of directives to allow programmers to specify the placements of data and computations on the shared address space and a new loop scheduling clause, `affinity`, to schedule the iterations of a loop onto threads associated with the data mapping.

Use a RC model

Built on top of a DSM too, the solution proposed by Sven Karlsson *et al.* [6] and Jie Tao *et al.* [7] do not use a common shared area for OpenMP shared variables.

However they use the HLRC model of DSM systems to implement the OpenMP relaxed-consistency shared model. The HLRC model uses a page-based mechanism to ensure the consistency between the nodes in the system. This means that it uses the virtual memory page protection mechanism to detect accesses to shared memory and memory contents are replicated on a virtual memory page basis. In order to simplify the protocol, each page in an HLRC system has a home node which always holds an up-to-date copy of the page. Thus, whenever a processor accesses a page which is not present locally, a copy is retrieved from the home node. Several nodes can have write permissions to the same page in which case several data structures are used to locally keep track of changes to the page. These changes are transferred at synchronization points to the home node where they are merged. Synchronization points also forces updated pages to be invalidated. Synchronization points normally occur when synchronization primitives are executed.

Although the HLRC model can reduce the impact on performance of accesses to a common shared-memory area, this approach meets some problems. The first one is the difficulty to automatically identify all shared variables as the OpenMP API does not require an explicitly declaration of shared variables while the DSMs do. The second one concerns the implementation a high-performance `flush` directive and for the initialization shared variables. Furthermore, the `reduction` clause and the `atomic` directive also cause difficulties while being implemented on this model. For more details about these problems, refer to [6] and [7].

2.2.1.3 TreadMarks - A solution for Clusters of SMPs

TreadMarks [31] is a DSM that runs at the user level on most Unix and Windows NT-based systems. It provides a shared memory as a linear array of bytes via the release and multiple-writer consistency. In its simplest form, running TreadMarks on a network of SMPs could be archived by simply executing a process on each processor of each multiprocessor node and having all of these processes communicate through message passing. This direction, however, fails to take advantage of the hardware shared memory on the multiprocessor nodes. In order to overcome this limitation, a new version of TreadMarks has been built, in which POSIX threads is used to implement parallelism within a multiprocessor. As a result, the OpenMP threads within a multiprocessor share a single address space. Thus, memory areas are shared by default within processors on the same nodes and are private between the different nodes of the cluster. All nodes use the shared area provided by the TreadMarks DSM system.

Using the shared memory provided by the TreadMarks DMS, Y. C. Hu and *.al* presented in [32] an implementation of OpenMP for clusters of SMPs. In this solution, translation was quite straightforward: OpenMP synchronization directives were replaced by TreadMarks synchronization operations; parallel regions in OpenMP were encapsulated into separate functions and translated into the fork-join code. When implementing OpenMP's data environment, actual parameters provided to procedures were converted to shared data if necessary, as the translator

does not perform interprocedural analysis and cannot determine whether they are accessed from within a parallel region. Shared variables are allocated on the shared heap; private variables are allocated on TreadMark's stacks.

While testing on a cluster composed of four quad-processor SMP nodes, TreadMarks gives a high performance in the range from 7 to 30% of the MPI version. However, this solution has the same drawback as the ones using the shmem model and the HLRC model, that is the problem to automatically specify all shared variables that should be allocated on the system shared-memory area. Furthermore, the number of nodes for the tests was quite small, so that the latency of access to the system shared area might not have a significantly impact on the global performance.

2.2.2 Based on MPI

Due in part to its high performance, MPI has become the *de facto* programming tool for distributed-memory systems. One of the main drawbacks of MPI is the need to explicitly specify the data transfer between the nodes. On one hand, this helps to increase the performance thank to the reduction of the amount of data transaction on network. On the other hand, this requires efforts from programmers and this is highly time consuming. As a result, some attempts [8][9] tried to implement OpenMP on top of MPI, with the objective of exploiting its high performance.

The main challenge when porting OpenMP on MPI is implementing the OpenMP's shared-memory model. MPI uses a distributed memory model where each node of a system owns an independent memory space and there is no shared memory. In MPI programs, programmers have to explicitly write instructions for data exchange to take place, while in OpenMP all variables are shared by default which means that a modification on a variable by one thread is seen by the other threads at least after an implicit/explicit barrier. To solve this problem, two different approaches have been used in [8][9] each with their own advantages and drawbacks.

2.2.2.1 Extending OpenMP to specify exchanged data and to guarantee memory consistency

The approach of B. D. Martino *et al.* [8] consists in extending OpenMP with `llc` additional directives. They develop the *llCoMP* compiler, a source-to-source compiler implemented on top of MPI. This compiler translates a code annotated with `llc` directives into a C code with explicit calls to MPI routines. The resulting program is then compiled using the native back-end compiler, and properly linked with the MPI library. The `llc` language follows the One Thread is One Set of Processors [24] (OTOSP) computational model. The OTOSP model is a distributed memory computational model where all memory locations are private to each processor. A key concept of the model is the processor set. At the beginning of the program (and also in its sequential parts), all available processors in the system belong to the same unique set. Processor sets follow a fork-join model of computation: sets are divided (fork) into subsets as a consequence of the execution of a parallel construct, and they

join back together at the end of the execution of the construct. At any point of the code, all processors belonging to the same set replicate the same computation; that is, they behave as a single thread of execution. When different processors (sub-)sets join into a single set at the end of a parallel construct, partner processors exchange the contents of the memory areas they have modified inside the parallel construct. The replication of computations performed by processors in the same set, together with the communication of modified memory areas at the end of the parallel construct, are the mechanisms used in OTOSP to guarantee a coherent image of the memory.

llcCoMP uses two additional directives for shared variables. Any shared variable in the left-hand side of an assignment statement inside a parallel loop should be annotated with an `llc result` or `nc result` clause. Both clauses are employed to notify the compiler that a region of the memory is potentially modifiable by the set of processors executing the loop. Their syntax is similar: the first parameter is a pointer to the memory region (addr), the second one is the size of that region (size), and the third parameter, only present in `nc result`, is the name of the variable holding that memory region. Directive `result` is used when all the memory addresses in range `[addr, addr+size]` are (potentially) modified by the processor set. This is the case, for example, when adjacent positions in a vector are modified. If there are write accesses to non-contiguous memory regions inside the parallel loop, these should be notified with the `nc result` clause.

llCoMP uses Memory Descriptors (MD) to guarantee the memory consistency at the end of the execution of a parallel construct. MD are data structures based on queues which hold the necessary information about memory regions modified by a processor set. The basic information held in MD are pairs `(address, size)` that characterize a memory region. In most cases, the communication pattern involved in the translation of a `result` or `nc result` is an all-to-all pattern. The post-processing performed by a processor receiving a MD is straightforward: it writes the bytes received in the address annotated in the MD.

Although this allow to obtain a very high performance as shown in some experiments [9], this approach has some drawbacks. The major weak points are the requirement towards the programmer's specification for data exchange and the guarantee of the memory consistency. It looses the easiness to use which is the most important characteristics of OpenMP. Furthermore, with the OTOSP model, it is difficult to implement some OpenMP directives such as the `single`, `master`, `flush` directives.

2.2.2.2 Automatic translation using the *Partial Replication model*

A. Basumallik and R. Eigenmann in [9] use the *Partial Replication model* to avoid programmers to explicitly specify exchanged data between nodes. In this model, shared data is allocated on all nodes. However, no shadow copy or management structures need to be allocated. A special mechanism is used to ensure that only a part of the shared data is sent from the producer (the node producing data) to the

potential consumers (the nodes that may use the data in the future).

The execution model associated with the partial replication is SPMD, with the following characteristics:

- All participating processes redundantly execute serial regions and parallel regions demarcated by the `omp master` and `omp single` directives. Iterations of OpenMP `parallel for` loops are partitioned between processes using block-scheduling.
- Shared data is allocated on all processes. There is no concept of an owner for any shared data item. There are only producers and consumers of shared data.
- At the end of the parallel constructs, each participating process communicates the shared data it has produced to the other processes that may use them in the future.

The method that allows producers forwarding data to all potential consumers ensures that processes always make local accesses when they read shared data. This method may result in redundant communication where accesses are not completely analyzable. An exception to redundant execution of the serial regions is the use of file I/O. Reading from files is redundantly done by all processes (assuming that the file-system is visible to all processes). Writing to a file is done by only one process at a time (the process with the smallest MPI rank).

The translation from an OpenMP program to a MPI version contains three main steps.

1. *Interpretation of OpenMP Semantics Under Partial Replication.* The compiler converts the OpenMP program into a SPMD form, interprets OpenMP directives and performs the required partitioning work. Constructs for expressing parallelism are translated by extracting the parallel region or the loop body into a separate subroutine and inserting calls to a scheduling and dispatch runtime library. When the program execution reaches one of these constructs, the master thread invokes the corresponding subroutine on all threads, after explicitly invoking a synchronization barrier for ensuring shared memory consistency before the parallel region starts. After the work partitioning, the compiler builds the set of shared data used in the program using the Inter-Procedural Analysis Algorithm for Recognizing Shared Data [25]. Then the determined shared variables are allocated in the shared memory area of DMS.
2. *Generation of MPI Messages using Array Dataflow.* The compiler inserts MPI calls to communicate data from producers to potential future consumers. To solve the producer-consumer relationships, the compiler has to perform an array-dataflow analysis. The compiler builds bounded regular section descriptors [26] to characterize accesses to shared arrays. Then, the compiler creates a producer-consumer flow graph which is used to solve the producer-consumer

relationships for shared data. This graph is finally used to conform to the relaxed memory consistency model of OpenMP.

3. *Translation of Irregular Accesses.* A major challenge in translating OpenMP applications directly to message passing is the handling of irregular accesses. Irregular accesses are the reads and the writes that cannot be analyzed precisely at compile-time. A technique based on the property monotonicity is used for handling irregular accesses to shared arrays at compile-time. However, there may be applications where this compile-time scheme is not applicable, either because the indirection function is not provably monotonic or because the irregular write is not in the form of *ARRAY[indirection function]*. In such cases, a runtime mechanism is used as a fallback. In this mechanism, the compiler inserts code to record the writes at runtime. The inserted code allocates a buffer on each process and at runtime puts in *(location,value)* pairs for the written elements. At the end of the loop containing irregular writes, processes intercommunicate this buffer.

With the performance evaluation shown in [9], this implementation of OpenMP based on the translation into MPI involves a high performance. However, in their compiler, which extends the Cetus Compiler Infrastructure [27], only steps 1 and 2 are implemented. Step 3 and some other optimizations have to be manually. Furthermore, the SPMD execution model used in this approach can cause difficulties to implement the constructs running on one thread only such as `single` and `master`. These two facts prevent this approach from becoming a fully compliant implementation of OpenMP.

2.2.3 Based on Global Array

A translation of OpenMP to Global Array (GA) was presented by L. Huang *et al.* [10][29][30].

GA [28] is a collection of library routines that was designed to simplify the programming methodology on distributed memory systems. GA simplifies parallel programming by providing users with a conceptual layer of virtual shared memory for distributed memory systems. Programmers can write their parallel program on clusters as if they have a shared memory access, specifying the layout of shared data at a higher level. However, it does not change the parallel programming model dramatically since programmers still need to write SPMD style parallel code and deal with the complexity of distributed arrays by identifying the specific data movement required for the parallel algorithm. GA programs distribute data in blocks to the specified number of processes. The current GA is not able to redistribute data. Before a region of code is executed, the required data must be gathered from the participating processes; results are scattered back to their physical locations upon completion. GA relies upon MPI to provide it with the execution context.

L. Huang *et al.* [10][29][30] showed that almost all OpenMP directives, library routines and environment variables can be translated into GA or MPI library calls

at source level. Exceptions are those that dynamically set/change the number of threads, such as `omp_set_dynamic`, `omp_set_num_threads` since they would lead to performing data redistribution and GA is based upon the premise that this is not necessary.

The general approach to translating OpenMP into GA is to declare all shared variables in the OpenMP program to be global arrays in GA. The most important transformations are:

- OpenMP parallel regions are translated into GA by invoking `MPI_Init` and `GA_INITIALIZE` routines to initialize processes and the memory needed for storing distributed data array. To implement OpenMP parallel loops in GA, the generated GA program reduces the loop bounds according to the specified schedule so as to assign work. Based on the calculated lower and upper bounds, and the array region accessed in the local code, each process in the GA program fetches a partial copy of global arrays via `GA_GET`, performs its work and puts back the modified local copy into the global location by calling `GA_PUT` or `GA_ACCUMULATE`.

- Private variables can be declared as local variables that are naturally private to each process in a GA. If the parallel region contains shared variables, the translation will turn them into distributed global arrays in the GA program by inserting a call to the `GA_CREATE` routine. OpenMP `firstprivate` and `copyin` clauses are implemented by calling the GA broadcast routine `GA_BRDCST`. The reduction clause is translated by calling GA's reduction routine `GA_DGOP`.

- GA synchronization routines replace OpenMP synchronizations. As an OpenMP synchronization ensures that all computations in the parallel construct has completed, a GA synchronization does the same but also guarantees that the required data movement has completed to properly update the GA data structures. GA locks and Mutex library calls are used to protect a critical section and they are used to translate the OpenMP `critical` and `atomic` directives.

- The OpenMP `flush` directive is implemented by using GA `put` and `get` routines to update shared variables. This could be implemented with the `GA_FENCE` operations if more explicit control is necessary. GA provides the `GA_SYNC` library call for synchronization; it is used to replace OpenMP `barrier` as well as implicit barriers at the end of OpenMP constructs. The only directive that cannot be efficiently translated into equivalent GA routines is the OpenMP `ordered`. `MPI_Send` and `MPI_Recv` are used to guarantee the execution order of processes if necessary. Since GA works as a complement of MPI, and must be installed on a platform with GA, there is no problem invoking MPI routines in a GA program.

- The translation of the sequential program sections (serial regions outside parallel regions, OpenMP `single`, `master`, and `critical` constructs) become non-trivial besides that of parallel regions. The program control flow must

be maintained correctly in all processes so that some parts of the sequential section have to be executed redundantly by all processes. Subroutine/function calls in sequential regions need to be executed redundantly if these subroutines/functions have parallel regions inside. Three different strategies have been identified to implement the sequential parts, check the special requirements and guarantee memory consistency: the master execution running only on the master process, the replicated execution running redundantly the same code on all processes and the distributed execution on each processor that executes a portion of the work of the sequential part according to constraints of the sequential execution order.

Afters the results shown in [10][29][30], the approach of translating OpenMP into Global Array gives high performance. However, it meets a major drawback: the requirement to explicitly specify shared variables which are implicitly shared according to the OpenMP specification. This problem prevents it to become a fully OpenMP compliant implementation on distributed systems.

Table 2.3 summarizes the main properties of the above implementations of OpenMP on distributed systems.

2.2.4 Conclusion

There have been many attempts to port OpenMP on distributed memory systems. However, none have successfully provided a fully-compliant and high-performance OpenMP solution. The approach using SSI could have led to a fully compliant solution but performance are very low. Approaches accepting a local memory in threads meet the problem of specifying automatically shared variables which should be allocated on the system shared-memory region. Some of them also meet other problems such as the implementation to run the sequential sections on a single thread; the need to use additional directives... As a consequence, finding an fully-compliant implementation of OpenMP with high performance for distributed system remains a challenge.

Method	Platform	Principle	Advantages	Drawbacks
SSI [5]	DSM	uses a SSI as a global memory for threads	full OpenMP compliant	weak performance
Cluster OpenMP [11]	DSM	maps only shared variables on the global shared memory	high performance	use of additional directives
SCASH [23]	DSM	maps only shared variables on the global shared memory	high performance	use of additional directives
RC model [6] [7]	DSM	uses the HLRC model to implement the OpenMP memory model	high performance	difficulty to automatically identify shared variables and the implementation of `flush`, `reduction`, `atomic`...
TreadMarks [32]	DSM, cluster of SMPs	extends the original TreadMarks	high performance	difficulty to automatically identify shared variables
llCoMP [8]	MPI	translates to MPI	high performance	use additional directives; difficulty to implement `single`, `master`, `flush`
Partial Replication model [9]	MPI	translates to MPI; automatically specify exchanged data between nodes	high performance	not completely implemented; difficulty to implement `single`, `master`...
GA [10][29][30]	GA	translates to Global Array	high performance	difficulty to automatically identify shared variables

Table 2.3: The implementations of OpenMP for distributed systems.

2.3 Checkpointing

Checkpointing is the act of saving the state of a running program so that it can be resumed later in time [33].

To take the checkpoint of a process, it is necessary to save all its state information. When a program is executing, its state is composed of the values in memory, the CPU registers, and the state of the process in the system such as the ID of the process, the list of open file descriptors, signal masks... In Linux, the virtual space address is composed of two parts: the kernel address space and the user address space which also is divided into four segments: the text segment, the data segment, the heap and the stack. Typically, values in the kernel part and the text segment do not change during execution time, so they do not need to be saved in a checkpoint. Figure 2.7 shows the principle of saving a checkpoint of a Linux process.

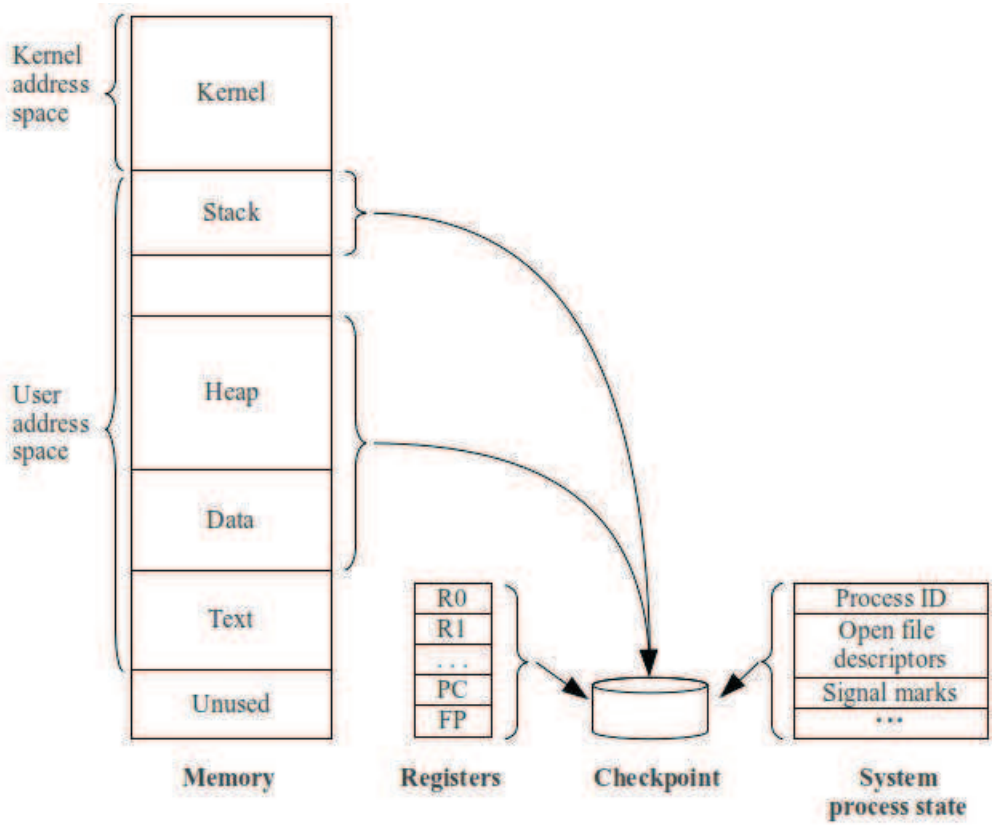


Figure 2.7: Saving a checkpoint of a Linux process.

2.3.1 Applications of checkpointing

Checkpointing is used in many applications. Belows are the most important ones [33][34][35].

2.3.1.1 Fault-Tolerance (Rollback recovery)

Along with the *recovery* technique, checkpointing is used in *fault-tolerance* applications that are used to resume the execution of a program to reduce the amount of lost work after a failure. This is typically the main application of checkpointing.

Typically, at periodic intervals while the program is executing, checkpoints are taken and saved in a stable memory. If a failure occurs, a recovering tool can resume the program from the last saved checkpoint, thus losing at most an interval's work of computation. This application is very powerful since it has no binding with the type of failure which can be hardware or software, or a power failure.

2.3.1.2 Process migration

In this application, a checkpoint is sent from one processor to another on which the process is resumed from the checkpointed state. After sending the checkpoint, the process in the initial processor is terminated. Process migration is very useful for load balancing, i.e. to move a process from a heavily loaded processor to a lightly loaded one. It can also be used for some other cases such as changing the hardware without having to stop down the program for long.

2.3.1.3 Periodic backup

As in fault-tolerance applications, checkpoints are periodically taken and saved. Using these checkpoints, users can return to previous states of the program.

2.3.1.4 Debugging

To help programmers find the errors when a program exit abnormally, some debug tools automatically create checkpoints called *core* files. Using these files, programmers can identify the bugs or locate the regions with potential bugs.

Another type of debug using checkpoints is *replay debugging*. In this case, checkpoints are periodically taken allowing users to turn the state of the examined program back to any previous state.

2.3.1.5 Job swapping

In this application, many jobs are sharing (not concurrently) a resource. A running job takes a checkpoint and then stops down to release the resource to another job. After waiting for awhile, the stopped job can resume its execution from the saved checkpoint and so on.

2.3.2 Different levels of checkpointing

Checkpointing may be categorized [33][36] in two main groups that are *user level* or *system level*, depending on the space level in which it is implemented. User-level checkpointing, in turn, may be divided into two subcategories which are *transparent*

and *non-transparent*, based on level of requirements the user's applications perform the checkpointing.

2.3.2.1 System-level checkpointing

In this category, the mechanism is implemented and performed by the operating system itself. Typically, any program can be checkpointed by the operating system without any effort from the programmer or the user. For example, standard process pre-emption (i.e. making a process relinquish the CPU and putting it on the ready queue) can be viewed as a simple form of OS checkpointing.

Most operating systems do not implement checkpointing beyond process scheduling. There are a few notable exceptions, such as Unicos [37], KeyKOS [38] and fault-tolerant Mach [39], which implement rollback recovery, and Sprite [40], Mosix [41], Gobelins [42], CHPOX [43] which are distributed operating systems that include process migration as a primitive operation, Kerrighed [44] which implements checkpoint/recovery mechanisms for a DSM cluster system.

2.3.2.2 User-level transparent checkpointing

In this case, checkpointing is performed in user space by the checkpointer without modification of the source code of checkpointed programs. Transparency is usually achieved by compiling the application program with a special checkpointing library, although other methods are possible, such as rewriting executable files [45] or injecting checkpointing into running processes [46]. The major drawback of user-level checkpointing is the lack of ability of the checkpointer to access and to recovery some system data such as the ID of the program. More details on these problems are presented in [47] and [48].

Table 2.4 shows some examples of user-level transparent checkpointers.

2.3.2.3 User-level non-transparent checkpointing

This type of checkpoint is also performed in user space, but programmers have to insert explicit checkpointing directives in the source code of the checkpointed program, usually with the help of libraries and preprocessors. On one hand, this increases the work of programmers. On the other hand, this increases flexibility and performance. Programmers can specify exactly the information that have to be saved and recovered and this usually leads to better performance. Another advantage of this checkpointing type is the ability to save checkpoints in a machine-independent format, which allows checkpoints be resumed on machines with a different architecture.

Table 2.5 shows examples of user-level non-transparent checkpointers.

2.3.3 Complete Checkpointing vs. Incremental Checkpointing

Complete Checkpointing is a technique in which all the information associated with

Name	Functionality	Computing platform
Libckpt [49]	Fault-tolerance	Uniprocessors
Condor [50]	Process migration	Uniprocessors
Igor [51]	Debugging	Uniprocessors
ckpt [46]	Fault-tolerance	Uniprocessors
Manetho [52]	Fault-tolerance	Message-passing distributed systems
CoCheck [53]	Fault-tolerance/migration	Message-passing distributed systems
CGS [54]	Fault-tolerance	Message-passing distributed systems
Ickp [55]	Fault-tolerance	Intel iPCs/860
CLIP [56]	Fault-tolerance	Intel Paragon
ZAP[57]	Process migration	Distributed systems
Bproc[58]	Process migration	Distributed systems
DMTCP[59]	Fault-tolerance	Distributed systems & Desktop
MPICH-V[60]	Fault-tolerance	Message-passing distributed systems
Pcl[61]	Fault-tolerance	Message-passing distributed systems

Table 2.4: Examples of user-level transparent checkpointers.

