Thèse

présentée par

François Galilée

pour obtenir le grade de Docteur

de l'Institut National Polytechnique de Grenoble

(Arrêté ministériel du 30 mars 1992)

Spécialité : Informatique

Athapascan-1 : interprétation distribuée du flot de données d'un programme parallèle

Date de soutenance : 22 septembre 1999

Composition du jury :

Jean-Marc Geib, *rapporteur*
François Irigoin, *rapporteur*
Guy Mazaré, *président du jury*
Brigitte Plateau, *directeur de thèse*
Jean-Louis Roch, *responsable de thèse*

Thèse préparée au sein de
l'Unité Informatique et Distribution (ID-IMAG)

Table des matières

Partie II

Flot de données et contrôle de la mémoire

6

Flot de données et modèle de coût

Ce chapitre étudie l'apport de la modélisation des exécutions d'une application par un graphe, qu'il soit de précédence ou de flot de données, dans la définition d'un modèle de coût permettant de garantir les performances de toute exécution. Les principales sections de ce chapitre traitent les points suivants :

- Les **grandeurs caractéristiques** de l'application, grandeurs déterminées à partir d'une analyse théorique du graphe (section 6.2 page 116).
- Le **modèle de coût** qui permet de prédire, *a priori*, les performances de toute exécution (section 6.3 page 118).
- Les **garanties de performances** dans le cas d'Athapascan-1. Ces garanties sont liées à la politique d'ordonnancement utilisée (section 6.4 page 123).

6.1 Introduction

Comme vu au chapitre 2 page 21, la plupart des langages parallèles de « haut niveau » permettent de décrire le parallélisme d'une application à un degré bien supérieur à celui du nombre de processeurs. L'implantation du langage est alors responsable de l'exploitation du parallélisme exprimé sur la machine cible : l'efficacité de l'exécution reposera entièrement sur l'ordonnancement des tâches qui sera effectué par le système. Cette efficacité est mesurée en terme de durée d'exécution et de volume mémoire requis.

Afin de permettre une analyse formelle et théorique de l'exécution d'une application, toute exécution est modélisée par un graphe. C'est à partir de ce graphe, donc de cette modélisation, que les performances en temps et en mémoire de cette application seront évaluées dans le modèle de coût associé au langage. La constitution de ce graphe dépend des besoins du modèle : il peut être réduit à un graphe de précédence (le cas de Cilk par exemple) ou contenir une information sur les données accédées et devenir un graphe de flot de données (le cas d'Athapascan-1 ou de Jade par exemple).

Un ordonnancement glouton peut être effectué dynamiquement et à la volée et permet un exécution efficace en temps : les processeurs prennent, chaque fois qu'ils deviennent inactifs, une tâche à exécuter dans une liste de tâches prêtes. Ces techniques d'ordonnan-

cement exploitent le parallélisme de l'application à un degré permettant de maintenir au maximum les processeurs en activité. Ainsi sur un modèle de machine simple (processeurs identiques et absence de communication) un tel ordonnancement permet d'obtenir un temps d'exécution effectif à un facteur 2 de l'optimum. Cependant, ceci peut mener à une utilisation abusive de la mémoire : soit parce qu'un nombre trop important de tâches est maintenu à un instant donné par le système, soit parce que des tâches allouant de la mémoire sont exécutées avant d'autres tâches qui en libèrent. Il existe en effet, dans un cadre de graphe général, des applications parallèles pour lesquelles tout gain en temps, même minime, ne peut s'effectuer qu'au prix d'une consommation mémoire prohibitive [16], comme présenté section 6.3.2.1 page 119

