Checkpointing Aided Parallel Execution : modèle d'exécution et optimisation

Van Long TRAN[1,2], Éric RENAULT[1]

[1]SAMOVAR, Télécom SudParis, CNRS, Université Paris-Saclay,
9 rue Charles Fourier - 91011 Évry Cedex, France
[2] Hue Industrial College, Vietnam

Introduction

Afin de minimiser les difficultés liées au développement d'applications parallèles, un outil de haut niveau doit être aussi facile d'utilisation que possible.
Deux outils sont généralement utilisés :

- OpenMP est très simple d'utilisation. Il nécessite d'identifier les parties du code à exécuter en parallèle et fonctionne sur système à mémoire partagée.
- MPI est dédié aux systèmes à mémoire distribuée. Il est plus difficile d'utilisation.

Des efforts ont été faits pour fournir OpenMP sur architectures à mémoire distribuée. En dehors de CAPE, aucune solution n'est véritablement conforme aux spécifications.

	Entièrement compatible avec OpenMP	Haute performance
SSI (Single System Image)	**Oui**	Non
SCASH	Non	**Oui**
Utiliser le modèle RC	Non	**Oui**
Traduire en MPI	Non	**Oui**
Global Array	Non	**Oui**
Cluster OpenMP	Non	**Oui**
CAPE	**Oui**	**Oui**

Figure 1 : OpenMP sur système de mémoire distribuée.

Principe de CAPE

Pour traduire les programmes OpenMP, CAPE utilise des *templates*. Après la traduction source à source du code OpenMP, le code est compilé à l'aide de gcc. La Fig. 2 montre les différentes étapes de compilation pour les programmes OpenMP écrits en C ou C++.

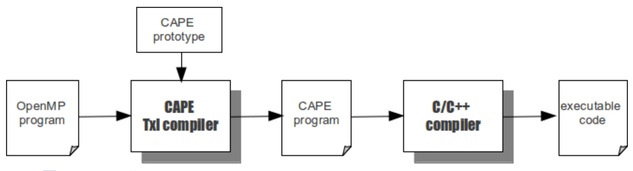

Figure 2 : Traduction de programmes OpenMP avec CAPE.

La Fig. 3 présente le nouveau modèle d'exécution de CAPE (exemple basé sur 3 nœuds). Tout d'abord, le programme démarre les fils d'exécution. Quand le programme atteint la section parallèle, le processus maître distribue les *jobs* aux processus esclaves en utilisation DICKPT. Les nœuds esclaves reçoivent les points de reprise, injecte leur contenu dans leur espace d'adressage et exécute le *job*. Le résultat est retourné au nœud maître en utilisant à nouveau DICKPT. À la fin de la section parallèle, le nœud maître envoie le point de reprise à tous les nœuds esclaves pour synchroniser la mémoire du programme.

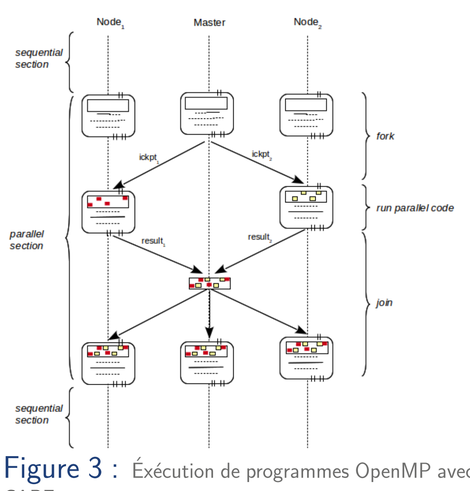

Figure 3 : Éxécution de programmes OpenMP avec CAPE.

```
# 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, )
19   stop ( )
20   send ( after,, master )
21   if ( ! last parallel ( ) )
22           receive ( final )
23           inject ( final )
24   else
25           exit
```

Figure 4 : Traduction de programmes OpenMP avec CAPE.

CAPE traduit automatiquement les construction `parallel for` en un ensemble d'appels de fonction contenant les opérations fondamentales de CAPE. Le *template* pour les constructions `parallel for` est présenté en Fig. 4.

Remarques

Les bonnes performances de CAPE par rapport à celles de MPI, et le fait qu'il soit totalement conforme aux spécifications d'OpenMP, font de CAPE une bonne alternative pour porter OpenMP sur les architectures à mémoire distribuée.
À ce jour, CAPE n'est pas finalisé. Les développements en cours sont les suivants :

- le nœud maître peut devenir un goulot d'étranglement lorsqu'il attend les points de reprise des nœuds esclaves, pour les réunir et renvoyer le résultat aux autres nœuds (cf Fig. 3);
- après distribution des *jobs*, les nœuds esclaves exécutent chacun le leur alors que le nœud maître attend le retour des résultats, ceci constituant clairement un gâchis de ressources ;
- CAPE ne peut être utilisé que sur les programmes OpenMP respectant les conditions de Bernstein.

Optimisation des checkpoints

Un point de reprise peut être défini par un triplet *adresse*, *valeur* et *horodatage*, que l'on peut représenter sous la forme :

$$C = (R_t, \{(a, v)_t\})$$

où:

- t est la date, $t \in \{0, 1, 2, 3...N\}$;
- R_t contient l'ensemble des registres à l'instant t ;
- $S \equiv \{(a, v)_t\}$ est l'ensemble des couples *adresse-valeur* ayant été modifiés à l'instant t ;

Opérations sur les points de reprise :

- Combinaison de points de reprise, noté $\oplus$:

$$C = C' \oplus C'' = (R_{max(t',t'')}, S' \oplus S'')$$

- L'exclusion, notée $\ominus$:

$$C = C' \ominus C'' = (R_{max(t',t'')}, S' \ominus S'')$$

Conclusion et travaux futurs

Avec l'arithmétique sur les points de reprise et le nouveau modèle d'exécution proposé, les performances et les capacités de CAPE sont grandement améliorées. Ceci est mesuré à la fois par une approche théorique et les résultats expérimentaux.