Name	Functionality	Computing platform
Libft[62]	Fault-tolerance	Distributed systems
Dome[63]	Fault-tolerance / load balancing	Distributed systems
PUL[64]	Fault-tolerance	Distributed systems
Calypso[48]	Fault-tolerance / load balancing	Distributed systems
COSMOS[65]	Fault-tolerance	Distributed systems
PM2[66]	Process migration	Distributed systems
DEMOS/MP[67]	Process migration	Distributed systems

Table 2.5: Examples of user-level non-transparent checkpointers.

process is saved in each checkpoint. Its major drawback is the redundancy of data when several checkpoints are taken consequently without many changes in process space. Moreover, this is not efficient, for example for both the text segment and data that can be easily retrieved from the executable code of the program.

With the *Incremental Checkpointing* technique [68][69], only the information that have been modified since the beginning of the execution or since the previous checkpoint are effectively considered. Most of incremental checkpointers use the page-fault mechanism to specify the modified data in which, following a checkpoint, all pages in memory are set to be read-only. When the program attempts to write a read-only page, an access violation occurs, and the checkpointer processes the resulting interrupt by storing the identity of the page in a list, and resetting its protection to read-write. At the time to save the checkpoint, only the pages that have caused access violations (those stored in the list of accessed pages) are stored in the checkpoint file.

Incremental checkpointing improves performance over complete checkpointing if the few pages saved by the checkpointer are smaller than the penalty for setting the page protections and processing access violations. Unless almost all memory is altered between two checkpoints, this is usually the case. More details about this technique are presented in the next chapter.

2.4 CAPE with complete checkpoints (CAPE-1)

The initial idea behind CAPE is the use of checkpoints to implement the OpenMP fork-join model. In this model, a program initially run on a single thread called the master thread. Whenever the program meets a parallel region, the master thread is duplicated several times to create slave threads using instruction `clone` and the job to be computed in the parallel section is distributed among them. The `join` phase of the model is associated with the termination of slave threads after they have finished their particular job and updated their result to the master's memory.

Figure 2.8 presents the relation between the OpenMP fork-join model and the CAPE fork-join model for a program's section composed of three parallel parts executed in three different threads in case of OpenMP and in three processes in the case of CAPE. The reason for replacing threads by processes is explained in the Sec. 2.2.4. Note that in this model, CAPE is restricted to programs containing parallel sections that matches the Bernstein's conditions and processes may be located on different machines. The principle of CAPE is described below.

At the beginning, the program runs on a single process called the master process. When this process meets a parallel region:

- `fork` phase: the job is divided into smaller parts or chunks by the master process. At the beginning of each chunk, the master process creates a complete checkpoint and sends it to a distant machine to create a slave process. Then, the master process jumps to the next chunk if any. After the last chunk, the master process waits for the results from slave processes. The distant machines

use the received checkpoint to resume the program at the begin of the chunk. After the chunk is completed, the result is extracted using another checkpoint and reported to the master process. Then, the slave process terminates its execution.

- `join` phase: after all slave processes have terminated their execution and the master process has received all results from the slave processes, the master process resumes its execution to the next region of the program. At that time, there is only one active process in the system.

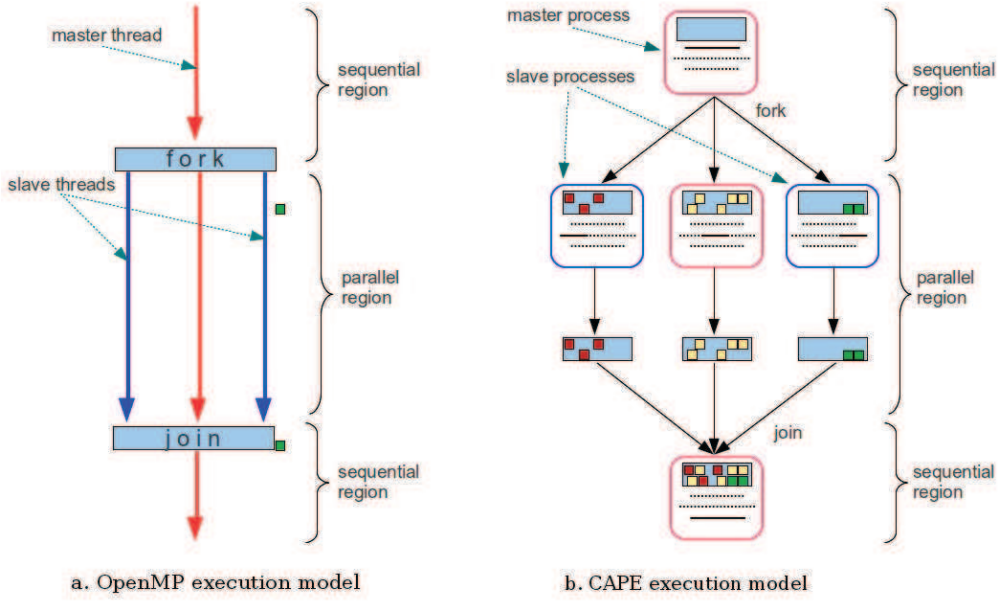


Figure 2.8: OpenMP fork-join model vs. CAPE fork-join model.

The CAPE model offers two main advantages:

- It is compatible with the relaxed-consistency shared-memory model of OpenMP: the memories in the master process and in slave processes are matching the *memory* and the *temp views* in the master and the slave threads of the OpenMP model respectively. During the execution of the job of each chunk, slave processes only use their own memory spaces. As a result, this model can reduce the access time to the common memory and thus increase the execution speed of the program. Note that synchronization between the memory of slave processes at execution time is not necessary since the need to verify the Bernstein's conditions. The solution to overcome this restriction is presented in Chap. 5.
- The ability to automatically distribute parallel sections and collect results from slave processes (based on checkpoints) overcomes the drawbacks of the

other approaches, based on MPI or Global Array for example. In fact, as the identification of memory locations that have been modified during the execution of a slave process, that might be called the result of the execution of a slave thread, is automatically extracted from the difference of two checkpoints while the same result can be very hard to achieve when implementing OpenMP on distributed systems using the other approaches.

2.4.1 CAPE prototype for `parallel for` loops

By using complete checkpoints, the above model can easily be installed and in this document, it is referred to as CAPE-1 to differ with the implementation using incremental checkpoints called CAPE-2 and presented in the Chap. 4. In the `fork` phase, complete checkpoints contain process space and thus, when used to resume the execution in the slave processes, they create the same process space as the space created when using system calls `fork` or `clone` to duplicate the process. The extraction of results in a slave process, after finishing its job, can be done by comparing the checkpoint taken at this time and the initial checkpoint that was used to resume the execution of the slave processes.

Figure 2.9 presents the effective transformation that is performed on a code that specifies a `parallel for` with all loop iterations `D` satisfying the Bernstein's conditions when using OpenMP directives. The `parallel section` construct can be implemented by using a similar prototype [12] or by a translation into an intermediate `parallel for` form. To ease the presentation, assume that the number of loop iterations is equal to the number of slave processes. The parent, i.e. the master node, is in charge of managing the slaves only and does not execute any loop iteration in the parallel part. This is not mandatory and the master node can also take part in the execution of one or more loop iterations. The translation is based on the following functions:

- `create ( < file > )` generates a checkpoint and saves it in the file provided as a parameter. The value returned by the function is used to identify whether the function has just created the checkpoint and returned, or the process has been created after resuming the execution from the checkpoint. This function is very similar to the `fork` system call, except that `create` returns `TRUE` after generating the checkpoint and `FALSE` after resuming the execution from the checkpoint.

- `copy ( < file_1 > , < file_2 > )` duplicates the content of a file into another one.

- `diff ( < file_1 > , < file_2 > , < file_3 >)` saves into the last file provided as a parameter the list of modifications that should be applied on the first file to obtain the second one.

- `merge ( < file_1 > , < file_2 > )` applies the list of modifications saved in the second file provided as a parameter to the checkpoint file provided as

```
# pragma omp parallel for
    for ( A ; B ; C )
        D
```

↓ automatically translated into ↓

```
parent = create ( original )
if ( !  parent )
    exit
copy ( original, target )
for ( A ; B ; C )
    parent = create ( before_i )
    if ( parent )
        ssh host_x restart ( before_i )
    else
        D
        parent = create ( after_i )
        if ( !  parent )
            exit
        diff ( before_i, after_i, delta_i )
        merge ( target, delta_i )
        exit
parent = create ( final )
if ( parent )
    diff ( original, final, delta )
    wait_for ( target )
    merge ( target, delta )
    restart ( target )
```

Figure 2.9: Template for OpenMP `parallel for` loops with complete checkpoints.

the first parameter.

- `wait_for ( < file > )` returns after the file whose name is provided as a parameter is available.
- `restart ( < file > )` resumes the execution of the current process from the checkpoint file provided as a parameter.

Note that the operation that consists in resuming the execution of checkpoints generated for each loop iteration, the line in italic in Fig. 2.9, is executed on the master node but delegated to an external process in charge of managing the distribution of processes on a set of remote resources. BOINC [70], used in the scope of the Seti@Home project, is probably one of the most famous tool aiming at distributing works among a set of computing resources. However, many other solutions are also available such as AC2 [71], Nimbus [72][73], OpenNebula [74] and KOALA [75][76]. Details of this prototype including its proof and its evaluation are presented in [12][13]. [14] presents a model to determine the optimal number of nodes to use to minimize the execution time of a parallel program based on CAPE.

2.4.2 Some possible improvements for CAPE

Although the feasibility and the consistency of CAPE have been proved both by a theoretical analysis and by practical tests on its implementation, this prototype is not optimal as some elements significantly reduce its global performance and limit its applicability. The most important problems are:

1. *The large amount of data transfered over network.* The master process needs to send one snapshot of the process image in the form of a complete checkpoint to each slave node. This transfer leads to two important elements that increase the amount of data sent over the network. First, these snapshots are large because they contain all the information related to the process: shared libraries, program's code, program's data, etc. Second, they are different for each loop iteration, so the master process has to sequentially create and send them to the slave processes.

2. *The large number of comparisons in complete checkpoints to extract the result on slave nodes.* This is critical when large checkpoints are generated and the final amount of data is small. In this case, a lot of comparison operations are performed that significantly increase the execution time on slave nodes and reduce the global performance.

3. *A large amount of memory space is required to store the temporary data.* In this prototype, several checkpoints need to be saved: `original`, `target`, `before`. In case of big checkpoints, they significantly affect the memory usage.

4. *The delay involved to start/resume the processes.* Both slave processes and the master process have to start/resume in each parallel region. The first ones

happen to start slave processes from the received checkpoints. The second happen on the master process after it receives the results from slave processes, to include all these results in its process space. The ability to directly update a process memory would avoid this drawback.

5. *The requirement to verify the Bernstein's conditions.* OpenMP allows the synchronization of memory between the threads in parallel regions. In CAPE-1, there is no mechanism to implement this ability. Furthermore, OpenMP directives and clauses related to data-sharing problems such as `private`, `threadprivate`, `firtsprivate`, `lastprivate`, `reduction`... are not considered in this prototype. These two restrictions limit the types of programs that can be executed with CAPE.

The next chapters aim at presenting the solutions to overcome these drawbacks. First, Discontinuous Incremental Checkpointing, presented in Chap. 3 provides the ability to extract the changes in specified regions of programs. Based on this tool, a new prototype called CAPE-2, presented in Chap. 4 bypass problems 1 to 3. Problem 4 is solved by introducing function `inject` in this checkpointing technique, that directly includes the changes contained in checkpoints into the process space without the need to restart them. The shared-memory model and the processing of data-sharing primitives and clauses, presented in Chap. 5 finally solve the problem 5.

Chapter 3

Discontinuous Incremental Checkpointing

Contents

3.1 Linux memory architecture

Checkpointing technique is used in all architectures and all operating systems even though it is not always implemented in the same way. The following description is limited to Linux with x86 microprocessors.

3.1.1 Memory address

80x86 microprocessors distinguishes three kinds of addresses [77]:

- *Physical addresses* are used to address memory cells in memory chips. They correspond to the electrical signal sent along the address pins of the microprocessor to the memory bus. Physical addresses are represented as 32-bit or 36-bit unsigned integers.
- A *linear addresses (also known as virtual address)* is a single unsigned integer. Linear addresses are usually represented in hexadecimal notation. In 32-bit systems, their values range from `0x00000000` to `0xFFFFFFFF`. In most cases, Linux programmers use linear address to refer to a memory position.
- *Logical addresses* are included in the machine language instructions to specify the address of an operand or of an instruction. This type of address embodies the well-known 80x86 segmented architecture that forces MS-DOS and Windows programmers to divide their programs into segments. Each logical address consists of a segment and an offset (or displacement) that denotes the distance from the start of the segment to the actual address. In Linux, addresses begin with `0x00000000`, so logical addresses and linear addresses are the same as the value of the offset field of a logical address is always the same of the corresponding linear address.

The Memory Management Unit (MMU) translates a logical address into a linear address by means of a hardware circuit called the segmentation unit; subsequently, a second hardware circuit called the paging unit translates a linear address into a physical address (see Fig. 3.1). This translations is transparent to programmers in most cases and thus they do not need to be interested in the physical address.

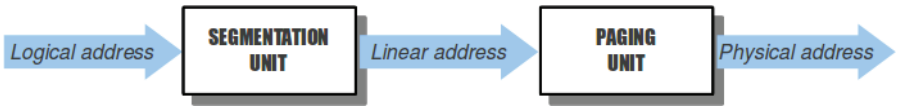


Figure 3.1: Logical to Linear, and Linear to Physical address translations.

3.1.2 The Process Address Space

A process is a program in execution. An executable program on a disk contains a set of binary instructions to be executed by the processor together with the data used in the program. While running, all these elements are loaded in the process memory space. Furthermore, the process memory also contains the shared libraries which are included in program, the kernel code and data. Figure 3.2 [78] shows the layout of a process memory, in which the gray regions represent virtual addresses that are mapped to the physical memory, whereas white regions are unmapped. Belows are the description for each region.

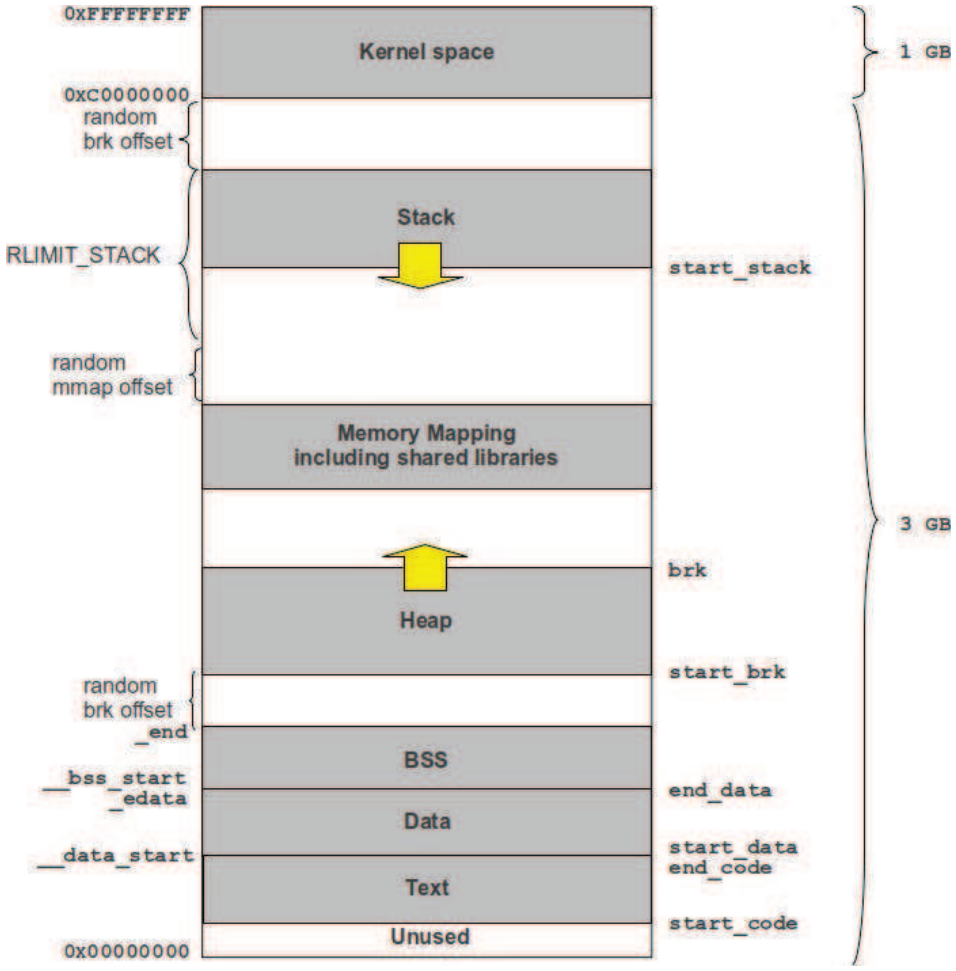


Figure 3.2: Linux process memory layout.

- *Text, BSS and data segments.* A process image starts with the program code and data. Code and data consists in the program instructions, and both initialized and uninitialized local static and global data respectively. The address of each region is specified at compile time and unchanged during execution time. Typically, the text segment is also unchanged at the execution time.

- *Stack segment.* This is the topmost segment in the process address space, which stores automatic local variables and function parameters in most programming languages. Calling a method or function pushes a new stack frame onto the stack. The stack frame is destroyed when the function returns. This simple design, made possible thanks to the LIFO order, means that no complex data structure is needed to track stack contents – a simple pointer to the top of the stack will do. Pushing and popping are thus very fast and deterministic. Also, the constant reuse of stack regions tends to keep active stack memory in CPU caches, thus speeding up access. Each thread in a process gets its own stack.

- *Heap segment.* Above the data segment is the heap region which contains the dynamically allocated variables. As the amount of memory space used by a program can vary while the program is running, the position of the top of the heap also can change during execution time. Most languages provide heap management functions to programs. Satisfying memory requests is thus a joint cooperation between the runtime language and the kernel. In C, the interface to the heap allocation is `malloc()` and friends. If there is enough space in the heap to satisfy a memory request, it can be handled by the language runtime without kernel involvement. Otherwise, the heap is enlarged via the `brk()` system call to make room for the requested block.

- *Memory mapping segment.* In this region the kernel maps the content of files directly to memory. Any application can ask for such a mapping via the Linux `mmap()` system call. Memory mapping is a convenient and high-performance way to do file I/O, so it is used for loading dynamic libraries, for example the `libc*.so`. In Linux, if one requests a large block of memory via `malloc()`, the C library will create such an anonymous mapping instead of using the heap memory. 'Large' means larger than MMAP_THRESHOLD bytes, 128 kB by default and adjustable via `mallopt()`.

Some addresses of the above segments are fixed and specified at compile time, including the ones of text, BSS and data segments. Furthermore, they can be extracted in program, for example by using the associated variables such as the `__data_start`, `_edata`, `__bss_start`, which contain address of the beginning and the end of the data segment, and the beginning of the BSS segment respectively. On Fig. 3.2, these variables are presented on the left side. Some others are changed at execution time and there are not associated system variables. However, there are some other methods to find these addresses, for example using the `sbrk` system

call with a parameter equal 0 that returns the current position of the top of the heap. In kernel mode, using the `mm_struct` that contains the fields that refer to addresses of most the segments of processes, except addresses of the end of the BSS segment, and the end of the stack. On Fig. 3.2, these fields are presented on the right side. Another method in kernel mode to find all allocated memory regions of a process at the execution time which uses the `mmap` field of `mm_struct` that refers to the first element of a `vm_area_struct` list. Each member of this list is a structure containing the fields that specify both start and end addresses of the associated region and some other properties.

Another important element is Linux randomizing stack, memory mapping segment and heap by adding an offset. This aims at preventing remote attacks and does not cause any problem in case of complete checkpoints. However, in case of incremental checkpoints, the problem is more complex and requires a special processing to resume the process or to inject a checkpoint into a process memory as the memory spaces of the process are different between the different executions. A possible solution consists in modifying the addresses of memory regions in checkpoints when injecting them into the process memory. On one hand, this is slightly complex and consumes execution time. On the other hand, this cause the requirement to update value of pointers while modifying associated dynamically allocated variables. This requirement in turn, is very complex to solve dues to difficulties to specify all pointers of the program. As a result, we have decided to set all the randomize regions to 0, to avoid this problem. The following command must be executed in the terminal before processes are started.

```
sysctl -w kernel.randomize_va_space=0
```

The "snapshot" of process memory contains all these above regions. However, since some of them are read-only and/or do not change in execution time, they can be built from the executable code of the program and thus, do not need to be saved, and this reduces the size of checkpoints. Furthermore, this prevents system fatal errors due to write operations in read only regions of text or memory mapping segments.

3.1.3 Paging in hardware

The paging unit translates linear addresses into physical ones. One key task in the unit is to check the requested access type against the access rights of the linear address. If the memory access is not valid, it generates a Page Fault exception. For the sake of efficiency, linear addresses are grouped in fixed-length intervals called pages; contiguous linear addresses within a page are mapped into contiguous physical addresses. This way, the kernel can identify the physical address and the associated access rights of a page instead of those of all the linear addresses includes. The page size depends on the architecture. On the 32-bit x86 architecture, it is 4kB. Table 3.1 shows the page size for some 64-bit architectures.

Platform name	Page size	Number of paging levels
alpha	8 kB[1]	3
ia64	4 kB[1]	3
ppc64	4 kB	3
sh64	4 kB	3
x86_64	4 kB	4

Table 3.1: Page sizes and paging levels in some 64-bits architectures.

The paging unit considers of all RAM as partitioned into fixed-length page frames (sometimes referred to as physical pages). Each page frame contains a page. A page frame is a component of the main memory, and hence it is a storage area. It is important to distinguish a page from a page frame; the former is just a block of data, which may be stored in any page frame or on disk, while the latter refers to a physical region in main memory.

The data structures that map linear to physical addresses are called page tables; they are stored in the main memory and must be properly initialized by the kernel before enabling the paging unit.

3.1.4 Paging in Linux

Linux adopts a common paging model that fits both 32-bit and 64-bit architectures. For 32-bit architectures, two paging levels are enough, while 64-bit architectures require a higher number of paging levels. Up to version 2.6.10, the Linux paging model consisted in three paging levels. Starting with version 2.6.11, a four-level paging model has been adopted[1]. The four types of page tables are:

- Page Global Directory (PGD)
- Page Upper Directory (PUD)
- Page Middle Directory (PMD)
- Page Table (PT)

The Page Global Directory includes the address of several Page Upper Directories, which in turn include the address of several Page Middle Directories, which in turn include the address of several Page Tables. Each Page Table entry points to a page frame. Thus, the linear address can be split into up to five parts. Figure 3.3 presents this division in the Linux paging model, on which the bit numbers are not shown, because the size of each part depends on the computer architecture.

[1]This architecture supports different page sizes; This is the typical value adopted by Linux.

[1]This change has been made to fully support the linear address bit splitting used by the x86_64 platform (see 3.1)

For 32-bit architectures with Physical Address Extension enabled, three paging levels are used. The Linux Page Global Directory corresponds to the 80x86 Page Directory Pointer Table, the Page Upper Directory is removed, the Page Middle Directory corresponds to the 80x86 Page Directory, and the Linux Page Table corresponds to the 80x86 Page Table. Finally, for 64-bit architectures, three or four levels of paging are used depending on the linear address bit splitting performed by the hardware (see Tab. 3.1).