Cependant, si l'on impose des restrictions sur le graphe d'exécution de l'application, il est possible de majorer simultanément la durée et la consommation mémoire de toute exécution. Par exemple, le langage Cilk impose un graphe de tâches série-parallèle et son implantation [51] garantit, en majorant à tout instant le nombre de *threads* en concurrence par p, une consommation en mémoire inférieure à pS_1, p étant le nombre de processeurs et S_1 l'espace mémoire requis pour une exécution séquentielle sur un processeur. De même l'implantation proposée de NESL dans [12] majore l'utilisation mémoire par $S_1 + p \log pT_\infty$, avec T_∞ la longueur en temps du plus long chemin du graphe. Avec une technique d'ordonnancement similaire le langage basé sur un graphe d'exécution série-parallèle proposé dans [83, 82] limite l'utilisation de la mémoire à $S_1 + pT_\infty$. La bibliothèque Athapascan-1 construit de manière dynamique un graphe orienté et sans cycle des accès aux données et aucune limitation n'est imposée sur ce graphe. Les performances de l'exécution dépendent de la politique d'ordonnancement choisie mais certaines garantissent de manière asymptotique des efficacités en mémoire de l'ordre de pS_1 ou $S_1 + O(pT_\infty + hC_\infty)$ (C_∞ représente la longueur en accès distants d'un plus long chemin du graphe).

Nous présentons tout d'abord les grandeurs caractéristiques associées à une exécution, puis la prédiction des performances en temps et en mémoire de ces exécutions à l'aide des modèles de coût associés aux langages de programmation. Nous présentons enfin le cas particulier d'Athapascan-1 et donnons deux politiques d'ordonnancement permettant de contrôler la consommation mémoire des exécutions tout en exploitant le parallèlisme de l'application (afin d'obtenir un gain en temps).

6.2 Grandeurs caractéristiques

Dans les langages offrant un modèle de coût, les performances de l'exécution peuvent être déduites du code de l'application. Ici, ces performances sont exprimées en fonction du graphe caractérisant l'instance de l'application et de la machine à l'aide des grandeurs suivantes :

– T_s, la durée d'exécution d'une implantation séquentielle de l'algorithme. Cette grandeur traduit le coût de la méthode de calcul utilisée par l'application : il ne contient aucun surcoût de parallélisme.

– T_1, la durée d'exécution du code parallèle de l'application sur un seul processeur. Cette grandeur représente le **travail de l'application**. Cette grandeur contient le coût de la méthode de calcul T_s et le surcoût introduit par la description du parallélisme. En particulier nous supposerons :

$$T_1 \geq T_s$$

De plus, cette valeur de T_1 est supposée indépendante de l'ordonnancement effectué des tâches[1].

– S_1, la consommation mémoire de l'exécution sur un seul processeur. L'exécution considérée est celle décrite dans la définition de T_1 et l'ordre d'exécution des instructions est celui calculé par l'ordonnanceur. Si cet ordre n'avait que peu d'influence sur la valeur de T_1, il est désormais primordial pour la définition de S_1.

– T_∞, la durée d'exécution théorique sur une infinité de processeurs : c'est à dire le meilleur temps d'exécution possible sur une machine. Cette grandeur est une borne inférieure pour la durée d'exécution sur toute machine parallèle et représente le maximum des durées d'exécution des ensembles d'instructions qui doivent être exécutées séquentiellement (suite à des contraintes de précédence). T_∞ représente la durée d'un plus long chemin d'exécution dans le graphe de tâches. Cette valeur ne contient pas le surcoût d'ordonnancement. Nécessairement :

$$T_\infty \leq T_1$$

– p, le nombre de processeurs physiques de la machine parallèle. Ces p processeurs sont considérés comme identiques et l'accès à la mémoire est uniforme : la machine considérée est de type *SMP (symmetric multiprocessors).*

– T_p, la durée d'exécution sur p processeurs. En particulier, comme T_1 et T_∞ sont des invariants :

$$\max\left(\frac{T_1}{p}, T_\infty\right) \leq T_p$$

– S_p, la consommation mémoire de l'exécution sur p processeurs.

– $\bar{p} = \frac{T_1}{T_\infty}$, le degré de parallélisme de l'application, ou accélération maximale possible puisque nécessairement $\frac{T_1}{T_p} \leq \frac{T_1}{T_\infty} = \bar{p}$. On définit également le degré de liberté du parallélisme, ou *parallel slackness* [103], comme étant le rapport $\frac{\bar{p}}{p}$.