Dans un futur proche, nous chercherons à comparer les performances de CAPE avec celles de MPI en s'appuyant sur les *benchmarks* NAS.

Le nouveau modèle d'exécution du CAPE

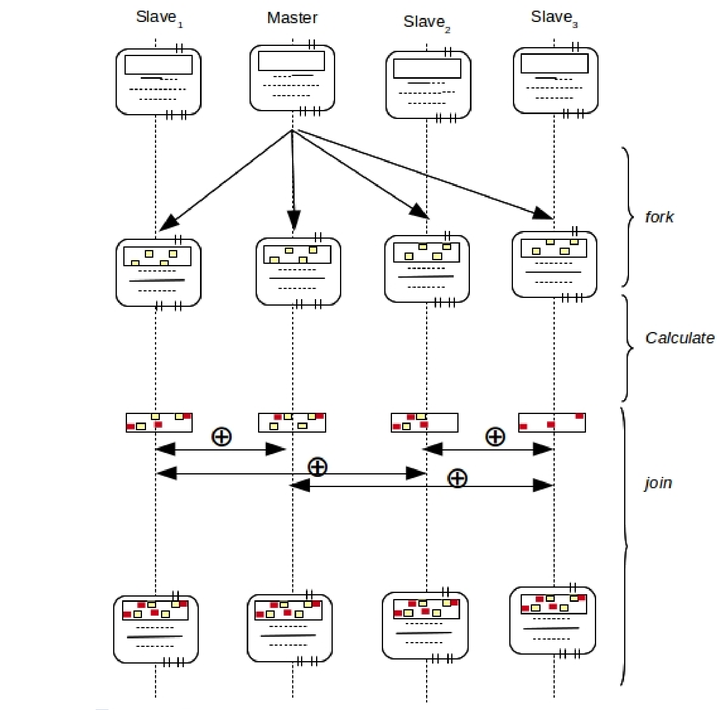

Figure 5 : Une nouvelle implémentation du modèle *fork-join* pour CAPE.

En utilisant notre arithmétique sur les points de reprise, nous avons réimplanté le modèle *fork-join* d'OpenMP. Ce nouveau modèle d'exécution est proposé en Fig. 5. Au début, le programme CAPE exécute et initialize tous les nœuds du système. Lorsqu'il atteint une section parallèle, le modèle *fork-join* est implanté de la façon suivante :

- phase *fork* : le nœud maître génère un point de reprise incrémental et le diffuse à tous les nœuds. Le point de reprise contient les données du programme ;
- phase de calcul : tous les nœuds requis (incluant le nœuds maître) exécute son *job*. Ensuite, un point de reprise différent est généré par chaque nœud ;
- phase *join* : les points de reprise incrémentaux sont réunis sur chaque nœud en utilisant l'algorithme *Recursive Doubling* (Cf Fig. 6) ;

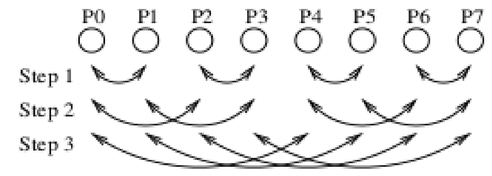

Figure 6 : Double récursivité pour *allreduce*.

Expérimentation

Les expérimentations ont consisté en un produit de matrices sur grappe. La taille des matrices varie de 1600×1600 à 9600×9600. Les grappes sont composées de 4 à 16 nœuds, chaque nœud intégrant un processeur Intel i3-2100 double cœus, 4 *threads* cadencé à 3,10 GHz et 2 Go de RAM. Les résultats sont présentés en Fig. 7 à 9.

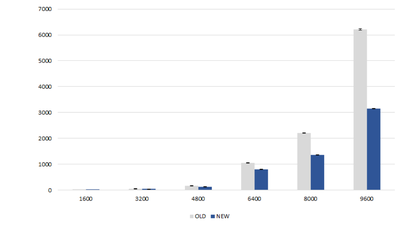

Figure 7 : Temps d'exécution total (en secondes) des deux modèles sur les *clusters* à 4 nœuds.

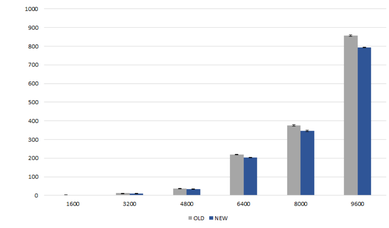

Figure 8 : Temps d'exécution total (en secondes) des deux modèles sur des *clusters* à 16 nœuds.

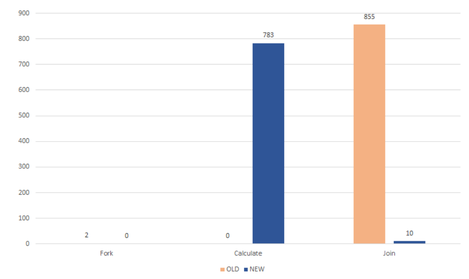

Figure 9 : Temps d'exécution (en secondes) pour *fork*, calcul et *join* pour les différents modèles d'exécution sur le noeud maître.