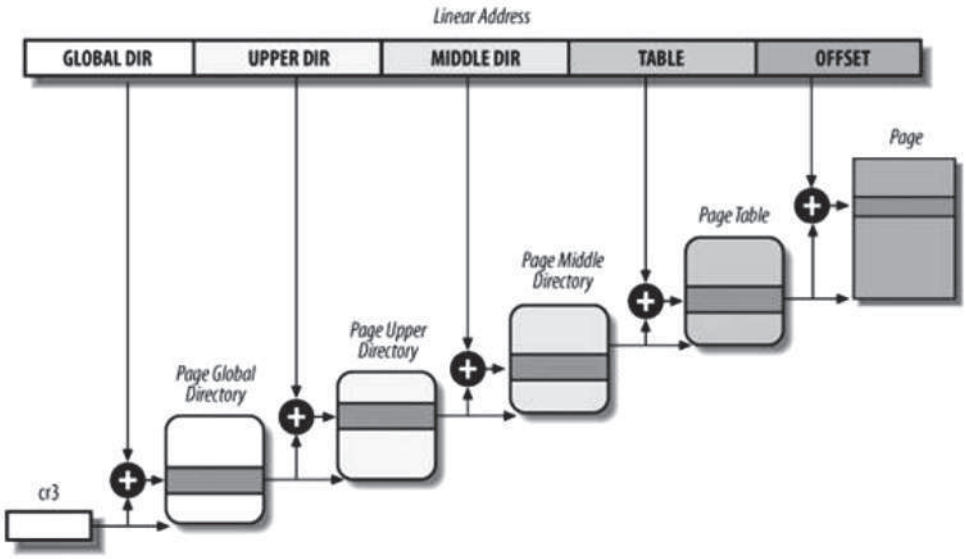


Figure 3.3: Linux paging model

3.1.5 Page Table Handling

Linux supports a set of data structures, functions and macros to handle Page Tables.

`pte_t, pmd_t, pud_t,` and `pgd_t` data types describe the format of Page Table, Page Middle Directory, Page Upper Directory, and Page Global Directory entries respectively. Five type-conversion macros `__pte`, `__pmd`, `__pud`, `__pgd` and `__pgprot`) cast an unsigned integer into the required data type. Five other type-conversion macros `pte_val`, `pmd_val`, `pud_val`, `pgd_val` and `pgprot_val` perform the reverse casting from one of the four previously mentioned specialized data types into an unsigned integer.

The kernel also provides several macros and functions to read or modify page table entries:

- Functions `pte_none`, `pmd_none`, `pud_none` and `pgd_none` check the existence of entries. They yield to value 1 if the corresponding entry has value 0; otherwise, they yield to value 0.

- Functions `pte_clear`, `pmd_clear`, `pud_clear` and `pgd_clear` clear an entry of the corresponding page table, thus forbidding a process to use the linear ad-

dresses mapped by the page table entry. The `ptep_get_and_clear()` function clears a Page Table entry and returns the previous value.

- Functions `set_pte`, `set_pmd`, `set_pud` and `set_pgd` write the given value into a page table entry; `set_pte_atomic` is similar to `set_pte`. However, when the Physical Address Extension Paging Mechanism (PAE) is enabled, it also ensures that the 64-bit value is written atomically.

The kernel also provides some macros to check the validity of an element in the Directories. The `pmd_bad` macro is used by functions to check Page Middle Directory entries passed as input parameters. It yields to value 1 if the entry points to a bad Page Table, i.e. if at least one of the following conditions applies:

- The page is not in main memory (`Present` flag cleared).
- The page allows only `Read` access (`Read/Write` flag cleared).
- Either `Accessed` or `Dirty` is cleared (Linux always forces these flags to be set for every existing Page Table).

The `pud_bad` and `pgd_bad` macros always yield 0. No `pte_bad` macro is defined, as it is legal for a Page Table entry to refer to a page that is not present in main memory, not writable, or not accessible at all. The `pte_present` macro yields to value 1 if either the `Present` flag or the `Page Size` flag of a Page Table entry is equal to 1, value 0 otherwise. Remind that the `Page Size` flag in Page Table entries has no meaning for the paging unit of the microprocessor; however, the kernel marks `Present` equal to 0 and `Page Size` equal to 1 for pages present in main memory but without read, write, or execute privileges. In this way, any access to such pages triggers a `Page Fault` exception as `Present` is cleared, and the kernel can detect that the fault is not due to a missing page by checking the value of `Page Size`.

The `pmd_present` macro yields to value 1 if the `Present` flag of the corresponding entry is equal to 1, i.e. if the corresponding page or Page Table is loaded in main memory. `pud_present` and `pgd_present` macros always yield to value 1.

To specify access properties of pages, each element in a Page Table Directory is adopted a set of flags. To read and set their values, the kernel provides a set of functions. Functions `pte_user()`, `pte_write()`, `pte_young()` and `pte_dirty()` read the value of flags `User/Supervisor`, `Read/Write`, `Young` and `Dirty` respectively. Functions `pte_wrprotect()`, `pte_rdprotect()`, `pte_mkdirty()` and `pte_mkold()` clear flags `Read/Write`, `User/Supervisor`, `Dirty` and `Accessed` respectively.

Another set of macros serve to find elements in Table Directories. The `pgd_offset(mm, addr)` function receives as parameters the address of a memory descriptor and a linear address `addr`. The macro returns the linear address of the entry in the Page Global Directory that matches address `addr`. `pud_offset(pgd, addr)` receives as parameters a pointer to a Page Global Directory entry and a linear address `addr`. It returns the linear address of the entry in the Page Upper Directory that matches `addr`. In a two- or three-level paging system, this macro returns `pgd`, the address of

the Page Global Directory entry. Macro `pmd_offset(pud, addr)` receives a pointer to a Page Upper Directory entry and a linear address `addr` as parameters. It returns the address of the entry in the Page Middle Directory that matches `addr`. In a two-level paging system, it returns pud, the address of the Page Global Directory entry. Finally, the `pte_offset_map(pmd, addr)` macro receives a pointer `pmd` to a Page Middle Directory entry and a linear address `addr` as parameters. It returns the linear address of the entry in the Page Table that corresponds to linear address `addr`.

There are many other functions and macros provided by kernel for the Table Handling. A complete list can be referred in the page 61–65 of [77].

3.1.6 Example: how to set a page to the writable status

All the above information are used in our checkpointer to write the operations managing the memory space of the checkpointed process. For example, to set the `Read/Write` flag of the page containing the address `addr` in the memory space of the process having process ID `pid`, the main steps are:

- *search the process memory object* (`mm_struct` object). This object is the initial clue to access most of the process memory managing objects. From the process ID, it is possible to get the associated `task_struct` object and the `mm` field of this object points to the effective memory object.

- *check the validity of the address.* Theoretically, the virtual memory space of a process consists of 4GB, from 0x00000000 to 0xFFFFFFFF as a global view and from the `start_code` address to 0xC0000000 for the user space. However, the kernel only allocates virtual memory areas (`vma`) for the regions that really used by the process. As a result, a verification of the validity of an address is equivalent to checking whether it is in a `vma`.

- *find the associated memory page and set its status to writable.* Sequentially find the associated objects in the Page Global Directory, the Page Middle Directory and the Page Table. There are three important notes in this steps. The first one consists in locking/unlocking the page table each time properties of page table elements are modified. This prevent conflicts from the other accesses in multi-processors/multi-cores systems. The second one consists in calling the `handle_mm_fault()` kernel function to solve the case where a memory region is valid but not located in the main memory. The last note consists in updating the Page Table from the cache, after it has been modified, to ensure that updates are seen by the next operation. The `pte_mkwrite( )` function is used to set the page to the writable status.

Below is an extraction from our checkpointer. This is a function which gets a process ID and an address as parameters and then sets the page including this address to the writable status.

```
int clear_write_protect(unsigned int pid, unsigned long addr){
    struct task_struct *t, *task = NULL;
    for_each_process(t) //find the task associated with process
         if (t->pid == pid) task = t; //search the task object
    if(task == NULL){ printk("pid invalid"); return 1; }
    mm = task->mm; //the memory object
    vma = find_vma(mm, addr); //the virtual memory area
    if(!vma) { printk("addr invalid"); return 2; } //this address is not allocated
    spin_lock(&mm->page_table_lock); //lock the Page Table
    pgd = pgd_offset(mm, addr); //search the PGD
    if (pgd_none(*pgd)) { ret = 3; goto out; }
    pmd = pmd_offset((pud_t *)pgd, addr); //search the PMD
    if (pmd_none(*pmd)){ //In case of setting a new value for the brk
              //(end of heap), the PMD may not be exist.
              //So, let the kernel try to process first.
         spin_unlock(&mm->page_table_lock); //unlock the PT before letting
              //the kernel to process
         ret = my_handle_mm_fault(mm, vma, addr, 0); //this will call
              //the handle_mm_fault( ) kernel function.
         spin_lock(&mm->page_table_lock);
    }
    if(pmd_none(*pmd)) { ret = 4; goto out; }
    pte = pte_offset_map(pmd, addr); //search the PTE
    if (pte_present(*pte)){ //page is in the main memory
         *pte = pte_mkwrite(*pte); //set its status to writable
         flush_tlb_page(vma, addr); //flush the cache to update the PT
    }else{ //page is not in the main memory, let the kernel to process first
         vma->vm_flags |= VM_WRITE;
         pte_unmap(pte);
         spin_unlock(&mm->page_table_lock);
         ret = my_handle_mm_fault(mm, vma, addr, 0);
         spin_lock(&mm->page_table_lock);
         vma->vm_flags &= ~VM_WRITE;
         if(ret == 0){
              pte = pte_offset_map(pmd, addr);
              *pte = pte_mkwrite(*pte);
              flush_tlb_page(vma, addr);
         }else{
              pte_unmap(pte);
              goto out;
         }
    }
    pte_unmap(pte);
    spin_unlock(&mm->page_table_lock);
    return ret;

out:
    printk("error:  %d\n", ret);
    spin_unlock(&mm->page_table_lock);
    return ret;
}
```

3.2 Discontinuous Incremental Checkpointing

Specially designed for CAPE, Discontinuous Incremental Checkpointing (DICKPT) technique is based on Incremental Checkpointing technique with the additional ability of executing the checkpointing in discrete sections of programs.

3.2.1 Mechanism for memory modification detection

As in incremental checkpointing, the first challenge of DICKPT is to recognize the updated regions of the process space while the process is running. At present, there are three mechanisms to solve this problem.

1. **Use the `dirty` bit**. When a page is written, the hardware sets the `dirty` bit in the corresponding page. Thus, when a checkpoint is taken, all the Page Table entries are examined to determine whether the page has been changed since the previous checkpoint or since the beginning of the program. Then, all the writable pages are marked as clean (non-dirty). There is an important problem inherent to this method: all `dirty` bits are cleared when the cache containing updated pages is written back to the main memory. A solution presented in [79] consists in mirroring the original `dirty` bit into one of the unused entry bits in the Page Table entry. Low-level functions used by the kernel to access this bit are properly updated.

2. **Use the `write` bit**. At the beginning of the program or after a checkpoint, all the writable pages are set to read-only. When a page is written for the first time, a page fault exception is generated and the page fault exception handler saves the address of the fault page in a list. Then, the page is set to writable status. When a checkpoint is taken, all pages having addresses in the saved list are read from the current memory space and saved to the checkpoint.

3. **Use the `write` bit and Save in a buffer**. This mechanism is the same as the previous one except that the page is also copied into a buffer before being set to writable.

Among the above mechanisms, the first one has the advantage of having no effect on the process performance. Only the third one can provide a detection of memory changed at variable-granularity. This ability is given by the comparison between the pages in the buffer and the current pages in process memory at the time the checkpoint is taken. As a response to the requirement of providing the list of memory modifications in slave nodes at variable-granularity, DICKPT uses this last mechanism to detect all memory modifications.

3.2.2 The additional directives

An incremental checkpointer can be transparently implemented i.e. no add extra information in the code of checkpointed program. At the beginning of the program,

the checkpointer set all writable pages of checkpointed process to the read-only status and then waits for page-fault signals. Checkpoints can be taken periodically or at any time after a signal is sent by the user.

It is different in the case of DICKPT. To exactly specify the checkpointed sections in a program, additional information have to be inserted into the original source code. This can be performed manually by programmers or automatically be another tool such as a compiler in the case of CAPE. Three following directives have been added.

- `pragma dickpt start` clears the buffer and then starts or resumes checkpointing. Any modifications occurring on the process after a call to `start` is reported in the buffer. A call to `start` while the checkpointer is active results in clearing the content of the buffer which is definitively lost.
- `pragma dickpt stop` stops checkpointing, i.e. any modifications that occurs on the process after a call to the `stop` is not reported in the buffer. A call to `stop` while the checkpointer is not active is just discarded.
- `pragma dickpt save` *`filename`* saves the content of the buffer in the file provided as a parameter. Several calls to `save` may occur inside a `start/stop` pair of directives. If the *`filename`* is the same as in the previous call, the current checkpoint is merged with the previous one. Otherwise, a new file is created.

Assume that a program consists of segments (A, B, C, D) in which, only B and D need to be checkpointed and two checkpoints are taken in B and one in D. Fig. 3.4 presents the prototype of the changed program, i.e. the directives that have been inserted to verify the above requirements.

```
A
# pragma dickpt start
B1
# pragma dickpt save <filename1>
B2
# pragma dickpt save <filename1>
# pragma dickpt stop
C
# pragma dickpt start
D
# pragma dickpt save <filename2>
# pragma dickpt stop
```

Figure 3.4: Example of pseudo-code for discontinuous incremental checkpoints.

According to the discontinuous feature, checkpoints of this type can be merged only when they have been taken in the same region, surrounded by a pair of

`start/stop` pragmas. Furthermore, they cannot be used to restart the execution of the program. However, they can be injected into the state of programs. In fact, resuming a program is more complex and contains the phases of injecting checkpoints interwoven with the phases of re-running the program. As a result, resuming is correct only when the conditions that consist in the phases resuming the program are the same as the conditions when the program was originally running.

3.2.3 Checkpointer level

As presented in 2.3.2, checkpointer can be developed in the user-level or in the kernel-level.

At the user-level, when the checkpointing is included in application programs, it makes programs lost their independence. When using an independent checkpointer, there are two important limits. The first one is the restriction of the access to the information system of checkpointed programs. The second one is the difficulties to built a high-speed data exchange mechanism between the checkpointer and checkpointed programs. For example, most of incremental checkpointers (i.e. those using the `write` bit) use the `ptrace` mechanism to catch signals from checkpointed applications. This mechanism limits the use of functions to read/write data from/to checkpointed programs as only one byte each call. Thus, if the checkpointer reads a page from the checkpointed program, a large number of system calls have to be used and this strongly reduces the global performance.

From the above analysis, we develop our own checkpointer as a two layer one, i.e. one part in the kernel space and another one in the user space. The first one is a collection of functions to manage and to read/write from/to the memory of checkpointed programs. While being developed at this level, there is no limit to access the information system and data of checkpointed programs. The part of the checkpointer at user space is a monitor that tracks signals from checkpointed programs and call the functions in the kernel space to perform the checkpointing tasks. In this way, all checkpoint tasks are executed by the monitor. In the checkpointed program, only the directives that might be implemented by the signals, are inserted.

Figure 3.5 shows the two layers of the DICKPT checkpointer with the main functions that are presented below.

- `lock_process_image()` sets all pages of the application process to the read-only status.
- `unlock_process_image()` sets all pages of the application process to their initial status.
- `unlock_a_page(` *`addr`* `)` sets the page including the address *`addr`* to its initial status.
- `read_range(` *`addr, length, dst`* `)` reads from the application process space *`length`* bytes, starting at address *`addr`*. Result is set to the location pointed by the *`dst`* pointer.

- `write_range( addr, length, values )` writes to the application process space *length* bytes, start at address *addr*. Values to write are pointed by the *values* pointer.

- `start_signal_proc()` process signals associated with the `dickpt start` pragma. It initializes the checkpoint buffer and calls the `lock_process_image()` function to set all pages of application process to the read-only status.

- `stop_signal_proc()` process signals associated with the `dickpt stop` pragma. It clears the checkpoint buffer and calls the `unlock_process_image()` function to set all pages of application process to their initial status.

- `save_signal_proc()` process signals associated with the `dickpt save` pragma. When called, it creates an incremental checkpoint by comparing the pages in the checkpoint buffer and the pages read from the application process space. Then, it clears the buffer and calls the `lock_process_image()` function to prepare the potential next checkpoint.

- `SIGSEGV_proc()` process SIGSEGV signals generated by the application program. It calls the `read_range()` function to read the associated page of application process then adds this page to the checkpoint buffer. Finally, it calls the `unlock_a_page()` function to set the page to the writable status.

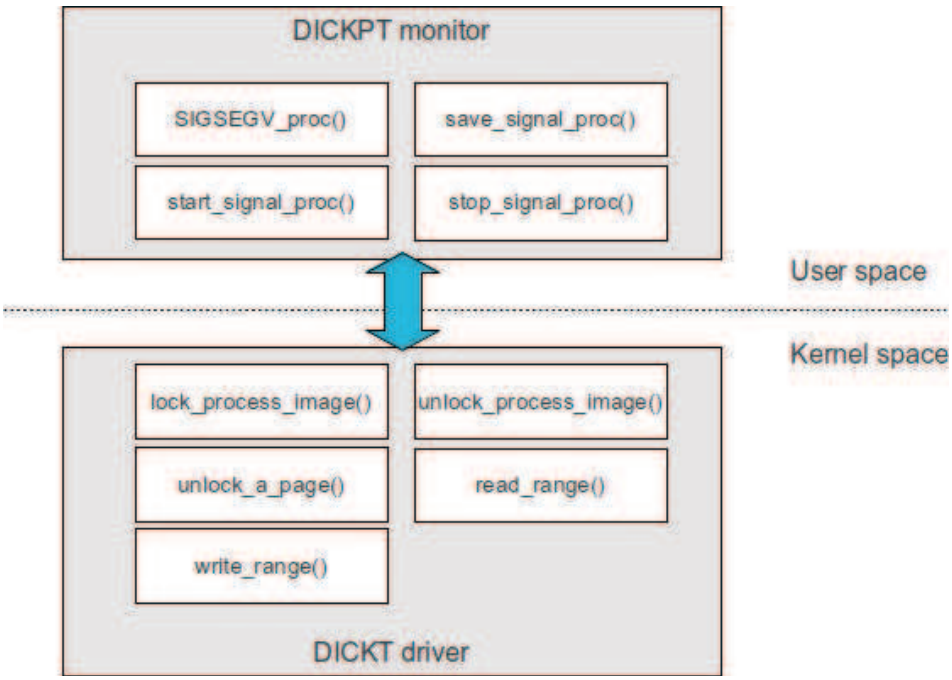


Figure 3.5: Preliminary design of DICKPT checkpointer.

3.3 Detailed design

In order to perform checkpoint operations, each process that may be checkpointed has to be associated a monitor. Typically, this monitor is in charge of starting the process which checkpoints will be computed, catching checkpointing signals, generating the checkpoint files or/and restoring the process state from checkpoint files and waiting for the termination of the process. However, this is not mandatory as a monitor may be attached to any already running processes.

3.3.1 Execution mechanism in checkpointing cases

The basic principle of DICKPT in the case of checkpointing is showed in Fig. 3.6. After starting the checkpointed program, the monitor waits for signals to be caught. There are three main cases: three of them are associated with the three checkpointing directives and the other two are associated with both SIGSEGV and exit signals.

1. **After the `start` signal**, the monitored process is suspended to set access rights to read-only to all its pages. Then, the process is ready to be checkpointed. From now, any SIGSEGV signal delivered to the monitored process is caught by the monitor.

2. **After the `SIGSEGV` signal**, there are two possibilities:

   - The SIGSEGV signal is delivered because the program wants to write in a read-only memory page which original access rights included the write capability. This means that this page is valid and it is the first time it is accessed since the last `start` signal had been generated. As a result, the content of the page is read by the monitor using the `read_range()` function of the DICKPT driver and stored in the monitor for future reference, then the writable property is added to the page and the monitored process is asked to resume its execution. No signal is delivered to the monitored process for which this operation is therefore completely transparent. In the case the page is valid, but does not exist in the current memory space, the checkpointer firstly call the `mm_handle_fault()` kernel function to process it.
   - In any other cases, this means that the page does not exist in the virtual address space of the monitored program or the original access rights of this page do not include the write capability. In other words, the SIGSEGV signal is not delivered because the monitor changed the access rights of this page at the beginning of the execution of the monitored process, but for any other reasons including a bug inside the program. In this case, nothing is performed by the monitor and the signal is left to the monitored process. If any mechanism was set inside the program to catch this signal, the associated handler is executed.

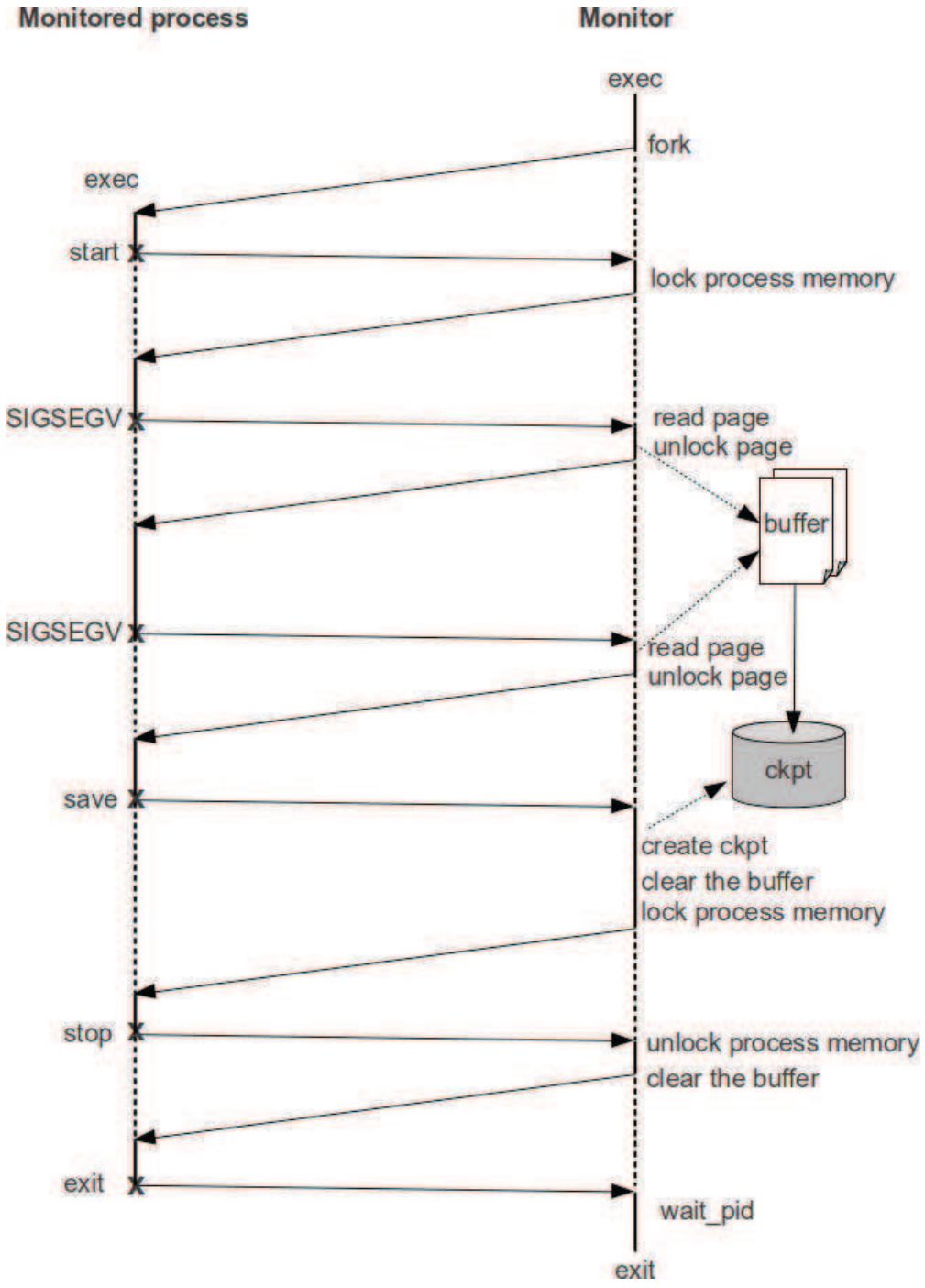


Figure 3.6: Principle of DICKPT in cases of checkpointing.