– $a_p = \frac{T_1}{T_p}$, l'accélération de l'exécution. Nécessairement :

$$a_p \leq \min(p, \bar{p})$$

[1] Ce qui est faux en toute rigueur : même si l'application est entièrement déterministe, la durée d'exécution de certaines instructions peut dépendre du passé de l'exécution. C'est typiquement le cas des actions sur la mémoire (`new`, `delete`) dont la durée n'est en général pas constante mais amortie.

6.3 Modèle de coût

Le modèle de coût associé à un langage permet de déterminer *a priori* la performance de l'exécution effective d'une application, et ce à partir de la seule connaissance du graphe caractéristique de l'exécution. La performance d'une exécution est caractérisée par sa durée et le volume mémoire qui lui a été nécessaire.

Nous présentons tout d'abord une technique simple permettant d'atteindre une accélération linéaire quasi optimale en temps puis les problèmes liés à la consommation mémoire. Le but des langages est de permettre[2] une accélération linéaire en temps, $T_p = O\left(\frac{T_1}{p}\right)$, sans pour autant pénaliser l'exécution par une consommation incontrôlée de mémoire.

6.3.1 Performances en temps

L'efficacité en temps est facilement obtenue par des techniques simples d'ordonnancement en ligne permettant d'atteindre des efficacités quasi optimales. Le surcoût d'ordonnancement doit cependant être maîtrisé afin de ne pas constituer l'essentiel de la durée d'exécution.

6.3.1.1 Algorithmes gloutons

Les algorithmes gloutons d'ordonnancement permettent d'obtenir des temps d'exécution situées à un facteur 2 de l'optimum [63, 64, 17, 94]. Le principe de base de ces algorithmes est de maintenir au maximum les processeurs en activité : chaque fois qu'un processeur devient inactif, une tâche prête à être exécutée (s'il en existe) lui est donnée. Ces algorithmes sont tels qu'à chaque étape de l'exécution, si il existe au moins p tâches prêtes alors aucun processeur n'est inactif, sinon toutes les tâches prêtes sont en cours d'exécution. Le théorème suivant ne tient compte ni des communications ni du surcoût d'ordonnancement.

Théorème 3 *[16] Quel que soit le nombre p de processeurs de la machine, toute application de travail T_1 et de profondeur T_∞ ordonnancée par un algorithme d'ordonnancement glouton s'exécute en un temps T_p tel que $T_p \leq \frac{T_1}{p} + T_\infty$.*

Nous rappelons la preuve donnée dans [63, 72] car son schéma intervient dans la démonstration de la proposition 10 page 125.

Preuve. Soit t_1 une tâche qui s'est terminée à la date T_p, et soit d_1 la date du début de cette tâche. Nous nous intéressons à ce qui s'est passé *avant* la date d_1. Deux cas peuvent être distingués :

1. Soit aucun processeur n'a été inactif avant d_1.
2. Soit au contraire il existe une date $d < d_1$ à laquelle au moins un processeur était inactif. Soit d' la plus grande de ces dates. L'algorithme étant glouton, si à la date d' la tâche t_1 avait été prête alors elle aurait débuté son exécution. Il existe donc une

[2] Au minimum de manière asymptotique.

tâche t_2 en cours d'exécution à la date d' telle que $t_2 \prec_G t_1$. Soit d_2 la date de début d'exécution de cette tâche.

L'application récursive de ce schéma permet de construire une séquence $t_k \prec_G \cdots \prec_G t_2 \prec_G t_1$ de tâches telles qu'à tout instant de l'exécution, soit tous les processeurs sont actifs, soit un des processeurs est en train d'exécuter une des tâches de cette séquence. La durée de la première situation est majorée par $\frac{T_1}{p}$ et celle de la seconde par la durée d'un chemin critique du graphe, c'est-à-dire par T_∞. Ainsi $T_p \leq \frac{T_1}{p} + T_\infty$. □

Il est à noter que ce résultat ne tient pas compte du surcoût dû à l'implantation de l'algorithme permettant d'assurer le caractère glouton de l'ordonnancement.