3. **The `save` signal**. When the monitor receives a request to generate a checkpoint of the monitored process, it 1) reads the content of all pages that have been modified since the last checkpoint or since the beginning of the program using the `read_range()` function of the DICKPT driver, 2) determines the list of memory locations that have been modified in each modified pages to generate the new checkpoint, and 3) removes the write permission to pages in order to prepare the potential incremental checkpoint.

4. **After the `stop` signal**, the monitored process is suspended to set access rights to writable status on all its pages. Then, the monitor is not responsible anymore for processing SIGSEGV signals from the monitored process.

5. **The `exit` signal**. When the monitored process finishes its execution, i.e. after the `exit` system call has been called, whenever it has been explicitly called or not, and the error code of the child process has been returned, the monitor returns from the `waitpid` system call and is therefore notified that the monitored process terminated. Then, the monitor can also terminate its execution.

3.3.2 Execution mechanism in recovering cases

In recovering cases, the monitor is started with a list of checkpoints. To ease the processing, all checkpoints that were taken in a single `start/stop` pair of directives were merged into an unique file.

After starting the checkpointed program, the monitor waits for signals from it. The three signals associated with the three DICKPT directives are processed as the following.

1. **After the `start` signal**, the monitored process is suspended and the monitor search the next checkpoint in the list. If it is not found, the monitor notifies an error and stop the execution. In the other case, the checkpoint is injected into the monitored process space. If it is the last checkpoint, the monitor sets access rights of all checkpointed process pages to read-only and changes to the checkpointing mode.

2. **The `stop` signal**. In recovery mode, this signal is just ignored.

3. **The `save` signal**. This requirement is valid only when in checkpointing mode, and in this case the monitor notifies an error and stops the execution.

   Note that in recovering mode, the monitor is not responsible for processing the SIGSEGV signals since it does not lock the memory of the monitored process. Table 3.2 summarizes the processing of the directives in both checkpointing and recovering modes.

Table 3.2: Processing the directives of the DICKPT on the monitor side.

Directive	Checkpointing mode	Recovering mode
start	set all pages to the read-only status	find the next checkpoint: - if found: + inject the checkpoint to the monitored process + if it is the last checkpoint: . set all pages to the read-only status . change to checkpointing mode - else: + notifies error + stop process
stop	set all pages back to their original status clear the buffer	ignored

save $file$	save memory locations that have been modified up to the current checkpoint
	if $file$ contains previous checkpoint:
	+ create $file$ and append current checkpoint
	else:
	+ save current checkpoint to $file$
	set all pages to the read-only status
		notifies error
		stop process

3.3.3 Implementation of the directives

In DICKPT checkpointer, directives have been implemented using signals. The signaling mechanism provides a quick asynchronous communication mechanism between processes. Linux defines a set of signals. Most of them are reserved for system tasks. Only two signals, SIGUSR1 and SIGUSR2, are free for user programs. Since DICKPT requires only three directives, it can be implemented using these two signals and another one that is not often used, e.g. SIGTSTP, i.e. one signal for each directive. However, this leads to two drawbacks. The first one is the loss of independence. These signals can be already used by the user program and in this case, there will be a collision. The second one is the need of additional cases, for example for the asynchronous data exchange when using the checkpointer in CAPE. To solve these problems, we used only the SIGTRAP signal which is used to set break points in debuggers. Typically, this signal is not used in user program at execution time, so there is not collision when using it. To branch the signal to many cases, an additional arguments is sent to the checkpointer. In the current version, we used the `dx` register as the argument.

The inline assembler code below implements the DICKPT directive to send the SIGTRAP signal with option 1 to the monitor process.

```
asm("push %edx; mov $1, %edx; int $3; pop %edx");
```

The command assigns the `dx` register the value 1, after saving its original value. Then, interrupt number 3 is called. This interrupt delivers the SIGTRAP signal to the monitor process and stops the monitored process execution. After the SIGTRAP signal has been processed, the original value is set to the `dx` register, and the process resumes its execution.

On the monitor side, when a SIGTRAP signal is received, it reads the `dx` register of the monitored process (using the `ptrace` system call) and performs the associate checkpointing task. After finishing all the work, a signal is sent to the monitored process to ask it to resume its execution.

3.4 Performance evaluation

To compare the performance of our new approach and the one of the normal incremental checkpointing technique, we have measured their impacts on a program computing the successive elements of a Markov Chain, see Fig. 3.7. Two cases were tested. The first one associated with the normal incremental checkpointing technique includes a directive to begin the checkpointing at location 0 (line 8) and takes a checkpoint at location 1 (line 21). The second one associated with DICKPT technique avoids the checkpoint at location 1 and begins checkpointing at location 1. For both cases, one hundred state vectors are computed at location 3 (line 28) and one checkpoint is generated after each computation. The testbed is composed of an Intel Core2 Duo E8400 running at 3 GHz with 3 GB RAM and operated by Ubuntu

9.10 based on Linux kernel 2.6.31-21-generic. Table 3.3 presents the performance evaluation for four vector sizes (N equals to 3320, 6640, 9960 and 13280 elements respectively). For each vector size, performance are measured 30 times (mean values are provided in the table) and a confidence interval of at least 99% has always been achieved for the measures. In order to avoid the disk effect pollution on measurements, all data (the virtual address space of processes, checkpoints, etc.) are resident in RAM.

```
1   # include <stdio.h>
2   # include <stdlib.h>
3   # define LOOP 100
4   # define N 9960
5   # define RAND 10000
6   float M [ N ] [ N ], V [ 2 ] [ N ] ;
7   int main ( int argc, char * argv [ ] ) {
8         //location 0
9         float sum, val ;
10        int i, j, k, l ;
11        for ( i = 0 ; i < N ; i ++ ) {
12              for ( sum = j = 0 ; j < N ; j ++, sum += val )
13                    M [ i ] [ j ] = val = rand ( ) % RAND ;
14              for ( j = 0 ; j < N ; j ++ )
15                    M [ i ] [ j ] /= sum ;
16        }
17        for ( sum = i = 0 ; i < N ; i ++, sum += val )
18              V [ 0 ] [ i ] = val = rand ( ) % RAND ;
19        for ( i = 0 ; i < N ; i ++ )
20              V [ 0 ] [ i ] /= sum ;
21        //location 1
22        for ( k = l = 0 ; l < LOOP ; l ++, k = 1 - k ) {
23              for ( i = 0 ; i < N ; i ++ ) {
24                    V [ 1 - k ] [ i ] = 0.  ;
25                    for ( j = 0 ; j < N ; j ++ )
26                        V [ 1 - k ] [ i ] += ( V [ k ] [ j ] * M [ j ] [ i ] ) ;
27              }
28              //location 2
29        }
30        return 0 ;
31  }
```

Figure 3.7: Program computing the successive elements of a Markov Chain.

The first section of Table 3.3 presents the size of checkpoints at location 1 (i.e. just after initialization as the case of normal incremental checkpointer) and at location 2 (i.e. just after the computation of a new vector). This difference is the size of the transition matrix which is initialized at the beginning of the program. These

data show how much disk space can be saved while abandoning the checkpoint at location 1 and, in the same way, how faster the checkpoint can be transfered over the network if necessary.

The second section of the table shows the time required to run the program without saving checkpoints, while saving all checkpoints and while saving location 2 checkpoints only. It highlights the fact that the overhead involved by the generation of location 2 checkpoints is very light (between 1.5% and 3.3% for 100 checkpoints, i.e. between 0.01% and 0.03% per checkpoint) compared to the overhead involved by the generation of both location 1 and location 2 checkpoints (the total execution time is multiplied by 8.5).

The third section of the table provides the execution time to run the process restarting from loop iteration number 50. Two cases are envisaged: the first one uses all checkpoints, i.e. the program is restarted, suspended at the beginning of function main, all checkpoints are injected in the process and the execution resumes at loop iteration 50; the second one uses location 2 checkpoints only, i.e. the program is restarted, suspended after the initialization step, all location 2 checkpoints are injected in the process and the execution resumes at loop iteration 50.

Performance measurements show that avoiding location 1 checkpoints is always beneficial.

Table 3.3: Performance evaluation of DICKPT.

	Matrix size			
	3320	6640	9960	13280
Checkpoint size (in MB)				
... at location 1	42.192	168.741	379.648	674.912
... at location 2	0.013	0.026	0.038	0.051
Total execution time (in seconds)				
... without generating checkpoints	11.34	41.30	108.27	168.69
... generating all checkpoints	13.10	58.14	393.60	1433.72
... generating location 2 checkpoints only	11.71	42.16	109.96	171.70
Execution time restarting after loop iteration #50 (in seconds)				
... using location 1 and location 2 checkpoints	6.41	23.14	59.56	93.91
... using location 2 checkpoints only	6.09	21.84	56.64	88.71

3.4.1 Advantages and drawbacks

While comparing with the normal incremental checkpointing technique, our new approach has the following strengths and weaknesses:

- In terms of performance, this solution increases the execution speed of the program in both periods of checkpointing and recovering; it also strongly decrease the size of checkpoints.

- Flexibility: DICKPT allows to select the segments to be checkpointed in programs. The case of normal (not discontinuous) incremental checkpointing is obtained by setting a `pragma dickpt start` and a `pragma dickpt stop` as the first and the last instruction respectively in the checkpointed program.

- It provides special abilities to CAPE, like the ability to exactly specify discrete sections that should be checkpointed and the ability to directly inject a checkpoint into process space of application programs. This leads to the ability to directly extract execution results on slave nodes, to integrate results into the memory space of the master thread without resuming it. All of them help to increase the performance of checkpointing.

- Change of the source code: in the role of a checkpointer, it is the most important drawback. Users have to insert directives to indicate the regions that may be checkpointed. However, when used in CAPE, this insertion is done by the compiler. As a result, this drawback has no impact on CAPE's users.

- Fragmentation of checkpoints: checkpoints which are not taken in a single block (surrounded by a pair of `# pragma dickpt start` and `# pragma dickpt stop`) can not be merged into a unique checkpoint. So, many files are needed to contain the checkpoints of different checkpointing blocks. This drawback is important when checkpoints reference the same memory area.

3.5 Checkpoint structure optimization

The structure of a complete checkpoint is usually quite straightforward. After some very specific data like the content of registers and the size of the memory, the rest of a complete checkpoint is usually composed of the content of all memory pages the one after the other one.

In the case of an incremental checkpoint, several cases have to be envisaged. All solutions are storing the content of registers. However, regarding memory updates, the best solution really depends upon the granularity of data, which ranges from one byte to one page with the most interesting case at the word level.

3.5.1 Memory granularity

There are two main drawbacks when the granularity is the page. The first one is that a complete page must be saved even though a single byte in the page has been modified, which is not memory efficient. Considering the size of today disks, this may not be a problem unless a very large number of checkpoints have to be saved. The problem may have a more important impact if for example these checkpoints

have to be sent over the network, especially with a limited bandwidth. The second main drawback is that there is no information on which bytes in the page have been modified effectively. The latter drawback definitively forbids any merge operation of successive incremental checkpoints.

Setting the granularity of the checkpoint to a single byte solves the memory inefficiency problem of the page granularity. However, it leads to other subtle problems, like for example the reference to memory locations that do not exist in the virtual address space of the process. Let $< a, b, c, d >$ be four bytes stored at a memory location and representing a pointer in memory. After a first checkpoint, this memory location may contain $< a, b', c, d >$. After a second checkpoint, the same memory location may contain $< a, b, c', d >$. If, for any reasons, it is required to merge the two checkpoints (and this is typically the case with CAPE), the result might become $< a, b', c', d >$ which may not be part of the virtual address space of the process.

Setting the granularity of the checkpoint to a word (i.e. four bytes) is the best compromise as it solves the problem of memory space efficiency and does not introduce any pointer problem as described above. This solution is not the perfect solution. However, problems involved by setting the granularity of the checkpoint to a word has no significant impact on the execution of the program.

Finally, one can note that setting the granularity of the checkpoint to the entire virtual address space turns an incremental checkpointer into a complete checkpointer.

3.5.2 Incremental checkpoint content

Apart from the specific values also stored in complete checkpoints, an incremental checkpoint should be composed of the list of memory locations that have been modified since the beginning of the execution of the program, or since the previous checkpoint, and the last value for each of these specific memory locations. The simplest structure to store such a list is to save the one after the other one both the addresses and their associated value. However, since the spatial locality of data in most programs implies that a modification at a memory location increases the probability for adjacent memory locations to be modified, this way of storing data is not necessarily efficient.

Thus, in order to take advantage of the spatial locality of updates and therefore reduce the size of checkpoints, several alternative methods for storing memory updates have been identified:

- *Single data.* This case occurs when a single memory location has been updated. In this case, the only information to store are the basic address of the memory location and the content at the memory location. Data to store all information into the checkpoint are:

$$< \text{addr}, \text{value} >$$

- *Several successive data.* This case occurs when more than one consecutive memory locations have been updated. For example, this is encountered when the content of an array has been modified. The best way to store all the information in this case is:

$$< \text{addr}, \text{size}, [\text{value...}] >$$

- *Many data.* This occurs when lots of non-successive memory locations have been updated on a single page. In this case, instead of storing a large number of Single data and Several successive data elements, it is more efficient to store the address of the page, the list of memory locations on the page that have been modified and for each modified memory location the associated value. The efficiency of this solution resides in the mapping, i.e. the list of memory locations on the page. As this is a binary information for each data in the page, it can be represented using a single bit per memory location. For example, for a 4-kB page, the size of the map is 1024 bits (or 128 bytes) with a granularity set a word.

$$< \text{addr}, \text{map}, [\text{value...}] >$$

- *Entire page.* This occurs when all memory locations on a memory page have been modified. This case is quite common when a new page is added to the virtual address space of a process. The best way to store the complete content of a page is:

$$< \text{addr}, [\text{value...}] >$$

  No size need to be provided in this case as it is implicit.

Table 3.4 compares the amount of memory needed to store updated data for all cases presented above. The size of a memory page is assumed to be 4 kB. Let a *chunk* be a set of contiguous memory locations that have been updated. Let c be the number of chunks in a memory page, let s_i be the number of elements in chunk i and let u be the number of updates in the memory page. By definition, $\sum_{i=1}^{c} s_i = u$.

Figure 3.8 shows a comparison of the amount of memory needed to store all updates in a 4-kB page as a function of the number of updated memory locations in the page. SD, MD and EP only depend upon the number of updated memory locations while SSD also depends on the distribution of the updated memory locations. As a result, Fig. 3.8 shows both the best case (SSD_{min}) that is when all updated memory locations are in a single chunk, and the worst case (SSD_{max}) that is the case when updated memory locations are distributed in the configuration that requires the maximum number of chunks. For 4-kB memory pages and 4-byte words, this maximum is given by:

$$\begin{cases} \lfloor u/2 \rfloor & \text{if } 0 < u \leq 682 \\ 1024 - u & \text{if } 682 < u \leq 1024 \end{cases}$$

Table 3.4: Amount of memory to store updates.

Method	Amount of memory for a single chunk	Amount of memory for a page
Single data (SD)	8	$8 \times u$
Several successive data (SSD)	$8 + 4 \times s$	$8 \times c + 4 \times u$
Many data (MD)	$132 + 4 \times u$	$132 + 4 \times u$
Entire page (EP)	4100	4100

One can note that when two successive memory locations have to be stored, the amount of memory needed to store the information for both Single data and Several successive data cases is the same.

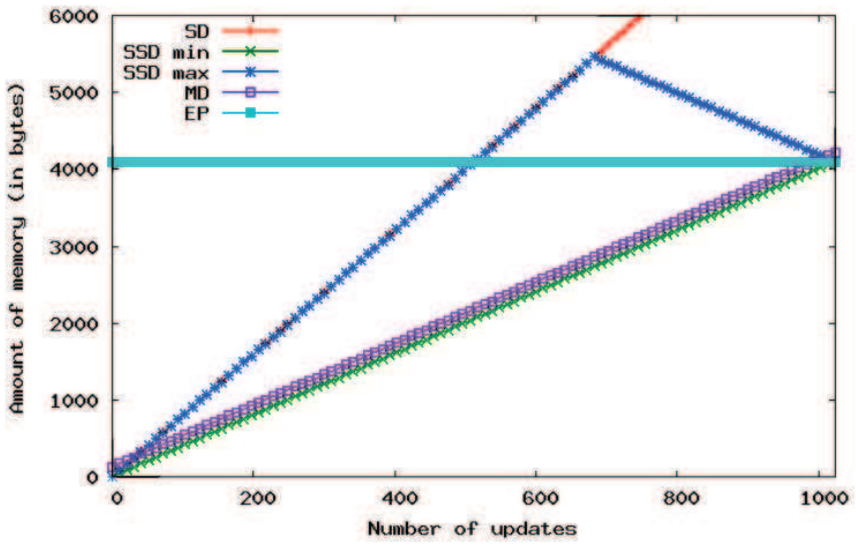


Figure 3.8: Amount of memory to store updates.

The most efficient solution, i.e. the one that reduces the most the memory usage, is identified this way. For each page, first the Many data representation is built. It requires at most 4228 bytes; second, a combination of Single data and Several successive data methods is built, having Single data chosen for isolated data and Several successive data chosen when at least two consecutive memory locations have been updated; third, the shortest representation between both computed is stored. Note that the Entire page method is left to the storage of new pages.

From the expressions provided in Table 3.4, one can demonstrate that the Several

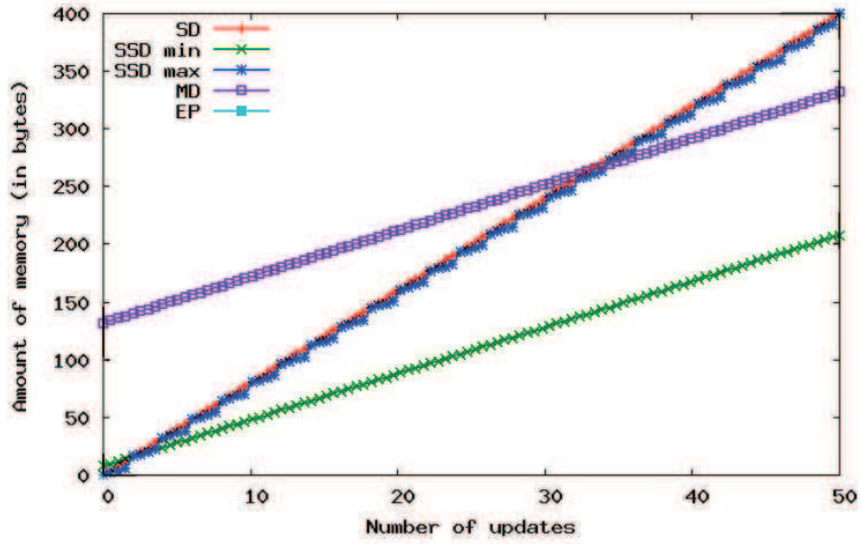


Figure 3.9: Trade-off between SSD and MD.

successive data method is always the most interesting solution when the number of updates is smaller than 34. Then, there is a trade-off between the Several successive data method and the Many data method that depends on the number of chunks. Figure 3.9 is a magnification of Fig. 3.8 for a number of updates in the range from 0 to 50.

3.5.3 Identifying the method

Considering that more than one method is used to store memory updates, it is important to identify which one was used when restoring the content of the checkpoint. A simple solution would have consisted in adding an extra integer or even a character before any data description or set of data description. However, in order to keep the size of checkpoints as small as possible, it has been decided to add no extra byte to the checkpoint.

Instead, considering that all methods require an address as the first field and that these addresses are necessarily aligned on a boundary of a word, i.e. these addresses are necessarily a multiple of 4 or the last two digits of their binary representation are necessarily 00, it is possible to use this "free" space to store which method was used to store the data. In our current implementation, 00 is associated with Single data, 01 with Several successive data, 10 with Many data and 11 with Entire page. When restoring the content of a checkpoint, these two bits are reset to 00 after the storage method has been identified and before the address is effectively used.

3.6 Conclusion

This chapter presented DICKPT, an improved version of the Incremental Checkpointing Technique and some algorithms serving to compress incremental checkpoints. DICKPT provides a new capacity to specify exactly in programs the sections that have been checkpointed. This significantly reduces both the size of checkpoints and the time of checkpointing and recovery. Its most important advantage consists in providing new abilities to improve CAPE execution model, as presented in the next chapter.

Chapter 4

CAPE using Incremental Checkpoints – CAPE-2

Contents

As of now, two versions of CAPE have been developed. The first one that uses complete checkpoints is referred to as CAPE-1, and the second one using incremental checkpoints is called CAPE-2. For the sake of simplification of names, the name CAPE without any additional number refers to CAPE-2 since this point.

4.1 Execution model

The initial idea behind CAPE is the use of checkpoints to implement the fork-join model of OpenMP for parallel constructs. The first important change is the replacement of threads by processes, each process usually running on an independent machine. A program initially runs on a set of machines in which one plays the role of the master process and the others are slave processes. Whenever the program meets a parallel region, the master process distributes jobs to the slave processes by sending incremental checkpoints if they are necessary (fork phase). Each slave process receives a checkpoint if it exists, merges it into the process memory space and executes the divided job. Then, results are collected by taking an incremental checkpoint and sent back to the master process. The master process receives results

from all slave processes and merges them into the process memory space (join phase). This execution mechanism is presented in Fig. 4.1 with two slave processes.

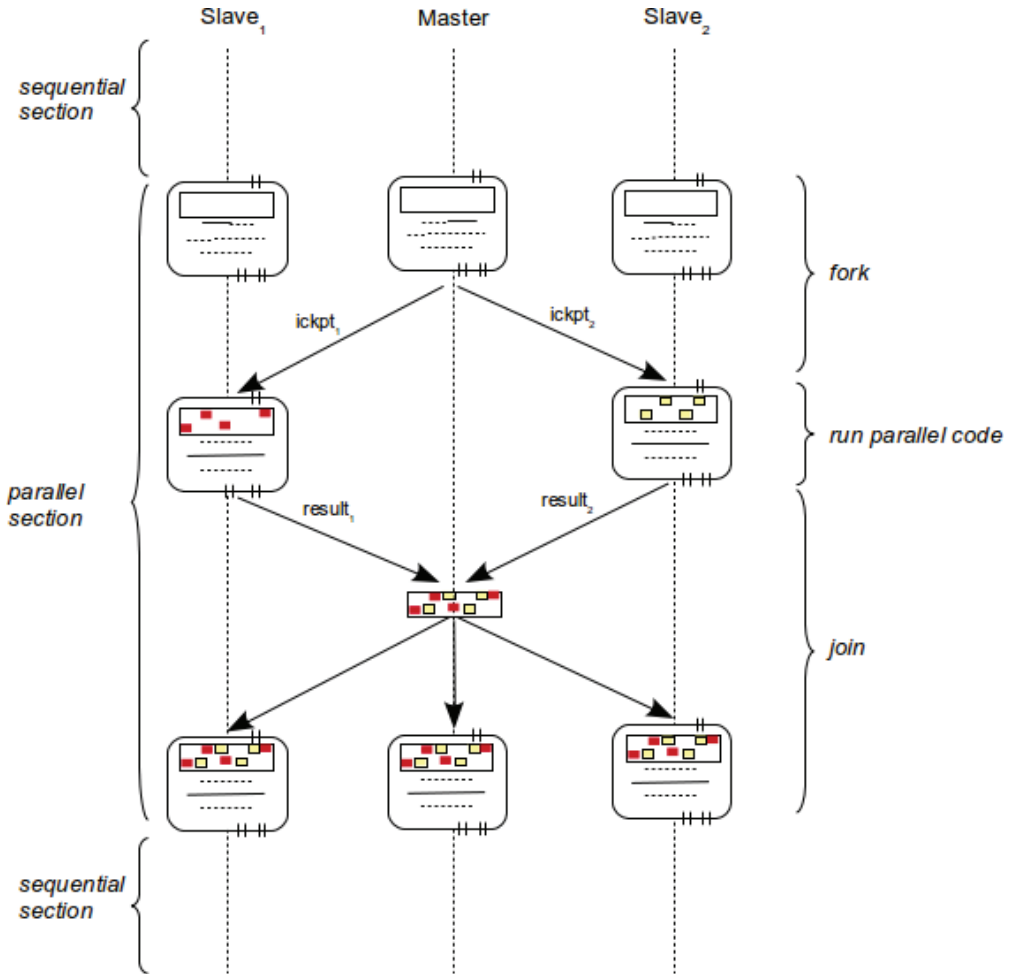


Figure 4.1: Execution of OpenMP programs with CAPE-2.

The model of running sequential sections on all threads/processes and distributing parallel sections on slave threads/processes is used in most OpenMP on distributed-memory systems. In [24], it is called the "One Thread is One Set of Processors" (OTOSP) model (cf 2.2.2.1). Although this model causes unnecessary occupation of CPUs, it provides a simple mechanism to guarantee a coherence of memories in the system. This is performed through the replication of computations performed by processors in the same set, along with the communication of modified memory areas at the end of the parallel construct.

A new and most important advantage of CAPE in this model is the OpenMP compliant property. Programs are automatically transformed without any auxiliary information. It differs from other implementations, such as llCoMP [8] in which an additional directive (`nc_result`) is used to specify the returned data from slave

threads, or in Cluster OpenMP by Intel [11] in which the `sharable` directive is used to specify shared variables.

Although sequential sections are executed on all nodes, sections that are strictly required to be run on a single node, including `master` and `single` sections, may be easily implemented with CAPE as presented in deeper details in Sec. 4.4.

4.2 System organization

In CAPE, each node consists in two processes. The first one runs the application program. The second one plays two different roles: first as a DICKPT checkpointer and second as a communicator between the nodes. As a checkpointer, it catches signals from the application process and executes appropriate handles. In the communicator role, it ensures the distribution of jobs and the exchange of data between nodes. Figure 4.2 shows the principle of this organization.

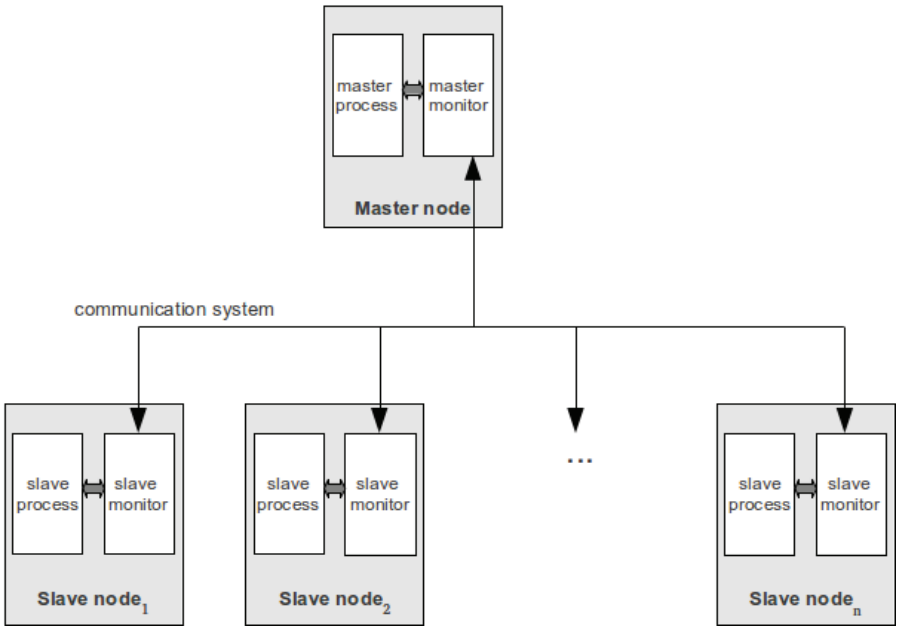


Figure 4.2: System organization.

In the current version, the master node is in charge of managing slave nodes and does not execute any application job in the parallel sections. However, this is not mandatory.

4.3 Transformations primitives

CAPE is based on a set of primitives, some of them are associated with the directives of DICKPT that are presented in Sec. 3.2.2. The others are presented below.

1. `start ( )` clears the buffer and starts or resumes checkpointing. It is associated with the `start` directive of DICKPT.

2. `stop ( )` suspends checkpointing. It is associated with the `stop` directive of DICKPT.

3. `create (` *file* `)` saves the content of the buffer in *file* . It is associated with the `save` directive of DICKPT.

4. `inject (` *file* `)` updates the current process with the information provided in the checkpoint *file* provided as a parameter. Differs from the original `inject` primitive of DICKPT as this primitive does not update the instruction pointer, i.e. after being updated, the process continues to run the instruction following the call to the primitive.

5. `master ( )` returns `TRUE` when executing on the master process and `FALSE` otherwise. This primitive can be derived from the `get_process_num ( )` primitive, presented below.

6. `send (` *file* `,` *node* `)` transfers the content of *file* to *node*.

7. `wait_for (` *file* `)` waits and merges all the components of the *file* .

8. `last_parallel ( )` returns `TRUE` when the current parallel block is the last one of the entire program and `FALSE` otherwise.

9. `merge (` *file_1* `,` *file_2* `)` applies the list of modifications saved in file_2 to checkpoint file_1.

10. `broadcast (` *file* `)` sends *file* to all slave nodes. This function can only be executed on the master node.

11. `receive (` *file* `)` waits for *file* to be available.

12. `get_process_num ( )` returns the process number, i.e. 0 for the master process, 1 for the first slave process and so on.

13. `get_num_processes ( )` returns the number of processes, including the master.

4.4 Transformation prototypes

The transformation of OpenMP programs into the form of CAPE programs is performed by a set of prototypes. At present, CAPE does not allow nested constructs like nested `parallel for` loops and a worksharing construct contained in `parallel` constructs. However, there is no important difficulty to extend the below prototypes to overcome these limits. Furthermore, the cases where worksharing constructs are

nested in `parallel` one can easily be transformed into a parallel worksharing construct as showed in the example below.

```
# pragma omp parallel
{
      # pragma omp for
      for ( A ; B ; C )
            D
}
```

can be translated to:

```
# pragma omp parallel for
for ( A ; B ; C )
      D
```

The above prototype can be applied in cases where `parallel` constructs does not contains a single construct by first dividing them into many `parallel` immediate `parallel` constructs.

The prototypes to translate the most important OpenMP constructs into the CAPE forms are presented in the next section, in which it is assumed that worksharing parts satisfy the Bernstein's conditions. Solutions to overcome this restriction are presented in the Chap. 5. Remind that, as presented in 4.1, CAPE follows the OTOPS execution model where sequential parts of application programs are executed on all processes and the master process distributes jobs of worksharing parts to slaves. Thus, programs initially run on all processes and each time a worksharing construct is met, its jobs are divided and executed on different processes of the system.

Using these prototypes, CAPE compiler translates OpenMP source codes into CAPE forms which do not contains anymore OpenMP directives and constructs. Then, a C/C++ compiler continues to compile them into executable codes that can execute on distributed systems equipped CAPE platform. This compilation chain is shown in Fig 4.3.

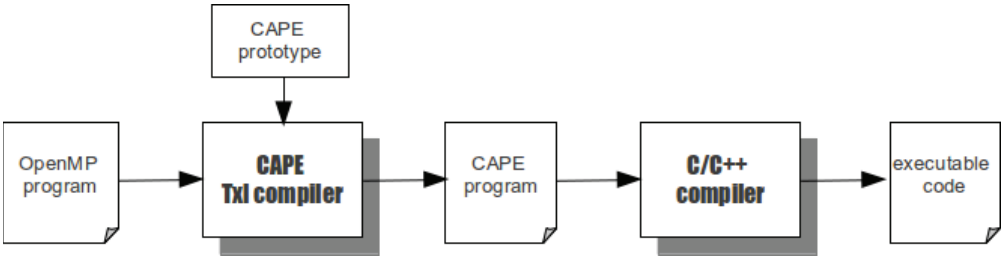


Figure 4.3: Translation OpenMP programs with CAPE.

4.4.1 Prototype for the `parallel for` construct

The `parallel for` construct is the most complex worksharing construct. It is also the most used one, as showed in the Table 4.1, a statistical work in [8] (assume the `for` is transformed to the `parallel for`). The translation of this construct is very general and can serve as a basic for the translation of other worksharing constructs.

	BT	CG	EP	FT	IS	LU	MG	SP
parallel	2	2	1	2	2	3	5	2
for	54	21	1	6	1	29	11	70
parallel for		3					1	
master	2	2	1	10	4	2	1	2
single		12		5		2	10	
critical			1	1	1	1	1	
barrier				1	2	3	1	3
flush						6		
threadprivate			1					

Table 4.1: Number of directives in the NAS Parallel Benchmark codes.

The job division to distribute the `parallel for` construct to slave threads is usually based on the computation of the differences between the low and the high boundaries of the count variable. In the following, let the number of iterations be equal to the number of slave nodes. The effective translation is shown in Fig. 4.4, in which the numbers on the left side are used to make the code easier to read, but are not part of it.

At each `for` loop, the master distributes jobs to each slave by taking and sending it an incremental checkpoint (lines 4 and 5). Since checkpoints are started from the beginning of the loop, they have very small size, usually a very few bytes. As a result, the job distribution is quickly performed.

After receiving an incremental checkpoint (line 16), a slave node inject it into the process memory (line 17) and then execute the corresponding part of the loop (line 19). The execution result on the slave node is then extracted into an incremental checkpoint (line 20) and sent back to the master node (line 22) to update its memory (lines 9 and 10). Finally, the master node broadcasts the result to all slave nodes (line 13) to guarantee the memory consistency if necessary. Then, the loop can be considered as has been executed on all nodes.

Therefore, this prototype performs both the devision jobs and collection execution results of `parallel for` constructs to/from different nodes of distributed systems. One can note that no additional user information is needed, i.e. the translation is transparent to the user and this leads to an OpenMP compliant implementation on distributed-memory systems. This property is one of the most

```
# pragma omp parallel for
    for ( A ; B ; C )
        D ;
```

↓ automatically translated into ↓

```
1   if ( master ( ) ) {
2         start ( ) ;
3         for ( A ; B ; C ) {
4               create ( before ) ;
5               send ( before, slave_x ) ;
6         }
7         create ( final ) ;
8         stop ( ) ;
9         wait_for ( after ) ;
10        inject ( after ) ;
11        if ( !last_parallel ( ) ) {
12              merge ( final, after ) ;
13              broadcast ( final ) ;
14        }
15  } else {
16        receive ( before ) ;
17        inject ( before ) ;
18        start ( ) ;
19        D ;
20        create ( after_i ) ;
21        stop ( ) ;
22        send ( after_i, master ) ;
23        if ( !last_parallel ( ) ) {
24              receive ( final ) ;
25              inject ( final ) ;
26        }
27        else
28              exit ( ) ;
29  }
```

Figure 4.4: Prototype for the `parallel for` with incremental checkpoints.

important advantage of CAPE while comparing with other approaches (cf Sec. 2.2). Furthermore, no common shared-memory mechanism is used in the prototype as each process only access its own memory space and the memory consistence between nodes is quickly performed at only the beginning and the end of worksharing constructs. This one overcomes the need of use commune shared-memory mechanisms for shared variables that significantly reduces the performance of systems (cf Sec. 2.2). Both of these advantages make CAPE becoming a high-performance and compliant implementation of OpenMP on distributed-memory systems.

Compared with the prototype of CAPE-1 presented in Sec. 2.4.1, the above prototype has the following advantages:

1. *Reduction of the amount of data transfered over the network*: the size of the `before` checkpoints are far smaller. In case of CAPE-2, these checkpoints usually contain only very few bytes, while complete checkpoints are stored in the case of CAPE-1. When a broadcast is performed at the end of the construct, the list of modifications is transferred over the network. However, the size of this list is usually smaller than the one of a complete checkpoint. Furthermore, a good implementation of the broadcast can strongly reduce the execution time. As a result, the execution time to distribute jobs to slave nodes is strongly reduced.

2. *Reduction of the number of comparisons to extract the result on slave nodes*: if the size of modified memory regions on slave nodes are smaller than the one of complete checkpoints, it is implicitly reduced.

3. *Reduction of the amount of memory space to store the temporary data*: checkpoints `original`, `target` and `before` in the CAPE-1 prototypes are not used in the CAPE-2. A new buffer for the `after` incremental checkpoint is needed. However, its size is usually far smaller than the one of a complete checkpoint.

4. *No delay time to start/restart processes*: thanks to the `inject` primitive of the DICKPT checkpointer, `before` and `after` checkpoints can be directly integrated into process spaces. Thus, processes are not necessarily started/restarted from checkpoints and as a consequence, the global execution time is reduced.

4.4.2 Prototype for the `parallel sections` construct

Two solutions have been identified to translate the `parallel sections` construct. The first one consists in translating it into a `parallel for` form before applying the existing prototype for this construct. The second one consists in using a dedicated prototype for the `parallel sections` construct.

Figure 4.5 shows an example of translation of a `parallel sections` containing three parts into a `parallel for` loop with three iterations. The translation is quite straightforward: the number of loop iterations is specified while counting the number of section; in each iteration, a branching is performed to select the appropriated job.

Combined with the above `parallel for` prototype, `parallel sections` constructs can be automatically executed on distributed systems with CAPE.

The dedicated prototype to translate the `parallel sections` construct is presented in the Fig. 4.6. It is assumed that the master node does not perform any part of the `parallel sections` construct. Thus, it just waits for execution results from slave nodes (line 2), then injecting them into the process memory (line 3) and broadcasting if necessary (line 5). At the slave nodes side, each one selects a part of the `parallel sections` construct (line 8, 15 or 22), based on its process number. After performing the appropriated part of jobs (line 10, 17 or 24), the node extracts its execution result (line 11, 18 or 25) and send it back to the master node (line 13, 20 or 27). Therefore, `parallel sections` constructs can be automatically distributed and executed on distributed systems with CAPE.

One can note that the solution using dedicated prototype is more complex than the solution that uses immediate `parallel for` constructs. However, the dedicated prototype provides better performance thanks to the elimination of the distribution phase of jobs to slave nodes. Similar with the case of `parallel for` prototype, no additional user information is needed in these both prototypes, so the translation is automatically performed. It also does not use common shared-memory for shared variables. Both of them assure the compliant and high-performance properties of CAPE as an OpenMP implementation on distributed-memory systems.

```
# pragma omp parallel sections
{
#   pragma omp section
    P0 ;
#   pragma omp section
    P1 ;
#   pragma omp section
    P2 ;
}
```

↓ can be translated into ↓

```
# pragma omp parallel for
    for ( i = 0 ; i < 3 ; i ++ ) {
          switch ( i ) {
          case 0 :
                P0 ;
                break ;
          case 1 :
                P1 ;
                break ;
          case 2 :
                P2 ;
          }
    }
```

Figure 4.5: Equivalence between `parallel sections` and `parallel for`.

```
# pragma omp parallel sections
{
    # pragma omp section
    P_0 ;
    # pragma omp section
    P_1 ;
    # pragma omp section
    P_2 ;
}
```

↓ automatically translated into ↓

```
1   if ( master ( ) ) {
2         wait_for ( after ) ;
3         inject ( after ) ;
4         if ( !last_parallel ( ) )
5               broadcast ( after ) ;
6   } else {
7         switch ( get_process_num ( ) ) {
8               case 1 :
9                     start ( ) ;
10                    P_0 ;
11                    create ( after_0 ) ;
12                    stop ( ) ;
13                    send ( after_0, master ) ;
14                    break ;
15              case 2 :
16                    start ( ) ;
17                    P_1 ;
18                    create ( after_1 ) ;
19                    stop ( ) ;
20                    send ( after_1, master )
21                    break ;
22              case 3 :
23                    start ( ) ;
24                    P_2 ;
25                    create ( after_2 ) ;
26                    stop ( ) ;
27                    send ( after_2, master ) ;
28                    break ;
29        }
30        if ( !last_parallel ( ) ) {
31              receive ( after ) ;
32              inject ( after ) ;
33        } else
34              exit ( ) ;
35  }
```

Figure 4.6: Dedicated prototype for the `parallel sections` construct.

4.4.3 Prototype for the `parallel` construct

The `parallel` construct specifies in a programs the regions that are executed in concurrence on all nodes. However, in the following prototype, it is assumed that the master node does not execute any application code, i.e. it does not take any application part of jobs of the construct. The solution to make the master node executing application code of worksharing constructs is presented in the next section.

This construct is the simplest case and is very similar with the `parallel sections` above. There are also two ways to perform the translation. The first one consists in translating it into a `parallel for` form before applying the existing prototype for this construct. The second one consists in using a dedicated prototype for the `parallel` construct.

Figure 4.7 presents the prototype to translate a `parallel` construct into a `parallel for` construct. Since the master node does not execute application codes, the number of loop iterations equals to the number of slave processes. The body of the loop exactly is the application codes of the `parallel` construct, since all slave processes execute this region.

```
# pragma omp parallel
    A ;
```

↓ can be translated into ↓

```
# pragma omp parallel for
for ( i = 0 ; i < get_num_processes ( ) - 1 ; i ++ )
    A ;
```

Figure 4.7: Equivalence between `parallel` and `parallel for`.

Figure 4.8 presents the dedicated prototype to translate `parallel` constructs into CAPE form. This prototype is very similar with the dedicated prototype for the `parallel sections` construct above. The master process just waits for execution results from slave nodes (line 2), then injecting them into the process memory (line 3) and broadcasting if necessary (line 5). At the slave nodes side, each one performs the application code of the construct (line 9), then extracts its execution result (line 10) and send it back to the master node (line 12). Therefore, `parallel` constructs can be automatically distributed and executed on different nodes of distributed systems with CAPE without using any shared-memory mechanism.

```
# pragma omp parallel
    A ;
```

↓ automatically translated into ↓

```
1   if ( master ( ) ) {
2         wait_for ( after )
3         inject ( after ) ;
4         if ( !last_parallel ( ) )
5               broadcast ( after ) ;
6   } else {
7         i = get_process_num ( ) ;
8         start ( ) ;
9         A ;
10        create ( after_i ) ;
11        stop ( ) ;
12        send ( after_i, master ) ;
13        if ( !last_parallel ( ) ) {
14              receive ( after ) ;
15              inject ( after ) ;
16        } else
17              exit ( ) ;
18  }
```

Figure 4.8: Dedicated prototype for the `parallel` construct.

4.4.4 Prototype for the `single` and the `master` constructs

Both `single` and `master` constructs specify regions that should be executed in a single thread and in the case of the `master`, this thread is the master.

Prototypes to translate these constructs are quite simple with CAPE. However, the assumption that sets the master node out of the execution of the application code of worksharing constructs leads to a small problem. On one side, a `master` construct is always binding with an enclosing parallel region, i.e. it can be considered as a part of the application code of this region. On the other side, as presented in the above prototypes, the application code of parallel regions is not executed in the master node with CAPE. Both of them leads to the fact that the application code of the `master` construct can not executed in the master node, i.e. the `master` construct can not be implemented with CAPE while assuming that the master node does not execute the application code of parallel regions.

There are two possible ways to solve the above problem. The first one consists in assigning the role of the "master thread" to a slave node and the former master node only plays the role of a task manager. The second one consists in modifying the prototypes for the worksharing constructs to ensure that a part of the application code of parallel regions is executed on the master node. These modifications are not complex as shown in an example (see Fig. 4.9) in which application codes of the `parallel` construct are also executed on the master node. This modified prototype is the same as the one for the `parallel` construct presented in the previous section, excepts two important modifications. The first one is the insertion of A in the master node (line 3) to make this node running the application code. The second one consists in appending the execution result on the master node to the execution result from the slaves (line 9), before broadcasting the final updating list (line 10).

```
# pragma omp parallel
    A ;
```

↓ automatically translated into ↓

```
1   if ( master ( ) ) {
2         start ( ) ;
3         A ;
4         create ( after_0 ) ;
5         stop ( ) ;
6         wait_for ( after ) ;
7         inject ( after ) ;
8         if ( !last_parallel ( ) ) {
9               merge ( after_0, after ) ;
10              broadcast ( after ) ;
11        }
12  } else {
13        start ( )
14        A ;
15        create ( after_i ) ;
16        stop ( ) ;
17        send ( after_i, master ) ;
18        if ( !last_parallel ( ) ) {
19              receive ( after ) ;
20              inject ( after ) ;
21        } else
22              exit ( );
23  }
```

Figure 4.9: Modified prototype to execute application codes on the master node.

Along with the above modified prototype, the prototype shown in Fig. 4.10 may be used to translate `single` and `master` constructs. The translation is quite simple. The application code is executed only on master node (line 4 or 10), thanks to a test at line 1 and 11. To reduce the impact on execution performance, checkpointing and broadcast are performed only in case necessary (line 3 and lines 5 to 7). Slave nodes just wait for the execution result from the master node (line 13) and inject it into process memories (line 14). Thus, this prototype make `single` and `master` construct run on one node. It also assures the memory consistency between nodes. No additional user information and no shared-memory for shared variables are needed in this prototype, similar as cases of prototypes in the above sections.

One can note that this prototype makes `single` sections run on the master node. This is not mandatory and the prototype can be modified to run these sections on a slave node. However, in the second case, an additional phase has to be added to send the execution result from the slave node to the master node before calling the `broadcast` function.

```
# pragma omp single
    A ;
```

or

```
# pragma omp master
    A ;
```

↓ automatically translated into ↓

```
1   if ( master ( ) ) {
2         if ( !last_parallel ( ) ) {
3               start ( ) ;
4               A ;
5               create ( after ) ;
6               stop ( ) ;
7               broadcast ( after ) ;
8         }
9         else
10              A ;
11  } else
12        if ( !last_parallel ( ) ) {
13              receive ( after ) ;
14              inject ( after ) ;
15        } else
16              exit ( ) ;
```

Figure 4.10: Prototype for both `single` and `master` constructs.

4.5 Performance evaluation

In order to validate our approach, some performance measurements have been conducted on a Desktop Cluster. This testbed is composed of nodes including Intel(R) Core(TM)2 Duo E8400 CPUs running at 3 GHz and 2 GB RAM, operated by Linux kernel 2.6.35 with the Ubuntu 10.10 flavor, and connected by a standard Ethernet at 100 MB/s. In order to avoid as much as possible external influences, the entire system was dedicated to the tests during performance measurements.

The program used for tests is a matrix-matrix product for which the size varies from 3,000×3,000 to 12,000×12,000. Matrices are supposed to be dense and no specific algorithm has been implemented to take into account sparse matrices. Each experiment has been performed at least 10 times and a confidence interval of at least 90% has always been achieved for the measures. Data reported here are the means of the 10 measures.

Size	Sequential	OpenMP
3,000	258.9	142.4
6,000	1,852.7	1,048.7
9,000	7,314.5	3,986.2
12,000	14,990.5	8,999.4

Table 4.2: Execution time (in seconds) on a single node.

The execution of both the sequential version and the OpenMP version of the program on one of the nodes gives the result provided in Table 4.2. A single core was used for the sequential execution of the program, while the OpenMP program takes benefits of the two cores. One can check that results in Table 4.2 are consistent as the execution time for both sequential and OpenMP versions are directly proportional to the cube of the matrix size. Typically, this means that no important cache effects have polluted the performance measurements, probably because almost all data fit into memory. Moreover, the speedup obtained by OpenMP is 1.8 for the first three matrix sizes and 1.65 for the fourth one, which are expected values.

4.5.1 General evaluation

Figures 4.13 and 4.14 present the execution time in seconds of the matrix-matrix program for various number of nodes and matrix size. Note that, despite the fact that processors are dual core, a single core was used during the experiments. Three measures are represented each time: the left one is associated with CAPE using complete checkpoints, the middle one is also associated with CAPE but with incremental checkpoints, and the right one is associated with MPI. The MPI program has been developed for reference as exchanges to keep all processes consistent between nodes are kept minimal.