6.3.1.2 Surcoût de l'ordonnancement

Le coût d'ordonnancement comprend le maintien de l'ensemble des tâches prêtes ainsi que l'attribution des tâches de cet ensemble aux processeurs.

Dans le cas du langage Cilk [14] pour des graphes de type série-parallèle, ce coût peut être majoré par $O(T_\infty)$ en utilisant un ordonnancement glouton basé sur une technique de vol de travail [16, 13].

Dans le cas de NESL, l'ordonnancement est effectué par étapes [12]. Le calcul du placement pour une étape est effectué [11] en un temps $\log p$. Le nombre d'étape est de l'ordre de $\frac{T_1}{p \log p} + T_\infty$, le coût de l'ordonnancement est donc majoré par $O\left(\frac{T_1}{p} + (\log p) T_\infty\right)$.

6.3.2 Performance en mémoire

L'obtention d'une efficacité linéaire en temps, c'est-à-dire $T_p = O\left(\frac{T_1}{p}\right)$ sur une machine à p processeurs, nécessite de maintenir actif les processeurs la majeure partie du temps. L'exécution en parallèle de tâches nécessite en général un espace mémoire plus important qu'une exécution séquentielle car, outre le fait qu'il faille maintenir les structures de données caractérisant les tâches, des allocations mémoire peuvent avoir lieu en concurrence (tandis que lors d'une exécution séquentielle chaque allocation aurait pu être suivie d'une libération). Nous montrons dans la suite de cette section, à travers l'étude d'un exemple, qu'il n'est pas possible dans un cas général d'obtenir une accélération linéaire en temps d'exécution tout en limitant l'espace mémoire nécessaire à cette exécution [16]. Cependant, des limitations sur le parallélisme exploité, et donc une restriction de la classe des graphes d'exécution possibles, permettent de contrôler cet espace mémoire nécessaire.

6.3.2.1 Cas général

Nous montrons dans cette section qu'il n'est pas possible, dans un cas général, d'obtenir à la fois une efficacité linéaire en temps et une consommation raisonnable de mémoire.

Proposition 9 *Pour toute durée T_1 et tout espace mémoire S_1 il existe des applications de durée d'exécution T_1 et de consommation mémoire S_1 sur 1 processeur telles que*

$T_\infty \approx \sqrt{T_1}$ et pour tout p :

$$T_p \leq T_1/2 \Longrightarrow S_p \geq \frac{\sqrt{T_1}}{2} S_1.$$

Les propriétés de cette application sont telles que $\bar{p} \approx \sqrt{T_1}$, ce qui laisse l'utilisateur en droit d'attendre une accélération raisonnable pour toute exécution parallèle.
Preuve. Étant donnés T_1 et S_1, nous exhibons une application possédant les caractéristiques exposées dans la proposition. Cette application est une version simplifiée de celle présentée dans [16]. Cette application, représentée figure 6.1 page 121, est composée de :

- $\frac{\sqrt{T_1}}{2}$ tâches W chacune de durée $\sqrt{T_1}$ et allouant un espace mémoire de taille 0.
- $\frac{T_1}{4}$ tâches A de durée 1 et allouant chacune un espace mémoire de taille S_1.
- $\frac{T_1}{4}$ tâches L de durée 1 et chacune libérant un espace mémoire de taille S_1.

Les tâches A et L sont appariées, ces paires étant elles-mêmes groupées par tranches de $\frac{\sqrt{T_1}}{2}$: il y a donc $\frac{\sqrt{T_1}}{2}$ telles tranches. À chacune de ces tranches est associée une tâche de type W. Les dépendances entre les tâches, représentées figure 6.1 page 121 sont les suivantes :

- Les tâches A n'ont aucune contrainte de précédence.
- Chaque tâche W doit attendre l'exécution des $\frac{\sqrt{T_1}}{2}$ tâches A de la tranche qui la précède avant de débuter son exécution. La tâche W de la première tranche n'a aucune contrainte de précédence.
- Chaque tâche L doit attendre la terminaison de la tâche W associée à la tranche et la terminaison de la tâche A qui lui est appariée.