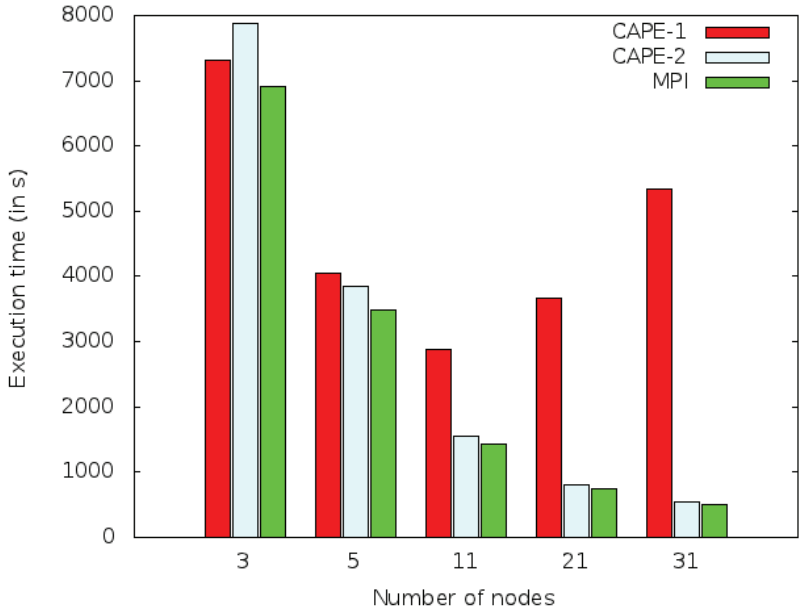


Figure 4.11: Execution time (in seconds) vs. number of nodes.

Figure 4.13 presents the execution time for different number of nodes. The size of matrices are 12,000×12,000. However, similar trends are observed for the other matrix sizes. One can remark that the 3-node case apart, the execution time when using incremental checkpoints is always better than the execution time using complete checkpoints. The larger the number of nodes, the smaller the execution time for both CAPE using incremental checkpoints and MPI. Moreover, the execution time for CAPE using incremental checkpoints is getting closer and closer as the number of nodes is increasing. The case for CAPE using complete checkpoints is different. When few nodes are used for computation (up to 11), the execution time is decreasing as the number of nodes is increasing and the value is quite similar to the other two cases (CAPE using incremental checkpoints and MPI). However, for larger number of nodes, the execution time for CAPE using complete checkpoints is directly proportional to the number of nodes. This is due to the amount of data that is transmitted over the network which is getting very important (there is at least one complete checkpoint for each slave node) even though the amount of data that are effectively interesting for each slave node is reduced. This clearly justifies the use of incremental checkpoints for CAPE.

At first, the performance for three nodes may look strange as the execution time of the program with CAPE using complete checkpoints is better than the execution time with CAPE using incremental checkpoints. In fact, for small number of nodes, the amount of data transmitted over the network between the different nodes is almost the same for both complete and incremental checkpoints, as in the case of incremental checkpoints, slave nodes receive a big part of matrices. However, in

the case of incremental checkpoints, processes are monitored in order to capture the memory pages that are accessed for writing. The monitoring of the slave processes involves a computing overhead that is reduced proportionally with the amount of computation, and therefore with the number of nodes, when a large number of nodes is used. Fortunately, this is not a problem for CAPE. Processors with 4 and even 8 cores are available on the market and, as a result, CAPE is targeting distributed architectures with a far larger number of nodes.

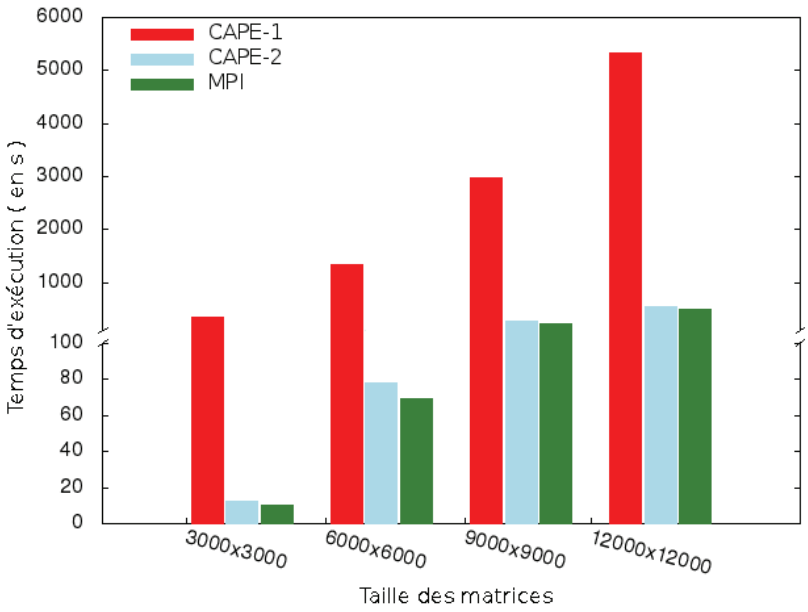


Figure 4.12: Execution time (in seconds) vs. problem size.

Figure 4.14 presents the execution time for difference matrix sizes. The number of nodes involved in the parallel machine is 31. However, remarks below would be the same with other number of nodes. The figure clearly shows that the execution time for CAPE using complete checkpoints is directly proportional to the square of the matrix size, while the execution time for both CAPE using incremental checkpoints and MPI is directly proportional to the matrix size. This is due to the fact that the virtual address space of the processes is mainly composed of the matrices, and that the complete virtual address space is transmitted over the network for complete checkpoints. However, for CAPE using incremental checkpoints and MPI, the complete virtual address spaces are not transmitted over the network and only the data that have been updated during the computation of the matrix-matrix product are considered. Moreover, one can remark that the execution time for CAPE using incremental checkpoints and MPI are very close. An in-depth analysis of the performance results shows that the execution time for CAPE using incremental checkpoints is only 10% higher than the execution time for MPI, except for 3,000×3,000 matrices where the ratio is 1.3 .

4.5.2 Detailed analysis

To more clearly know which elements affect on the global performance, a deeper analysis has been performed. This one also gives advices to choose the best solution when implementing CAPE.

Figures 4.13 and 4.14 present the execution time in seconds for the different phases. Three measures are represented each time: the left one is associated with CAPE-1, the middle one is also associated with CAPE-2, and the right one is associated with MPI.

For both figures, two series of graphs are provided. Upper series are related to the master node, while lower series are associated with slave nodes. Each series is composed of four graphs (refer Fig. 4.4 for line numbers):

- *Init* is the elapsed time between the beginning of the program and the beginning of the parallel for loop in the matrix-matrix product. On Fig. 4.4, these are all lines before the first one.
- *Before* is the time spent to create and send checkpoints (lines 2 to 5) on the master node. On slave nodes, this includes waiting for and receiving the checkpoint, and then updating the slave process using the checkpoint (lines 16 and 17). For the case of MPI, this is the time to send data to slave nodes.
- *Compute* is the time to generate the last checkpoint on the master node (lines 7 and 8) and the time to do the job (execute the code of matrix-matrix product) on the slaves (lines 18 and 19).
- *Update* is the time to wait for and receive all updates from the slave nodes and inject them into the master process memory (lines 9 and 10). On slave nodes, this is the time to generate the incremental checkpoints and send them to the master node (lines 20 to 22). For the case of MPI, this is the time to send results from slave nodes to the master node.

Figure 4.13 presents the execution time in the detailed phases with different number of nodes. The size of matrices are 9,000×9,000. However, similar trends are observed for the other matrix sizes.

On the master node, the most important phases are *Before* and *Update*. It is the consequence of not running the application code on the master node. In the *Before* phase of CAPE-2 and MPI, the time is very small and quite similar for the different number of nodes. It is very different for the case of CAPE-1, as this time is very important and directly proportional to the number of nodes. In phase *Update*, CAPE-2 has the biggest values for the cases with small number of nodes. However, for the last case, CAPE-1 has the biggest one. Theoretically, this phase has to be similar for all three methods. Differences here are due to the time on the master to wait for updated data from slave nodes.

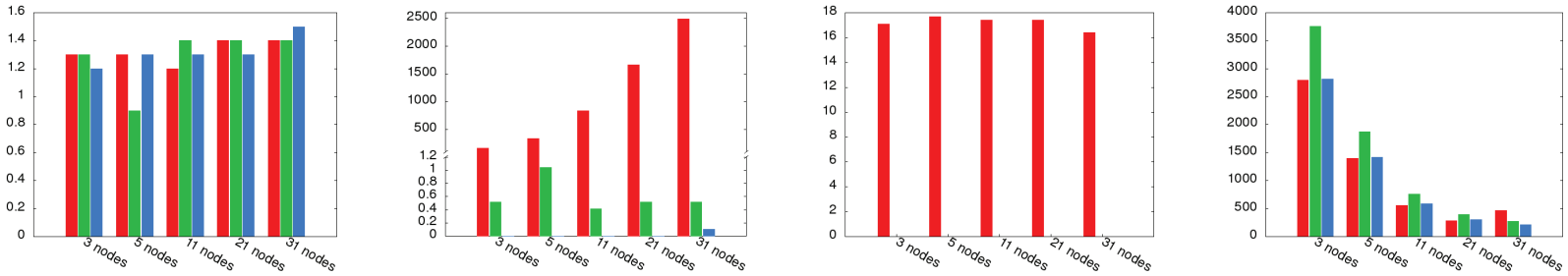
On slave nodes, the most important phase for CAPE-2 and MPI is the *Compute* and this proves the reasonableness of the new approach: the application job is

divided for the slave nodes with a small cost on auxiliary phases. This is very different in the case of CAPE-1, where the phases for job distribution and result collection take a long time. The affectation of the incremental checkpointing on the execution time of the checkpointed process is clearly shown in the *Compute* phase, where the time is always longer with CAPE-2. However, the larger the number of nodes, the smaller the difference between CAPE-2 and the two others. This explains why the global execution time of CAPE-2 is larger for 3 nodes and then closer and closer to the one of MPI while increasing the number of nodes.

Figure 4.14 presents the execution time for difference matrix sizes. The number of nodes involved in the parallel machine is 31. However, the remarks below would be the same with other number of nodes. The figure shows that the execution time of phase *Before* is very small for CAPE-2 and MPI and does almost not depend on the size of the matrix. It is very different for CAPE-1 where this time is very important and directly proportional to the square of matrix size. It is similar with phase *Final* on the master node, except that the execution time in this phase is less important. The figure also shows that phase *Update* is quite similar for CAPE-2 and MPI, and always larger for CAPE-1. This is due to the time on the master to wait for the updated data from slave nodes, as mentioned above.

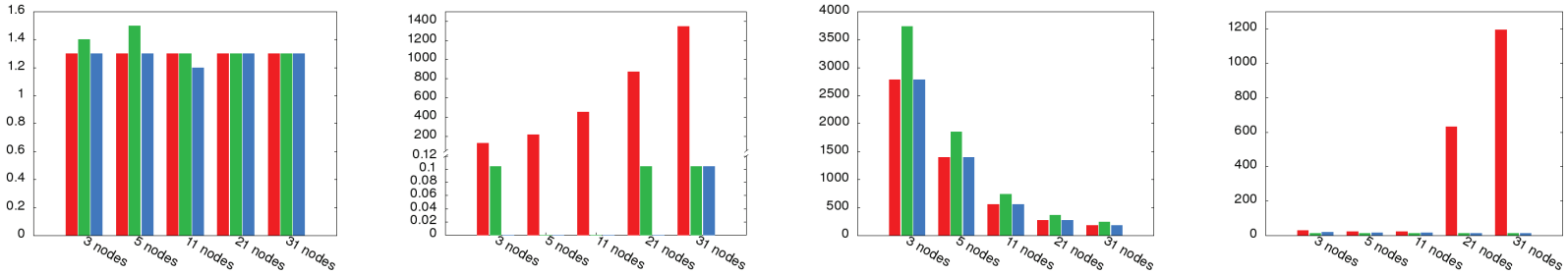
Note that graphs for phase *Compute* on the master node on Fig. 4.13 and 4.14 do not show any data for CAPE-2 and MPI as the execution time for both is far too small to be represented.

On the master node.



(a) Init (b) Before (c) Compute (d) Update

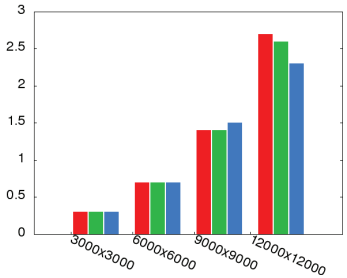
On slave nodes.



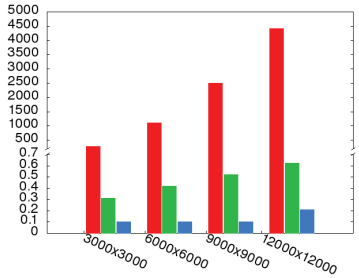
(e) Init (f) Before (g) Compute (h) Update

Figure 4.13: Execution time (in seconds) vs. number of nodes.

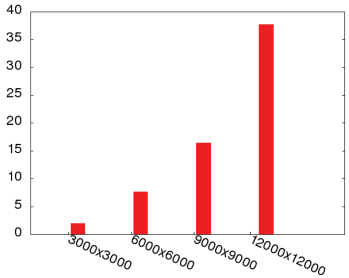
On the master node.



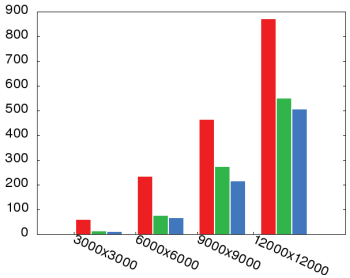
(a) Init



(b) Before

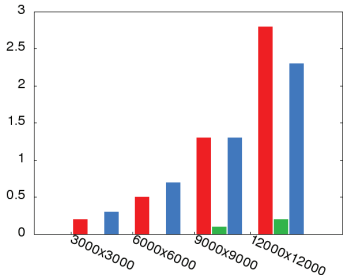


(c) Compute

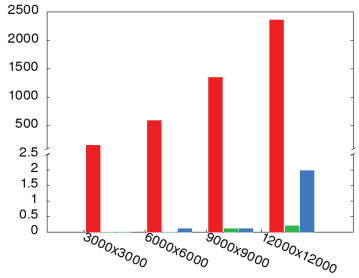


(d) Update

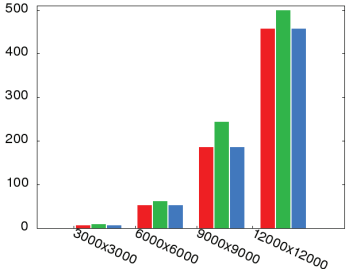
On slave nodes.



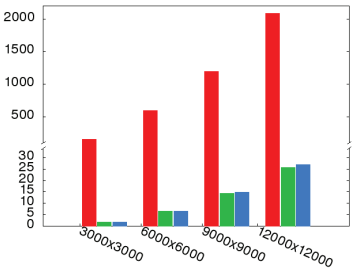
(e) Init



(f) Before



(g) Compute



(h) Update

Figure 4.14: Execution time (in seconds) vs. problem size.

4.5.3 Speedup

Figure 4.15 shows the speedup of CAPE using incremental checkpoints for various number of nodes and matrix sizes. The dotted line represents the theoretical maximum speedup. The figure clearly shows that the solution provides an efficiency (the ratio of the speedup over the number of nodes) in the range from 75% to 90% which is very good. Also, it highlights that the larger the size of matrices, the higher the speedup, which was not the case with the complete checkpoint implementation.

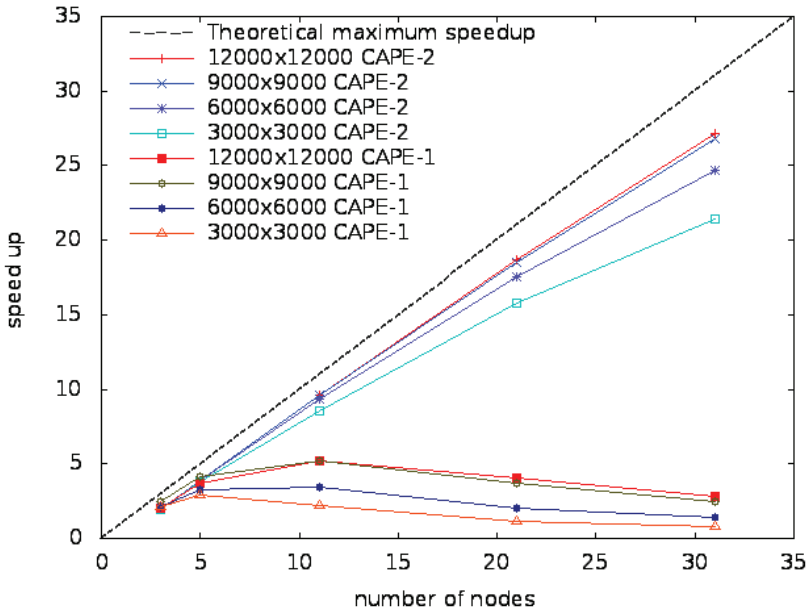


Figure 4.15: Speedup vs. number of nodes.

4.6 Conclusion

This chapter presented CAPE-2, i.e. CAPE based on incremental checkpoints. With the new capabilities of the DICKPT technique, this version has overcome the four first weak points of CAPE-1 (cf Sec. 2.4.2): reduction of the amount of data transfered over network, reduction of the number of comparison operations to extract execution results, reduction of the amount of memory space to store temporary data, and elimination of the `restart` process phase to integrate checkpoints into process spaces. All of these improvements significantly increase the global performance of CAPE. The last problem remaining in CAPE, relates to the requirement to match the Bernstein's conditions in application programs. This is solved in the next chapter.

Chapter 5

Data Sharing

Contents

OpenMP is based on a relaxed-consistency shared-memory model [1]. All OpenMP threads have access to a place to store and retrieve variables, called the *memory*. Additionally, each thread is allowed to have its own *temporary view* of the memory. The temporary view of the memory for each thread is not a required part of the OpenMP memory model. However, it allows to represent any kind of intervening structure. This memory model is strongly appropriate with the fork-join execution model in which the program initially executes in a single thread called the master thread; each time a parallel region is reached, the master thread is derived (fork) into a group of slave threads; then, the master thread divides the job to slave threads and waits for their results; slave threads terminate after their job and join in the master thread after the parallel region. As threads use a shared memory space, this execution model implicitly matches the shared-memory model of OpenMP. Furthermore, these properties are supported by most of the modern programming languages so that OpenMP can be easily implemented in SMP systems.

The context is completely different with distributed memory architectures. Typical threads cannot be created and run on a remote machine, nor they can directly share their respective address spaces except when using a DSM or a SSI. As a result, OpenMP implementations for distributed systems start programs on all machines, each machine executing a part of the parallel regions and the sequential regions being executed only on one (the master) or on all machines.

5.1 Shared-memory models on distributed systems

One of the most important element to take into account to built an OpenMP compliant is to cope with its shared-memory model. The approach using a SSI or a DSM as the common shared memory of all threads on distributed systems is a straightforward idea. However, the global address space is located across machines and causes a strong overhead to the global performance. This is caused by both the delay to access the remote memory and the synchronization mechanism to access the shared memory: the larger the number of processes, the longer the delay to access the shared memory. This was clearly shown in an experiment using Kerrighed SSI to run OpenMP programs on distributed systems (cf Sec. 2.2.1.1).

To reduce the overhead of the use a common shared memory across distributed systems for all threads, there are two main approaches, as presented in Sec. 2.2. The first one consists in mapping only the shared variables to the common shared memory. The second one focuses on using local memories for threads and building a mechanism to synchronize these memories.

The most difficult problem when mapping only the shared variables to the common shared memory is the identification of shared variables. This requires explicitly declare shared variables that are located in the common memory, while in OpenMP all variables are implicitly shared. Solution to this problem consists in extending OpenMP with additional directives to specify shared variables. However this means that the implementation is no longer fully compliant (cf Sec. 2.2.1.2).

In the approach that consists in using local memories for threads, shared data could be copied into several nodes. However, this leads to consistency issues when write operations on shared data have to be seen by the other processes. To solve this problem, most of today DSMs use the page-fault mechanism to catch accesses to shared variables. With this mechanism, all pages in shared regions are initially set to the read-only status to force the system to generate a page-fault signal whenever there is a write operation to one of the pages of this region. This signal is then caught and the whole page is sent to all the other processes. When many write operations occur on shared regions, the amount of exchanged data strongly increases which leads to a significant reduction of the global performance. As a result, in order to reduce the impact of the synchronization of shared memory regions, relaxed-consistency memory models are used.

The most restrictive model is *Sequential Consistency* [80] that ensures the result on any execution of a multiprocessor system is the same as if operations of all

processors would have been executed in the sequential order, and operations of each individual processor appear in this sequence in the order specified in the program. This model requires the memory system to propagate updates early and prohibits optimizations. To improve the performance, other models have been proposed to relax the constraints of the sequential consistency.

Relaxed Consistency (RC) model [81] [16] allows the propagation and application of coherence operations (e.g., invalidations) to be postponed to synchronization points. Synchronized memory accesses are divided into *Acquires* and *Releases* where an Acquire allows the access to shared data and ensures that data are up-to-date and a Release relinquishes this access right and ensures that all memory updates have been properly propagated. By greatly reducing the impact of false sharing and the frequency of coherence operations, the performance of relaxed consistency models are usually far better than the one of sequential models [82]. OpenMP uses this model as its standard memory model.

Lazy Release Consistency (LRC) [16] is a specific variant of the relaxed consistency model presented above. Instead of propagating modifications to the shared address space on each release, modifications are further postponed until the data is actually needed. To reduce the communications involved by false sharing, where multiple unrelated shared data are located on the same page, the LRC protocol usually supports a multiple-writer scheme. Within this scheme, multiple writable copies of the same page are allowed and a clean copy is generated after an invalidation. An implementation of LRC is *Home-based Lazy Release Consistency* (HLRC) [17], in which, each shared page has a home page. This home page always hosts the most updated content of the page, which can then be fetched by a non-home node that needs the last version. An example of use of this model, used to implement OpenMP on distributed systems, was shown in [7].

5.2 OpenMP `flush` directive and the Updated Home-based Lazy Release Consistency model

The OpenMP standard is based on the RC shared memory model (cf Sec. 5.1) where all threads share a common memory. However each thread can also execute on its own local memory which is a temporary view of the common memory. The consistency between the local memory and the common memory is performed through the use of the `flush` directive. Note that a call to `flush` enforces the consistency between a thread's temporary view and the memory, and does not affect the other threads [1]. Therefore, to ensure that a value written to a variable by one thread may be read by another thread, the programmer must make sure that the second thread has not written to this variable since the last variable's `flush`, and that the following sequence of events happens in this specific order:

1. The value is written to the variable by the first thread.
2. The variable is flushed by the first thread.

3. The variable is flushed by the second thread.

4. The value is read from the variable by the second thread.

There are two types of `flush` in OpenMP: one specifying a set of variables called the *flush-set* and another one without any parameter. In [6], they are called *selective flush* and *global flush* respectively and the same names are used in this document. For the selective flush, consistency is applied on the given flush-set, while for the global flush the consistency is applied on the whole memory space.

Considering the advantages of high performance that have been proved in [17][83], the HLRC model has been used with some modifications to implement the memory model of CAPE.

5.2.1 Updated Home-based Lazy Release Consistency model

As presented in Sec. 4.1, the basic implementation of CAPE in homogeneous systems ensures the consistency between the memory of the master thread and the ones of slave threads in the sequential regions and at the begin and end points of parallel regions. In the beginning and in sequential regions, all threads run the same set of instructions, so that they have the same memory spaces in these regions. It is the same case for the beginning of parallel regions (before the division into jobs of parallel constructs). At the end of parallel constructs, all threads inject the same set of updated memory items that makes their memory spaces becoming consistent. Thus, the only problem remaining to ensure the consistency between threads is implementing a mechanism for `flush` directives.

For most approaches using the HLRC model, a `flush` on a page located on a distant node involves three main phases:

1. On the distant node: compute the differences between pages (called the *diffs list*) by using the *diff* function [17], and then sending those differences to the home node.

2. On the home node: apply the received differences to the home page, compute the differences between pages and send those differences to the distant node.

3. On the distant node: apply the received differences to the page.

For the global flush, the above phases are applied for each process shared page.

In the case of CAPE, the above algorithm can be directly used by considering the master node is the home node and its memory pages are home pages. However, as a result of using checkpointing, the cost of the `flush` execution can be reduced in two ways. First, on slave nodes, the `create` function of incremental checkpointers can replace the `diff` function as they both do exactly the same job. Second, on the master node, the number of comparisons can usually be significantly reduced if a list of updated memory items of all shared pages is maintained and the `diff` function

is applied on this list instead of home pages. From another point of view, this list contains the immediate execution results of the slave nodes that have called the global flush functions. So, this result may be merged into the last result at the end of the parallel region. As a replacement of the updated list for the set of home pages, this model is called the Updated Home-based Lazy Release Consistency (UHLRC) model. Two other main operations are necessary to implement the mechanism:

1. The initialization of the updated list and merging this list with the `after` checkpoint on the home node. For the first one, after having divided jobs to slave nodes, the master thread creates a null list as the updated list. The second one is executed by a `merge` instruction after receiving the `after` checkpoint. Thus a part of the prototype in Fig. 4.4 is modified as in Fig 5.1 where additional line 8a initializes the updated list and line 9a merges this list into the `after` checkpoint.

```
...
8       stop ( )
8a      init ( update_list )
9       wait_for ( after )
9a      merge ( update_list, after )
...
```

Figure 5.1: Modified prototype for CAPE to implement the `flush` directive.

2. The organization of a mechanism to catch `flush` requests for the synchronization between slave nodes and the master node. In this design, an auxiliary task is added to the monitor–checkpointer (see Sec. 4.2) on each node and executed in event-driven mode. Each time a `flush` request occurs on a slave node, a signal is sent to the local monitor. This monitor then coordinates with the one on the master node to execute all `flush` operations. Note that in our design, that uses the DICKPT checkpointer, both the buffer for checkpoints and the checkpoints themselves are saved in the memory space of the monitor. As a consequence, the `update_list` is also contained in this memory space to ease the processing. From this point, the name monitor is used for the object that contains also the role of a checkpointer.

5.2.2 Global flush using the UHLRC model

A global flush on a slave node enforces the consistency between the local process memory and the master process memory. Follow the mechanism presented in the previous section, not whole memory spaces but only their updated regions are exchanged between nodes. Furthermore, on slave node, an incremental checkpoint is exactly a list of updated regions (*diffs list*) on the node and a update list containing

all updated regions from slave nodes is maintained on master node, as presented above. As a result, jobs of global flush can easily be performed while using incremental checkpoints and the function to create them of the incremental checkpointer. First, the *diffs list* is created using the `create` function on the slave node and sent to the master node. Then, the master node computes differences between the received list and the exist update list on the node and send them back to the slave node. Finally, the master node updates its update list and the slave node updates its process memory. Therefore, all updated regions in the slave node memory are updated into the master node and all the updated regions on the master node are injected into the memory of the slave node, i.e. jobs of global flush are performed. Details of this algorithm is presented in Fig. 5.2.

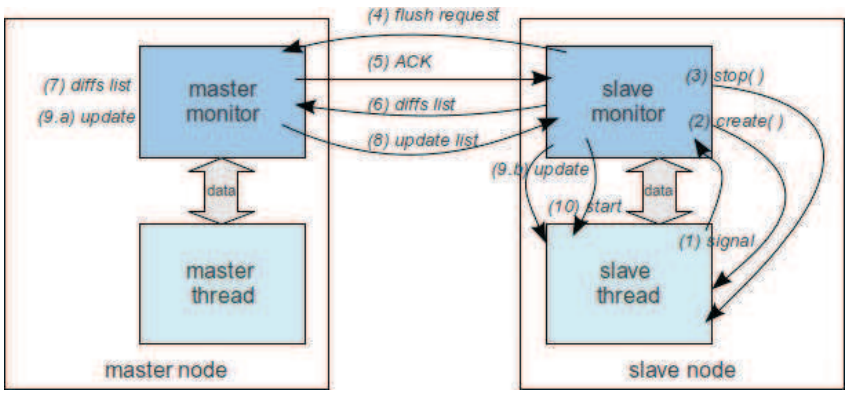


Figure 5.2: Global flush.

1. On slave nodes

- step 1: the slave thread sends a signal to the local monitor and stops its execution.
- steps 2–3: the monitor uses the `create` function (see Sec. 4.3) to create an incremental checkpoint that takes the role of a *diffs list* in this case, and saves this list in its memory space. Then, it calls the `stop` function to temporary suspend the checking process on the slave thread.
- step 4: the local monitor sends a request to the master's monitor and waits for the acknowledgement.
- step 6: after receiving the acknowledgement, the local monitor sends the *diffs list* to the master's monitor and waits for the returned data.
- steps 9b, 10: After receiving the returned *update list*, the local monitor merges it into the process memory space, calls the `start` function to resume the checking process and notifies the slave thread to resume its execution too.

2. On the master node

- step 5: after the master's monitor has received the *flush* request, it sends an acknowledgement to the slave's monitor and waits for the *diffs list*.

- steps 7, 8: after the master's monitor has received the *diffs list*, it computes the differences between the current updated list and the received list and sends the result back to the slave's monitor.

- step 9a: the master's monitor applies the received *diffs* list to the current updated list.

5.2.3 Selective flush directive using the UHLRC model

This case is slightly different from the global flush case and the associated algorithm is far simpler since only variables in the flush-set are synchronized. The below algorithm is designed for the specific case where the flush-set contains a single variable. For more than one variable, the solution is derived from the case with a single variable. Also note that OpenMP does not distinguish between reading or writing from/to the memory to/from the temporary view of the thread. This requires to use a special mechanism to check whether variables are modified, as presented in the next subsection.

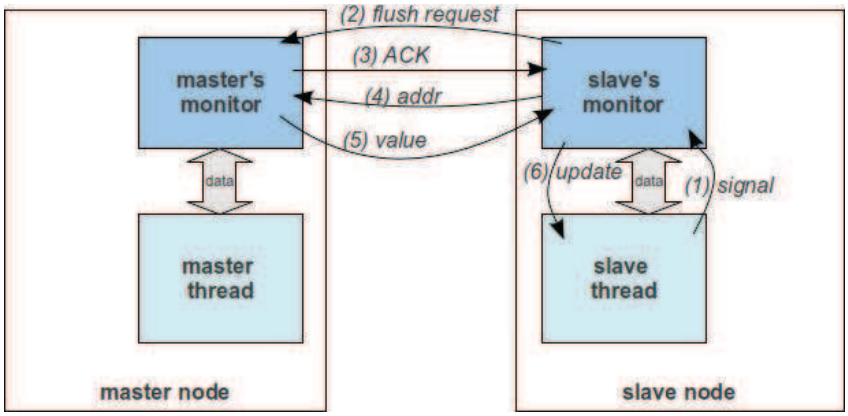


Figure 5.3: Selective flush, read case.

1. On slave nodes

- step 1: the slave thread sends a signal to the local monitor.

- step 2: the slave's monitor check whether the variable is modified since the last `flush`. If yes, this mean that this is a write flush. Otherwise, this is a read operation. Then, it sends a request to the master's monitor and waits for the acknowledgement.

- In case of a read flush:

  - step 4: after receiving the acknowledgement, the local monitor sends the address of the variable to the master's monitor and waits for the returned value.
  - step 6.a: after receiving the value, if it is not null, the monitor merges it into the process memory space. Then, the monitor notifies the slave thread to resume its execution.

- In case of a write flush:
  - step 4: the local monitor sends both the address and the value of the variable to the master's monitor and notifies the slave thread to resume its execution.

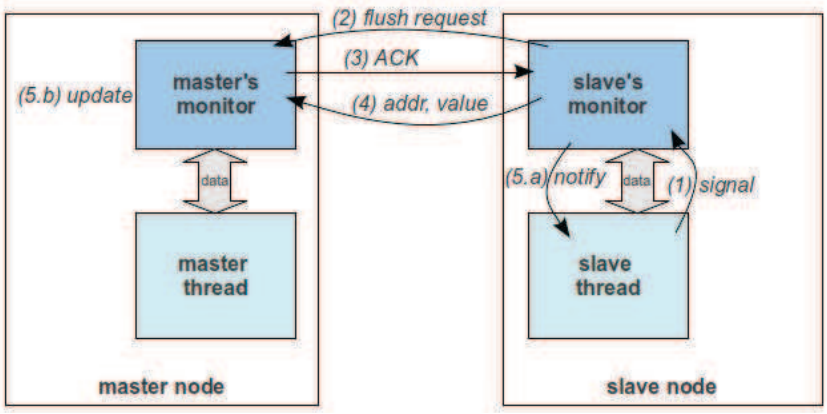


Figure 5.4: Selective flush, write case.

2. On the master node

- step 3: after receiving the flush request, the master's monitor sends the acknowledgement to the slave's monitor and waits for data.
- In case of a read flush:
  - step 5: after receiving the address of the variable, the master's thread searches for the address in the updated list. When found it reads and sends back the updated value to the slave's monitor. Otherwise it sends back a null value to the slave's monitor.

- In case of a write flush:
  - step 4: the master's monitor receives the address and the value of the variable, updates this value into the updated list and keeps on executing.

5.2.4 Mechanism to check whether variables are updated since the last `flush`

In OpenMP 3.1, the `flush` operation enforces the consistency between the temporary view of a thread and the memory. There is only one type of `flush` directive for the both two cases of reading from and writing to memory. Theoretically, a `flush` on a variable performs a write-flush if variable's value has been changed from the last `flush` or from the beginning of the program execution. Otherwise, it is a read-flush.

For an implementation targeting distributed-memory systems with the RC model, there is a difficulty to recognize whether a variable has been updated. The straightforward solution consists in comparing its current value with its initial saved value in the buffer (in the checkpoint buffer in case of CAPE). This mechanism leads to a false result for a write operation on a variable without changing its value.

Consider the example below, in which thread_A and thread_B make a `flush` on shared variable `x` having its initial value equal to 0.

thread_A	thread_B	x value in shared memory
x = 1		0
flush(x)		1
	x = 0	1
	flush(x)	?

While the first `flush` is considered as a write-flush, the second one is considered as a read-flush because until the current moment, in the local memory of thread_B, the value of variable `x` has not been changed. Therefore, instead of the desired 0 value for variable `x` in both (shared) memory and thread_B's local memory after the execution of the `flush` operation, value 1 is kept and updated. An alternative but not complete solution is adding before command `x = 0`, a command to assign `x` with another value and make a `flush` on it.

At the side of implementing a compiler for OpenMP, there is no perfect solution for this ambiguous problem. Below are some directions with their advantages and drawbacks.

The direction that uses the page-fault mechanism to recognize all write operation leads to requirement of re-lock all memory pages after each memory update and a bad consequence is the reduction of the global performance. Furthermore, for a write on an adjacent block of memory containing many variables, only one variable at the begin of block can be recognized to have been written in case this write does not change the value of this block. This is a consequence of using the SIGSEGV signal in the page-fault mechanism to recognize the updated memory and the lack of the length of the updated memory block. This means that this mechanism can provide the start location of updated memory block but not its size.

With CAPE, a possible solution consists in adopting for each page in the checkpoint buffer a map which contains updated flags for each byte/word in the page. These maps are updated in three cases. The first one consists in write operations.

As a result, this requires to catch all memory write operations to update flags of the associated memory regions. The second case is performed in the selective flush to update flags of variables in the flush-set. Finally, a global flush resets all maps to their initial status. This solution can completely solve the above ambiguous problem. However, the mechanism to maintain updated maps for all updated pages significantly decreases the global performance in cases application program performs a large number of memory write operations.

Another solution consists in requiring programmers to supply information to distinguish the cases of write and read `flush`. However, this leads to an un-compliant OpenMP. Another good solution would be the modification of OpenMP specifications to explicitly separate the two cases implementation of the `flush` operation.

5.3 OpenMP data-sharing rules implementation

5.3.1 OpenMP data-sharing categories on CAPE

Along with the relaxed-consistency shared memory model, OpenMP classifies variables into three categories:

- Shared: accessible from all threads.
- Threadprivate: local to a thread, unaccessible from another thread.
- Private: local to a construct or a region, unaccessible from outside the construct or region.

Implicitly, all variables are shared. The other cases are specified by data-sharing attributes and through the use of data-sharing primitives and clauses.

From the memory process memory structure (see Sec. 3.1.2), the principle of CAPE and the UHLRC model, attributes for variables in different regions of a process memory are:

- Data and BSS regions: shared. Their addresses are coherent on all processes and synchronizations are done at the beginning and the end of worksharing constructs, at the end of single constructs and when using `flush` directives.
- Heap region: shared. This case is quite complex as the address of a dynamic variable may be different when allocated in different slave processes. The heap's sizes may be also different on different processes. Thus, checkpoints created in these processes are different and cause incorrect updates when injected into process memories. To overcome this problem, a mechanic to ensure the coherence of dynamic variables over all the system is required. A possible solution consists in catching all memory allocation requests on slave processes and ensuring the consistency of the virtual address spaces on processes using the following steps:

1. A `malloc or free` or friend request is caught on a slave thread by the local monitor.
2. The local monitor sends a request to the master's monitor.
3. The master's monitor sends a request to the master process.
4. The master process executes a dynamic allocation and returns results to the master's monitor.
5. The master's monitor sends the received results to the slave's monitor.
6. The slave's monitor sends received results to the slave process.
7. The slave process updates the received results into process memory.

Along with the above mechanism, the heap region is shared and consistent on all processes.

- Stack region: shared or threadprivate. Variables which are put in the stack by the code of sequential sections of application programs are coherent in all processes, so they are shared. This case is similar with the case when running the program on SMP systems. Variables which are put in the stack by the code of worksharing sections are local to the thread running this code. Thus, these variables are threadprivate. From this point, the name *Stack1* is used to refer the stack region of the first case while the name *Stack2* refers to the regions of the second case.

- Other regions: shared. It is the case of text, shared libraries, program arguments and environment regions. These regions are coherent in all processes and thus variables allocated in them are shared.

As a result, in CAPE, all variables are implicitly shared except variables allocated in Stack2 regions. This is compatible with the OpenMP memory model, in which all threads share a common memory and each thread may have a local stack. Therefore, in CAPE, `threadprivate` and `private` variables should be implicitly or explicitly allocated in Stack2.

5.3.2 Implementation of OpenMP data-sharing attribute rules

OpenMP provides a set of rules specifying data-sharing attributes. In [1], they are presented in Sec. 2.9.1. Below is the list of rules available for C and C++.

5.3.2.1 *General rule*

- Data-sharing attributes of variables that are referenced in a construct can be predetermined, explicitly determined, or implicitly determined, according to the rules in the OpenMP specifications.

- Specifying a variable on a `firstprivate, lastprivate` or `reduction` clause of an enclosed construct causes an implicit reference to the variable in the enclosing construct.

5.3.2.2 *Detailed rules*

Some variables and objects have predetermined data-sharing attributes as follows:

- R1: Variables appearing in `threadprivate` directives are threadprivate.
- R2: Variables with automatic storage duration declared in a scope inside a construct are private.
- R3: Objects with dynamic storage duration are shared.
- R4: Static data members are shared.
- R5: Loop iteration variables in the associated for-loops of a `for` or `parallel for` construct are private.
- R6: Variables with const-qualified type having no mutable member are shared.
- R7: Variables with static storage duration that are declared in a scope inside a construct are shared.
- R8: Loop iteration variables in the associated for-loops of a `for` or `parallel for` construct may be listed in a `private` or `lastprivate` clause.
- R9: Variables with const-qualified type having no mutable member may be listed in a `firstprivate` clause.

The data-sharing attributes of variables that are referenced in a region, but not in a construct, are determined as follows:

- R10: Variables with static storage duration declared in called routines in the region are shared.
- R11: Variables with const-qualified type having no mutable member, and that are declared in called routines, are shared.
- R12: File-scope or namespace-scope variables referenced in called routines in the region are shared unless they appear in a threadprivate directive.
- R13: Objects with dynamic storage duration are shared.
- R14: Static data members are shared unless they appear in a `threadprivate` directive.
- R15: Formal arguments of called routines in the region that are passed by reference inherit the data-sharing attributes of the associated actual argument.
- R16: Other variables declared in called routines in the region are private.

5.3.2.3 Ways to satisfy the OpenMP data-sharing rules on CAPE

Table 5.1 presents the positions of variables associated with the above rules and the ways to satisfy these rules. The names of memory regions in the second column are specified in Sec. 5.3.1.

Table 5.1: Ways to satisfy the OpenMP data-sharing rules.

	Allocation				Status		
Rule	Stack	Heap	BSS	Data	Other	Implicit	Explicit by
R1	✓						the `threadprivate` implementation
R2	✓					✓	
R3		✓				✓	
R4			✓	✓		✓	
R5	✓						the `private` implementation
R6	✓				✓	✓	
R7			✓	✓		✓	
R8	✓						the `for` or
							the `parallel for` implementation
R9	✓						the `firstprivate` implementation
R10			✓	✓		✓	
R11					✓	✓	
R12	✓		✓	✓		✓	the `threadprivate` implementation
R13		✓				✓	
R14	✓		✓	✓		✓	the `threadprivate` implementation
R15	✓	✓	✓	✓	✓		inheriting the other rules
R16	✓					✓	

5.4 Implementation of OpenMP data-sharing directives and clauses

The last problem relates to the data-sharing requirements for the implementation of OpenMP data-sharing directives and clauses. For both data-sharing clauses and data-sharing directives, including the `shared`, `threadprivate`, `private`, `firstprivate`, `lastprivate`, `copyin`, `copyprivate` and `reduction`, the solution consists in declaring and using auxiliary local variables in each thread and inserting the necessary instructions to initialize these variables and/or update the value of original variables afterward. The `reduction` case is a little bit more complex and requires additional processing. A possible solution consists of a combination of the partial values into the checkpoints on slave nodes and a re-collection them on the master node to arise

the reduction value.

The translation of an OpenMP program containing data-sharing directives and clauses is divided into three steps when using CAPE: (1) directives and clauses containing the same variables are merged; (2) directives and clauses are replaced by a set of equivalent instructions; (3) CAPE templates for OpenMP constructs are applied to translate the program into base language form.

5.4.1 Merging directives and clauses

There are some cases for which a variable is contained in many directives and/or clauses. These cases may be translate into a single form. For example, in the following piece of code:

```
# pragma omp threadprivate ( x )
# pragma omp parallel for copyin ( x )
for ( A ; B ; C )
    D ;
```

the directive in the first line can be deleted and merged with the `copyin( x )` clause of the second line. This rule can also be applied to merge this directive with the `copyprivate` clause.

Note that OpenMP allows combinations of a few directives and clauses, like `threadprivate` and `reduction` directives. Table 5.2 shows the possible combination between OpenMP directives and clauses.

	shared	threadprivate	private	firstprivate	lastprivate	reduction	copyin	copyprivate
shared								
threadprivate							✓	✓
private								
firstprivate					✓			
lastprivate				✓				
reduction								
copyin		✓						
copyprivate		✓						

Table 5.2: Possible combinations for OpenMP directives and clauses.

5.4.2 General template

Figure 5.5 presents the general template to translate `parallel for` loops.

```
# pragma omp parallel for clause_U
    for ( A ; B ; C )
        D ;
```

↓ automatically translated into ↓

```
before_block ;
{
    enter_block ;
    # pragma omp parallel for
    for ( A ; B ; C )
    {
        D ;
        exit_block ;
    }
}
after_block ;
```

Figure 5.5: Template for data-sharing primitives and clauses in `for` loops.

At initialization, all translated blocks (`before_block`, `enter_block`, `exit_block` and `after_block` are empty. Then, the template is iteratively applied for each directive or clause by merging the translated parts of the directive or clause at each step.

5.4.3 Translation details

5.4.3.1 Summary of OpenMP data-sharing directive and clauses

Below is the list of OpenMP data-sharing directive and clauses with a brief description for each one. Their full descriptions can be referred in section 2.9 of [1].

1. `threadprivate` Directive

Syntax

```
# pragma omp threadprivate(list) new-line
```

Summary description

The `threadprivate` directive specifies that variables are replicated, with each thread having its own copy.

Each copy of a threadprivate variable is initialized once, in the manner specified by the program, but at an unspecified point in the program prior to the first reference to that copy. The storage of all copies of a threadprivate variable is freed according to how static variables are handled in the base language, but at an unspecified point in the program.

2. `default` clause

Syntax

```
default(shared | none)
```

Summary description

The `default` clause explicitly determines the data-sharing attributes of variables that are referenced in a parallel or task construct and would otherwise be implicitly determined.

The `default(shared)` clause causes all variables referenced in the construct that have implicitly determined data-sharing attributes to be shared.

The `default(none)` clause requires that each variable that is referenced in the construct, and that does not have a predetermined data-sharing attribute, must have its data-sharing attribute explicitly determined by being listed in a data-sharing attribute clause.

3. `private` clause

Syntax

```
private (list)
```

Summary description

The `private` clause declares one or more list items to be private to a task.

5.4.3.2 `firstprivate` clause

Syntax

```
firstprivate (list)
```

Summary description

The `firstprivate` clause provides a superset of the functionality provided by the `private` clause.

The `firstprivate` clause declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

4. `lastprivate` clause

Syntax

`lastprivate (`*list*`)`

Summary description

The `lastprivate` clause declares one or more list items to be private to an implicit task, and causes the corresponding original list item to be updated after the end of the region.

5. `reduction` clause

Syntax

`reduction (`*operator:list*`)`

Summary description

The `reduction` clause specifies an operator and one or more list items. For each list item, a private copy is created in each implicit task, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator.

6. `copyin` clause

Syntax

`copyin (`*list*`)`

Summary description

The `copyin` clause provides a mechanism to copy the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the parallel region.

The copy is done after the team is formed and prior to the start of execution of the associated structured block.

7. `copyprivate` clause

Syntax

`copyprivate (`*list*`)`

Summary description

The `copyprivate` clause provides a mechanism to use a private variable to broadcast a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the parallel region.

To avoid race conditions, concurrent reads or updates of the list item must be synchronized with the update of the list item that occurs as a result of the `copyprivate` clause.

The effect of the `copyprivate` clause on the specified list items occurs after the execution of the structured block associated with the `single` construct, and before any of the threads in the team have left the barrier at the end of the construct.

5.4.3.3 Dedicated translated blocks

Table 5.3 shows the translated blocks for all OpenMP data-sharing directives and clauses. The cases of `copyprivate` and `reduction` clauses are not completely provided in this table and are presented in the next sections. At present, along with an assumption that all variables are shared by default, CAPE does not consider the option `none` for the case of the `default` clause. Thus, this clause is not considered in the table. One can easily verify that translations using translated blocks in the tables match OpenMP requirements for data sharing directives and clauses presented above.

Table 5.3: Transformed blocks for OpenMP directives and clauses.

	before_block	enter_block	exit_block	after_block
shared (x)				
threadprivate (x)	typeof (x) __x__ ;	typeof (__x__) x ;		
private (x)	typeof (x) __x__ ;	typeof (__x__) x ;		
firstprivate (x)	typeof (x) __x__ ; __x__ = x;	typeof (__x__) x ; x = __x__		
lastprivate (x)	typeof (x) __x__ ;	typeof (__x__) x ;	if (thread_num == (num_threads - 1)) __x__ = x ;	x = __x__ ;
reduction (x)	typeof (x) __x__ ;	typeof (__x__) x; init_value (x) ;	send (x, master) ;	*continue in subsec. 5.4.3.5*

copyin (x)	typeof (x) __x__ ;
	if (master ( ))
	__x__ = x ;
		typeof (__x__) x ;
		if (master ( )) {
		x = __x__ ;
		broadcast (x) ; }
		else {
		receive (x) ;
		inject (x) ; }		

5.4.3.4 Template for the `copyprivate` clause

The `copyprivate` clause cannot be inserted in `parallel` regions but only in `single` regions. The current version of CAPE executes sequential regions in all threads, so that at present it cannot implement this clause. However, this is scheduled in the next version, thanks to the ability to run sequential regions on a single thread as presented in the Sec. 4.4.4. Thus, template in Fig. 5.6 will be applied where the original `single` construct is separated into two parts: one executed on the master node and the other on slave nodes. The first part consists of a `single` construct without the `copyprivate` clause. As presented Sec.4.4.4, this construct executes on the master node. Thus, the `broadcast ( )` at the end of this construct sends the value of the associated variable to all slave nodes. The part executed on slave nodes receives the broadcast value and inject it into process memories.

```
# pragma omp single copyprivate ( x )
    D ;
```

↓ automatically translated into ↓

```
# pragma omp single
{
    D ;
    broadcast ( x ) ;
}
if ( !master ( ) )
{
    receive ( x ) ;
    inject ( x ) ;
}
```

Figure 5.6: Template to translate the `copyprivate` clause.