Il est aisé de vérifier que l'exécution sur un processeur unique peut être effectuée en temps T_1 dans un espace mémoire S_1 : il suffit pour cela d'exécuter les tranches les unes après les autres, et pour chaque tranche d'abord la tâche W puis pour chaque paire la tâche d'allocation A puis la tâche de libération L.

En remarquant que les tâches A, L et W sont indépendantes entre-elles, que les tâches A n'ont aucune contrainte de précédence, que les tâches W n'ont de dépendance qu'avec les tâches A et enfin que les tâches L sont dépendantes des tâches A et W, il vient que $T_\infty = \sqrt{T_1} + 2$. Il est donc *a priori* possible d'atteindre des accélérations allant jusqu'à $\bar{p} = \frac{T_1}{T_\infty} = \frac{1}{1+\frac{2}{\sqrt{T_1}}}\sqrt{T_1}$.

Considérons l'exécution de cette application sur une machine possédant p processeurs. Si aucune des tâches W n'est exécutée en concurence, alors nécessairement $T_p \geq \frac{\sqrt{T_1}}{2} \times \sqrt{T_1}$, et donc $\frac{T_1}{T_p} \leq 2$. Obtenir une meilleure performance implique donc l'exécution en concurrence d'au moins deux tâches W. Considérons dans la suite les tâches W numérotées, W_i avec $1 \leq i \leq \frac{\sqrt{T_1}}{2}$, selon l'ordre de gauche à droite de la figure 6.1 page 121.

Plaçons nous à un instant t où deux tâches W_i et W_j, $i < j$, s'exécutent en concurrence. Comme W_j est en cours d'exécution, les $\frac{\sqrt{T_1}}{2}$ tâches A qui précèdent son exécution sont terminées. Étudions l'état de la tâche W_{j-1}. Si cette tâche n'a pas encore commencé

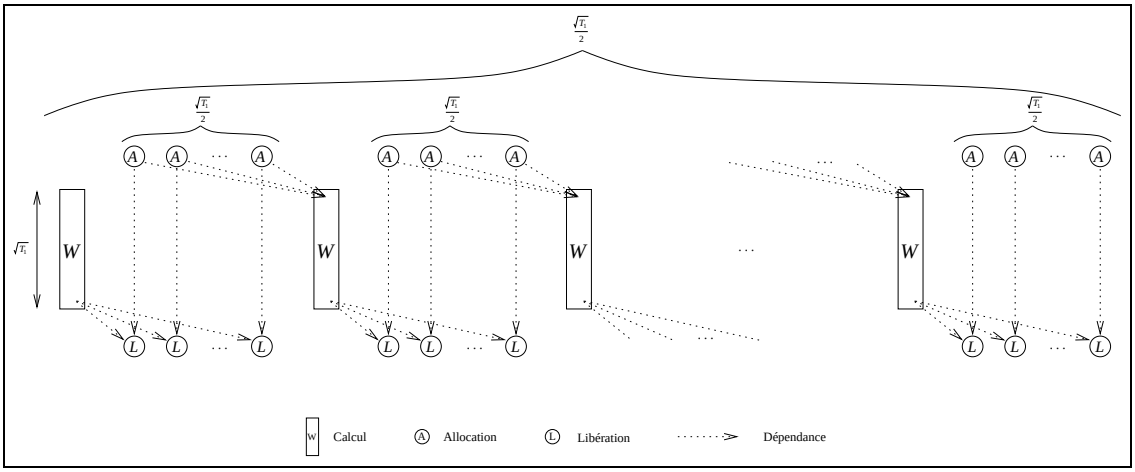

Figure 6.1 *Application consommatrice de mémoire en cas d'accélération lors d'une exécution parallèle.*
Cette application, s'exécutant sur un processeur en temps T_1 et consommant un volume mémoire S_1, est constituée de $\frac{\sqrt{T_1}}{2}$ tâches W de travail, de $\frac{T_1}{4}$ tâches d'allocation et de $\frac{T_1}{4}$ tâches de libération. Les contraintes de précédence entre ces tâches sont telles que toute exécution parallèle vérifiant $T_p < \frac{T_1}{2}$ implique une consommation mémoire $S_p \geq \frac{\sqrt{T_1}}{2} S_1$.

son exécution ou est en cours d'exécution (par exemple si $j = i + 1$), alors les tâches L correspondant aux allocations n'ont pas pu être exécutées : le volume mémoire correspondant à ces allocations est $\frac{\sqrt{T_1}}{2} S_1$. Si par contre cette tâche W_{j-1} est terminée, on peut supposer que les tâches L ont été exécutées. Mais la terminaison de cette tâche implique que les $\frac{\sqrt{T_1}}{2}$ tâches A qui la précède ont été terminées. On se retrouve donc dans la situation précédente mais cette fois avec la tâche d'indice $j - 1$. L'itération de cette analyse se terminera nécessairement puisque W_i n'est pas terminée mais en cours d'exécution.

Pour cette application, toute efficacité supérieure à 2 implique donc une consommation mémoire lors de l'exécution parallèle importante : $S_p \geq \frac{\sqrt{T_1}}{2} S_1$. □

L'application réelle doit créer l'ensemble de ces tâches. Le travail associé à cette création est égal au nombre de tâches, c'est-à-dire à $\frac{T_1}{2} + \frac{\sqrt{T_1}}{2}$. Cette création peut par exemple être menée en parallèle pour chaque tranche et de manière séquentielle à l'intérieur de chaque tranche, la profondeur de la création est alors de l'ordre de $\sqrt{T_1}$. Cette phase de création ne modifie donc pas les caractéristiques de l'application (T_1 et T_∞ restent du même ordre).

6.3.2.2 Restriction sur la classe des graphes d'exécution générables

En limitant les possibilités de synchronisations entre les tâches, il est possible d'interdire la construction d'applications telles que celle présentée dans la section 6.3.2.1 page 119. Un ordre d'exécution particulier des tâches du graphe permet alors la prédiction de l'espace mémoire nécessaire à l'exécution.

Le langage Cilk n'autorise que la programmation d'applications *strictes*, applications

dans lesquelles toute tâche ne peut imposer une contrainte de précédence que sur la continuation de la tâche mère qui l'a créée. L'application précédente ne peut donc plus être codée dans un tel modèle. Avec de telles restrictions il est possible de construire un algorithme d'ordonnancement [15] tel que l'espace mémoire nécessaire à l'exécution de l'application sur p processeurs avec une accélération linéaire $\frac{T_1}{T_\infty} = \Omega(p)$ est telle que $S_p \leq pS_1$. Ce résultat est obtenu en effectuant un parcours de type profondeur d'abord du graphe de tâches : lorsque sur un processeur une tâche crée une nouvelle tâche, alors le processeur continue l'exécution avec la tâche nouvellement créée. À tout instant de l'exécution le graphe en cours de développement possède donc au plus p feuilles et chacune de ces feuilles est en cours d'exécution sur un processeur. Chaque feuille nécessitant un espace mémoire inférieur à S_1 pour s'exécuter, il vient que $S_p \leq pS_1$.

Le langage NESL impose également un type de graphe série-parallèle et l'ordre d'exécution des tâches est un parcours en profondeur du graphe (parallélisme emboîté), parcours exécuté par p processeurs [82]. Un tel ordre d'exécution permet de garantir [12] que l'ordonnancement sur p processeurs de toute application est tel que $S_p \leq O(S_1 + p \log pT_\infty)$ avec $T_p \leq O\left(\frac{T_1}{p} + \log pT_\infty\right)$.