5.4.3.5 Special processing for the `reduction` clause

`reduction` is the most complex data-sharing clause as it requires to accumulate in a variable on the master node the different values from all slave nodes. For the case of CAPE, two solutions are proposed: the first one consists in integrating the value of the variable into the checkpoint of all slave nodes and then extract these values on the master node; the second one consists in using separate functions to send and receive these values on the slave nodes and the master node respectively. For the second solution, beside the translation using the translated blocks from table 5.3, a modification of the template for parallel constructs is also required. Figure 5.7 presents the modified template for the `parallel for` loop (see Fig. 4.4 and Fig. 5.1) to integrate the translation of the `reduction` clause.

```
0   init ( x ) ;
1   if ( master ( ) ) {
...
8         stop ( ) ;
8a        init ( update_list ) ;
9         wait_for ( after ) ;
9a        merge ( update_list, after ) ;
9b        receive_reduction ( x ) ;
9c        merge ( x, after ) ;
...
15  } else {
...
22        send ( after_i , master ) ;
22a       send ( x , master ) ;
```

Figure 5.7: Modified prototype of CAPE to implement the `reduction` clause.

In this prototype, new function `init ( x )` (line 0) which executes on all nodes, aims at initializing the reduction variable x. Each slave node, after sending its execution result in form of a checkpoint (line 22), sends its own value of variable x to the master node (line 22a). On the master node, after receiving execution results from slave nodes, new function `receive_reduction ( x )` (line 9b) receives different values of x from all slave nodes and accumulates them into x. Then, the reduction variable is added into the final checkpoint (line 9c) before this checkpoint is injected into process memories of the nodes. Therefore, the reduction variable is accumulated and updated on all nodes. Rules to initialize and accumulate reduction variables is presented in Sec. 2.9.3.6 of [1];

5.5 Performance evaluation

At present, the integration of the UHLRC model and the processing of data-sharing directives and clauses into the CAPE implementation is being investigated. However, from the description above, it appears that this integration does not significantly decrease the global performance as the reasons below.

In general, the UHLRC model does not significantly change the current memory model of CAPE. The unique additional fact is the mechanism used to implement the `flush` directive. Whether an auxiliary task is added to the monitor, there are not many changes in its execution. On slave nodes, the mechanism to implement this task is the same as the main task for the incremental checkpointer. In fact, while running with the role of an incremental checkpointer, the monitor has to keep on listening for signals from the checkpointed program. Thus, the work consists in including the new case to process the added signal for the `flush` request from the

program. Furthermore, the master node, when executing parallel regions, aims at distributing the jobs to slave nodes and then waiting for results. This means that typically, in the parallel regions, the node is idle almost all the time. As a result, this new task only requires the node to listen and process `flush` requests from slave nodes and does not affect their respective time.

The advantage of the HLRC model to implement the shared-memory of OpenMP has been approved in [7]. With the UHLRC model, the number of comparisons — the main operator to implement the `flush` directive — decreases. As a result, it should theoretically provide better performance.

As presented in Sec. 5.4, the processing of data-sharing directives and clauses in most cases is performed at compile time. The execution of additional codes does not consume a lot of time either as there are only very few instructions executed to create local variables and to assign their initial values. The special cases for both `reduction` and `copyprivate` directives require the transaction of a small amount of data between nodes which does not affect the global performance.

5.6 Conclusion

This chapter presented the solutions for two aspects of OpenMP data-sharing problems: implementing a shared-memory model and implementing data-sharing directives and clauses.

For the implementation a shared-memory model, UHLRC has been presented. This new distributed-shared memory model is based on the HLRC model. Algorithms for implementing the `flush` operations on this memory model are also designed. Within the improvements that reduces the number of comparison operations — the most important operation to guarantee the memory consistency — this new model promises good performance.

For the implementation of data-sharing directives and clauses, a global template and detail parts for all directives and clauses have been presented. This implementation completely performs all the OpenMP shared directives and clauses without significantly affectation to the global performance.

Chapter 6

Conclusion and Future Work

Contents

OpenMP is a simple and strong API for shared-memory parallel programming. Many attempts have tried to port it on distributed-memory systems. However, they all meet difficulties to provide a fully-compliant and high-performance solution. CAPE is an approach that uses checkpoints to automatically distribute parallel sections of OpenMP programs to distributed machines, then automatically collect results from these machines to the original machine. Along with this important characteristic, CAPE promises to become a fully-compliant solution of OpenMP on distributed-memory systems.

CAPE-1 that refers to CAPE using complete checkpoints, has proved the feasibility of the approach but the use of complete checkpoints as the base tool strongly decreased the global performance. Furthermore, CAPE-1 is only applicable for problems matching the Bernstein's conditions, i.e. OpenMP data-sharing problems are not completely solved.

Our works in this thesis have successfully improved the performance of CAPE and overcome its restriction on data-sharing aspects.

6.1 Principle contributions

1. In chapter 3, we proposed DICKPT that stands for *Discontinuous Incremental Checkpointing*, a new checkpointing technique based on the incremental checkpointing technique. Thanks to the new ability to take independent incremental checkpoints on discontinuous sections of a program, the performance of checkpointing, in both aspects of execution time and checkpoint size, is significantly reduced in some cases. This is very useful for CAPE to take checkpoints only on necessary parts of transformed OpenMP programs, that help to increase the global performance. Exploits the special structure of incremental checkpoints, some methods to compress checkpoints also proposed in this chapter.

2. In chapter 4, we presented CAPE-2, a new version of CAPE that uses incremental checkpoints. Exploits the new abilities of DICKPT, a new execution model and new prototypes to transform OpenMP work-sharing constructs have been proposed. In this version, the drawbacks of CAPE-1 that affect

to the global performance have been overcome: all the amount of transfered data on network, the number of operations to extract execution results, the amount of memory space for temporary data, have been significantly reduced. Furthermore, thanks to the ability to directly inject updated data into process space, the requirement to restart process execution after integrating updated data is removed. All of them leads to a strong improvement of performance in CAPE-2, while comparing with CAPE-1. This was clearly showed in an experiment at the last part of this chapter, where performance of CAPE-2 is quite similar with the one of optimized MPI programs and speedup of CAPE-2 is quite linear with the number of used machines.

3. In the last contribution of the thesis, presented in chapter 5, we proposed the methods to satisfy the data-sharing requirements of OpenMP. Both the two aspects related to OpenMP data-sharing problems were presented and solved: 1) the implementation of OpenMP relaxed shared memory model and 2) the implementation of OpenMP directives and clauses.

   In the first part of this chapter, we proposed UHLRC that stands for Updated Home-Based Relaxed Consistency memory model. This model is based on the HLRC model and has two important changes. The first one is the use of `create` primitive of DICKPT checkpointer instead of the use of `diff` function to compute differences between the current content of process memory and the one in the buffer. The second one related to the replacement home pages in the master thread by a list of modified memory items. In one side, this reduces the number of comparison operations in `flush` function that is used to guarantee the coherence between the memories of the master and slave threads. In the other side, thanks to the location of the list of modified regions in memory spaces of CAPE's monitors, the amount of interprocess data transfer is reduced and that in turn helps to increase the global performance. Algorithms to implement the UHLRC model on CAPE also were presented. Based on the UHLRC model, the second part of this chapter showed a list of OpenMP data-sharing rules and the methods to satisfy these rules in CAPE.

   The third part of this chapter presented the implementations of OpenMP directives and clauses in CAPE. Most directives and clauses are implemented by use of transformation's prototypes. Some other ones are implemented by special algorithms.

   Although at present, we have not experiments but we believe that the integration into CAPE the new memory model and the data-sharing abilities does not significantly affect the global performance. This is due to the two most important arguments. The first one is the appropriation of the new memory model to the previous execution model of CAPE, that leads to only some light changes in CAPE's prototypes have to be added to implement the new memory model. The second one related to the implementation of OpenMP directives and clauses by some simple instructions that do not consume many

of execution time and memory space. Detailed arguments were presented in the last part of this chapter.

While comparing with initial purposes of the thesis, we can say that it has been completely finished. The performance of CAPE has been significantly increased in the new version. All data-sharing problems have been analyzed and solved by the use of a new memory model and the special prototypes and algorithms.

6.2 Future work

In the near future, we would like to develop new CAPE's components in which the UHLRC and all OpenMP data-sharing directives and clauses are completely implemented. Another planed work due to carry more experiments on execution CAPE with different problems and compare it with other OpenMP implementations. This can provide a better evaluation and thus better prove the advantages of CAPE as a fully-compliant and high-performance OpenMP implementation on distributed-memory systems.

Bibliography

[1] *OpenMP specification 3.1.* OpenMP Architecture Review Board. 2011.

[2] Blaise Barney. *Introduction to Parallel Computing.* Available at: *https://computing.llnl.gov/tutorials/parallel_comp/#Abstract.*

[3] http://www.mpi-forum.org/

[4] http://www.top500.org/

[5] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, R. Badrinath and Louis Rilling. *Kerrighed: A Single System Image Cluster Operating System for High Performance Computing.* Euro-Par 2003 Parallel Processing, Klagenfurt, Austria, LNCS 2790, pp. 1291–1294, August 2003.

[6] Sven Karlsson, Sung-Woo Lee, Mats Brorsson, Sahni Sartaj, Viktor K. Prasanna and Shukla Uday. *A fully compliant OpenMP implementation on software distributed shared memory.* Proceedings of the International Conference on High Performance Computing, Bangalore, India, LNCS 2552, pp. 195–206, December 2002.

[7] Jie Tao, Wolfgang Karl, Carsten Trinitis. *Implementing an OpenMP Execution Environment on InfiniBand Clusters.* In proceeding of the First International Workshop on OpenMP (IWOMP 2005). Eugene, Oregon, June 2005.

[8] Beniamino Di Martino, Dieter Kranzlmüller and Jack Dongarra *Implementing OpenMP for clusters on top of MPI.* Proceedings of 12th European PVM/MPI Users' Group Meeting Sorrento, LNCS, Volume 3666/2005, 148–155, DOI: 10.1007/11557265_22, Italy, September 2005.

[9] Ayon Basumallik and Rudolf Eigenmann. *Towards automatic translation of OpenMP to MPI.* Proceedings of the 19th annual international conference on Supercomputing, Cambridge, MA, pp. 189–198, 2005.

[10] Lei Huang and Barbara Chapman and Zhenying Liu. *Towards a more efficient implementation of OpenMP for clusters via translation to global arrays.* Journal of Parallel Computing, 31(10–12):1114–1139, October–December 2005.

[11] Jay P. Hoeflinger. *Extending OpenMP* to Clusters - White paper.* Available at: http://140.110.240.196/grid/raw-attachment/wiki/Osaka/ Intel_Extend_OpenMP_Cluster.pdf

[12] Éric Renault. *Distributed Implementation of OpenMP Based on Checkpointing Aided Parallel Execution.* International Workshop on OpenMP (IWOMP), Beijing, China, LNCS 4935, pp. 183–193, June 2007.

[13] Éric Renault. *Parallelization of For Loops Using Checkpointing Techniques.* Proceedings of the 2005 International Conference on Parallel Processing Workshops, Oslo, Norway, pp. 313–319, June 2005.

[14] Laura Mereuta and Éric Renault. *Checkpointing Aided Parallel Execution Model and Analysis.* High Performance Computation Conference (HPCC), Houston, TX, LNCS 4782, pp. 707–717, September 2007.

[15] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. *Memory consistency and event ordering in scalable shared-memory multiprocessors.* In Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 15–26, Seattle, Washington, May 1990.

[16] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel *Lazy Release Consistency for Software Distributed Shared Memory.* In Proceeding of the 19th Annual International Symposium on Computer Architecture. Queensland, Australia, May 1992.

[17] Yuanyuan Zhou, Liviu Iftode and Kai Li. *Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems.* Proceedings of the second USENIX symposium on Operating systems design and implementation (OSDI). New York, NY, USA 1996.

[18] Blaise Barney. *OpenMP.* Available at https://computing.llnl.gov/tutorials/openMP

[19] www.adapteva.com

[20] http://www.kerrighed.org

[21] Jens Breitbart. *Analysis of a memory bandwidth limited scenario for NUMA and GPU systems.* In Proceeding of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), Kassel, Germany May, 2011.

[22] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto and T. Takahashi. *Dynamic Home Node Reallocation on Software Distributed Shared Memory.* Proceeding of tje HPC Asia 2000, Beijing, China, May 2000, pp. 158—163.

[23] Mitsuhisa Sato , Hiroshi Harada, Atsushi Hasegawa and Yutaka Ishikaw. *Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system.* Journal Scientific Programming, Volume 9 Issue 2,3, August 2001.

[24] Antonio J. Dorta, Jesús. A. González, Casiano Rodríguez, Francisco de Sande. *llc: A parallel skeletal language.* Parallel Processing Letters 13(3) (2003) 437–448.

[25] Seung-Jai Min, Ayon Basumallik, Rudolf Eigenmann. *Optimizing OpenMP Programs on Software Distributed Shared.* International Journal of Parallel Programming, 31(3):225–249, June 2003.

[26] Paul Havlak and Ken Kennedy. *An implementation of interprocedural bounded regular section analysis.* IEEE Transactions on Parallel and Distributed Systems, 2(3):350–360, 1991.

[27] Sang-Ik Lee, Troy A. Johnson and Rudolf Eigenmann. *Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation.* In Proceeding of the Workshop on Languages and Compilers for Parallel Computing(LCPC'03), pages 539–553. (Springer Verlag Lecture Notes in Computer Science), Oct. 2003.

[28] Jaroslaw Nieplocha, Robert J. Harrison and Richard J. Littlefield *Global Arrays: A non-uniform memory access programming model for high-performance computers.* The Journal of Supercomputing, 10:197-220, 1996.

[29] Lei Huang, Barbara Chapman, Zhenying Liu and Ricky Kendall. *Efficient Translation of OpenMP to Distributed Memory.* Lecture Notes in Computer Science, 2004, Volume 3038/2004, 408-413.

[30] Lei Huang, Barbara Chapman and Ricky Kendall. *OpenMP for Clusters.* In the Fifth European Workshop on OpenMP, EWOMP'03, Aachen, Germany, 2003.

[31] *TreadMarks: shared memory computing on networks of workstations.* Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu and Willy Zwaenepoel. The Journal Computer, 29(2): 18–28, Feb 1996.

[32] Y. Charlie Hu, Honghui Lu, Alan L. Cox and Willy Zwaenepoel *OpenMP for Networks of SMPs.* The Journal of Parallel Distributed Computing, 60: 1512–1530, 2000.

[33] Jame S. Plank. *An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance.* Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, Juilly 1997.

[34] Kalaiselvi S. and Rajaraman. *A survey of checkpointing algorithms for parallel and distributed computers.* In Sadhana (Academy Proceedings in Engineering Sciences), 25 (5). pp.489-510, October 2000.

[35] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung and Chandra Kintala. *Checkpointing and its applications.* 25th Annual International Symposium on Fault-Tolerant Computing, (FTCS-25), pp.22-31, June 1995.

[36] John Mehnert-Spahn, Eugen Feller and Michael Schoettner. *Incremental checkpointing for grids.* In Linux Symposium, 2009.

[37] *Unicos Operating System (Unix).* Available at: http://www.operating-system.org/betriebssystem/_english/bs-unicos.htm .

[38] Charles R. Landau. *The checkpoint mechanism in KeyKos.* In the Proceedings of the Second International Workshop on Object Orientation in Operating Systems, September 1992.

[39] Mark Russinovich and Zary Segall. *Fault-tolerance for off-the-shelf applications and hardware.* In 25th International Symposium on Fault-Tolerant Computing, pp. 67-71, Pasadena, CA, June 1995.

[40] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. *The sprite network operating system.* IEEE Computer, 21(2):23-36, February 1988.

[41] Amnon Barak. *Scalable cluster computing with mosix for linux.* In Proceedings of Linux Expo, pp.95–100, 1999.

[42] Christine Morin , Pascal Gallard , Renaud Lottiaux and Geoffroy Vallée. *Towards an efficient single system image cluster operating system.* Proceeding of the Fifth International Conference on Algorithms and Architectures for Parallel Processing, 2002.

[43] Oleksandr O. Sudakov, Ievgenii S. Meshcheriakov and Yuriy V. Boyko. *CHPOX: Transparent Checkpointing System for Linux Clusters.* In the 4th IEEE workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS). 2007.

[44] Ramamurthy Badrinath, Christine Morin and Geoffroy Vallée. *Checkpointing and Recovery of Shared Memory Parallel Applications in a Cluster.* In Proceeding of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003). Tokyo, Japan, May 2003.

[45] Robert H.B. Netzer and Mark H. Weaver. *Optimal tracing and incremental reexecution for debugging long-running programs.* In Proceeding of the ACM SIGPLAN 1994 (PLDI'94) on Programming language design and implementation, pp. 313–325, Orlando, FL, June 1994.

[46] ckpt home page. http://pages.cs.wisc.edu/ zandy/ckpt/

[47] Miron Livny and Mike Litzkow. *Making Workstations a Friendly Environment for Batch Jobs.* In Proceedings of Third Workshop on Workstation Operating Systems. April 1992.

[48] Arash Baratloo, Partha Dasgupta and Zvi M. Kedem. *Calypso: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms.* In 4th IEEE International Symposium on High Performance Distributed Computing, August 1995.

[49] *Diskless checkpointing.* James S. Plank, Kai Li, Member and Michael A. Puening. Journal of Parallel and Distributed Systems, 9: 971–986, October 1998.

[50] Michael Litzkow, Todd Tannenbaum, Jim Basney and Miron Livny. *Checkpoint and migration of unix processes in the condor distributed processing system.* Technical report 1346, University of Wisconsin-Madison, 1997.

[51] S. I. Feldman and C. B. Brown. *Igor: A system for program debugging via reversible execution.* In the ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging, 24(1):112–123, January 1989.

[52] Elmootazbellah N. Elnozahy, and Willy Zwaenepoel. *Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit.* In the IEEE Transactions on Computers Special Issue on Fault-Tolerant Computing, 41(5), May 1992.

[53] Georg Stellner. *CoCheck: Checkpointing and process migration for MPI.* In Processing of the 10th International Parallel Symposium, pp. 526–531, April 1996.

[54] Manuel Costa, Paulo Guedes, Manuel Sequeira, Nuno Neves and Miguel Castro. *Lightweight logging for lazy release consistent distributed shared memory.* In Proceedings of the second USENIX symposium on Operating systems design and implementation, October 1996.

[55] James S. Plank and Kai Li. *Ickp - a consistent checkpointer for multicomputers.* In the Journal IEEE Parallel & Distributed Technology, 2(2):62–67, Summer 1994.

[56] Yinqun Chen, James S. Plank and Kai Li. *CLIP: A checkpointing tool for message-passing parallel programs.* In Proceeding of the SC97: High Performance Networking and Computing, San Jose, November 1997.

[57] Steven Osman, Dinesh Subhraveti, Gong Su and Jason Nieh. *The design and implementation of zap: A system for migrating computing environments.* In Proceedings of the 5th Operating Systems Design and Implementation (OSDI 2002), Boston, MA, December 2002.

[58] Erik Hendriks. *BProc: The Beowulf Distributed Process Space.* Technical report, Advanced Computing Laboratory, Los Alamos National Laboratory, 2001.

[59] Jason Ansel, Kapil Arya and Gene Cooperman. *DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop.* In Proceeding of the 32rd IEEE International on Parallel and Distributed Processing Symposium (IPDPS09), Rome, Italy, May 2009.

[60] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri and A. Selikhov. *MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes.* In ACM/IEEE 2002 Conference on Supercomputing. IEEE Press, 2002.

[61] Thomas Herault, Pierre Lemarinier, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault-tolerant MPI. In Proceedings of International Symposium on High Performance Computing and Networking (SC2006), 2006.

[62] Yennun Huang, Chandra Kintala and Yi-Min Wang. *Software tools and libraries for fault toleranc.* In IEEE Technical Committee on Operating Systems and Application Environments, 7(4):5–9, Winter 1995.

[63] Adam Beguelin, Erik Seligman and Peter Stephan. *Application Level Fault Tolerance in Heterogeneous. Networks of Workstations.* Journal of Parallel and Distributed Computing, 43(2): 147–155, June 1997.

[64] Luis M. Silva, Simon Chapple, João G. Silva and Lyndon Clarke. *Portable checkpointing and recovery.* In Proceeding of the Fourth IEEE International Symposium on High Performance Distributed Computing, August 1995.

[65] David Cummings and Leon Alkalaj. Checkpoint/rollback in a distributed system using coarse-grained dataflow. In Proceeding of the Twenty-Fourth International Symposium on Fault-Tolerant Computing, June, 1994.

[66] Gabriel Antoniu, Luc Bougé and Raymond Namyst. *An efficient and transparent thread migration scheme in the PM2 runtime system.* In Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, 1999.

[67] Michael L. Powell and Barton P. Miller. *Process Migration in DEMOS/MP.* In Proceedings of the Ninth ACM Symposium on Operating Systems Principles, 1983.

[68] http://hpc.pnl.gov/sft/tick.html

[69] Sangho Yi, Junyoung Heo, Yookun Cho, Jiman Hong, Jongmoo Choi and Gwangil Jeon. Ickpt: An Efficient Incremental Checkpointing Using Page Writing Fault - Focusing on the Implementation in Linux Kernel. In Proceedings of the ISCA 19th International Conference on Computers and Their Applications (CATA04), Seattle, WA, pp. 209-212, March 2004.

[70] David P. Anderson. *BOINC: A System for Public-Resource Computing and Storage.* In Proceedings of 5th IEEE/ACM International Workshop on Grid Computing, Pittsburg, PA, pp. 4–10, November 2004.

[71] *Amazon Elastic Compute Cloud (EC2)*. http://aws.amazon.com/ec2/.

[72] *Nimbus Home Page*. http://www.nimbusproject.org

[73] Paul Marshall, Henry Tufo, Kate Keahey, David LaBissoniere and Matthew Woitaszek. *Architecting a Large-Scale Elastic Environment: Recontextualization and Adaptive Cloud Services for Scientific Computing*. In Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT). Rome, Italy. July 2012

[74] *OpenNebula Home Page*. http://opennebula.org/

[75] *KOALA Home Page*. http://code.google.com/p/koalacloud/

[76] Christian Baun, Marcel Kunze, Viktor Mauch. *The KOALA Cloud Manager Cloud Service Management the Easy Way.* In Proceeding of the 2011 International Conference on Cloud Computing (CLOUD 2011), Washington DC, July 2011.

[77] Daniel P. Bovet and Marco Cesati. *Understading the Linux Kernel.* 3rd version. O'Reilly 2006.

[78] Sandeep Grover. *Buffer Overflow Attacks and Their Countermeasures.* http://www.linuxjournal.com/article/6701

[79] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, Fabrizio Petrini and Kei Davis. *Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers.* In Proceeding of the the 2005 ACM/IEEE conference on Supercomputing. Washington DC, 2005.

[80] Leslie Lamport. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. In the IEEE Transactions on Computers, 28(9):241-248, 1979.

[81] L. Iftode and J. P. Singh. *Shared Virtual Memory: Progress and Challenges*. In Proceeding of the IEEE, Special Issue on Distributed Shared Memory, volume 87, pages 498-507, 1999.

[82] Yong-Kim Chong and Kai Hwang. *Evaluation of relaxed memory consistency models for multithreaded multiprocessors*. In Proceeding of the 1994 International Conference on Parallel and Distributed Systems, p474-480, Hsinchu, Taiwan, December 1994.

[83] Alan L. Cox, Eyal de Lara, Charlie Hu and Willy Zwaenepoel. *A Performance Comparison of Homeless and Home-based Lazy Release Consistency Protocols in Software Shared Memory.* In Procceding of the Fifth International Symposium On High-Performance Computer Architecture, Janvier 1999.