6.3.2.3 Utilisation d'un ordre séquentiel implicite

Comme nous l'avons vu précédemment, la consommation mémoire S_p d'une exécution parallèle est comparée à la consommation mémoire d'une exécution séquentielle sur un processeur de la même instance d'application.

La consommation mémoire d'une application dépend énormément de l'ordre d'exécution des tâches. Considérons par exemple l'application possédant n tâches a_i allouant 1 bit et n tâches l_i effectuant les libérations correspondantes, telles que les seules contraintes de précédence soient du type $a_i \prec l_i$. Considérons les deux ordres d'exécution séquentiels suivants :

$$a_1 < b_1 < \ldots < a_i < b_i < \ldots < a_n < b_n \quad (1)$$
$$a_1 < \ldots < a_i < \ldots < a_n < b_1 < \ldots < b_i < \ldots < b_n \quad (2)$$

Le volume mémoire requis pour l'exécution correspondant à l'ordre (1) est de 1 bit, tandis que celui correspondant à l'ordre (2) est de n bits. Le volume mémoire requis dépendant fortement de l'ordre, le principe pour contrôler cette consommation lors d'une exécution parallèle est de **suivre l'ordre séquentiel** qui a été utilisé pour définir S_1 [83]. Il est alors possible de comparer S_p à S_1.

Nous présentons ici la stratégie d'ordonnancement présentée dans [83, 9] et qui permet de limiter la consommation mémoire de toute exécution parallèle par $S_1 + pKT_\infty$. Le parallélisme est de type série-parallèle avec une description emboîtée du graphe. De plus chaque tâche est supposée de durée unitaire et alloue au maximum un volume K de mémoire.

Cette stratégie maintient une liste de tâches prêtes ordonnées par priorité. La priorité de chaque tâche correspond à son numéro dans l'ordre d'exécution séquentiel (le

parcours en profondeur d'abord du graphe)[3]. Les tâches les plus vieilles dans l'ordre séquentiel sont donc plus prioritaires que les autres. Il est montré [83] qu'au plus T_∞ tâches par processeur peuvent être exécutées simultanément en avance sur leur numéro d'ordre. Chaque tâche allouant au plus un volume K de mémoire, le volume consommé en plus de l'exécution séquentielle est donc majoré par pKT_∞.

Il est possible d'étendre cette preuve aux tâches allouant une volume m arbitraire de mémoire : il suffit d'introduire $\frac{m}{K}$ tâches fictives avant cette tâche d'allocation. Les tâches effectuant de grosses allocations seront alors précédées par un ensemble de ces tâches fictives qui vont avoir pour effet de retarder leur exécution. Ainsi, ces allocations étant retardées, elles n'auront que peu de chance d'être effectuées en parallèle. Ainsi, dans le cas de l'exemple de la section 6.3.2.1 page 119, chacune des tâches A seraient précédées par un ensemble de tâches fictives, ce qui aurait pour effet de retarder leur exécution jusqu'à, par exemple, la terminaison de la tâche W. Une tâche de libération L pourra donc être exécutée après chaque exécution de tâche d'allocation.

6.3.3 Bilan

La durée de l'exécution d'une application peut facilement être obtenues et, moyennant certaines restrictions sur le graphe d'exécution et en se basant sur un ordre séquentiel implicite des tâches, la consommation mémoire controlée. Ces garanties permettent donc d'associer un modèle de coût au modèle de programmation du langage.

Nous présentons dans la suite le cas de l'interface applicative Athapascan-1 pour laquelle un modèle de coût, basé sur la description de l'exécution par un graphe de flot de données, est défini. Les performances dépendant de l'ordonnancement et l'ordonnancement étant séparé de la bibliothèque, les garanties offertes par le modèle de coût dépendent donc de la politique d'ordonnancement choisie lors de l'exécution.

6.4 Contrôle de la consommation mémoire d'un programme Athapascan-1

Les performances de l'exécution des applications dépendent directement de la politique d'ordonnancement utilisée. L'ordonnancement est effectué dans Athapascan-1 par un module séparé de la bibliothèque. Ce module est présenté section 7.2 page 130. Nous décrivons dans cette section deux politiques d'ordonnancement qui permettent de garantir les performances de l'exécution au niveau de la consommation mémoire.

L'exécution est supposée être effectuée sur une machine à mémoire distribuée possédant p processeurs identiques. Afin de permettre la majoration des temps d'accès aux mémoires distantes, les algorithmes proposés dans [71] sont utilisés pour simuler une mémoire globale pour les p processeurs à l'aide de fonctions de hachage universel. Afin d'obtenir une simulation quasi-optimale (le délai h ne dépendant quasiment pas de p) une

[3]Le numéro n'est pas connu en réalité, mais l'ordre est maintenu par une technique de chaînage de la liste : les tâches filles sont insérées à l'ancienne position de la mère dans la liste.

technique de multiprogrammation légère permettant d'exploiter le degré de parallélisme, *parallel slackness* [89, 103], est utilisée. Cette technique consiste en l'émulation préemptive de plusieurs processeurs virtuels sur chaque processeur physique de la machine.

Nous reprenons les grandeurs caractéristiques de l'exécution présentées section 6.2 page 116. L'ordre séquentiel considéré pour la définition de ces grandeurs est l'ordre de référence de la définition 7 page 70.

À ces grandeurs caractéristiques nous ajoutons les grandeurs suivantes afin de tenir compte de la particularité de la machine qui est à mémoire distribuée et des communications nécessaires :

- q, le nombre de processeurs virtuels émulés sur les p processeurs réels de la machine. Nécessairement :
$$p \leq q$$
- h, le délai d'accès à distance d'un bit de donnée.
- C_1, le volume total d'accès distant. Cette valeur représente la somme sur tout le graphe d'exécution G des tailles des données accédées en lecture directe (mode d'accès `r` ou `r_w`).
- C_∞, le volume d'accès distant effectué par un plus long chemin dans ce graphe (selon ce critère d'accès).
- σ, la taille du graphe G.
$$\sigma = |V| + |E|$$
Le graphe nécessite donc σ opérations pour être construit. La gestion des états associés aux nœuds permettant de résoudre les contraintes de précédence est basée sur un mécanisme de terminaison et correspond donc à une « dé-construction » du graphe : il y a donc également σ telles actions.

La première technique, notée $\mathcal{O}_1$ et présentée dans la définition 8 page 124, est l'utilisation directe de l'algorithme glouton présenté section 6.3.1.1 page 118. Les performances de l'exécution sont celles de la proposition 10 page 125. La seconde, notée $\mathcal{O}_2$ et présentée dans la définition 9 page 126, est une restriction du modèle de programmation au cadre des graphes de type série-parallèle, ce qui permet d'utiliser une politique d'ordonnancement basée sur le vol de travail similaire à celle utilisée dans le langage Cilk [51]. Les performances de l'exécution sont celles de la proposition 11 page 126.

6.4.1 Politique d'ordonnancement $\mathcal{O}_1$

Nous donnons dans cette section la définition de cette politique et évaluons théoriquement ses performances. Cet algorithme est un algorithme de type liste centralisée ordonnée avec une priorité donnée par l'ordre de « référence » $\prec_r$.

Définition 8 *La **politique d'ordonnancement** $\mathcal{O}_1$ consiste à exécuter de manière gloutonne les tâches prêtes en suivant au plus prêt l'ordre de « référence » $\prec_r$.*

L'implantation maintient de manière centralisée un graphe $G' \subset G$ représentant la portion créée et non encore terminée de G. Une liste P_I de processeurs inactifs, et une

liste $R_{\prec_r}$ de tâches prêtes, triées selon $\prec_r$, sont également maintenues. La cohérence de ces structures est assurée par un verrou global qui sera pris avant chaque modification.

L'ordonnancement est effectué de la manière suivante :

- *Chaque fois qu'un processeur devient inactif, il prend la première tâche de la liste $R_{\prec_r}$ et l'exécute. Si cette liste est vide il s'insère dans la liste P_I des processeurs inactifs.*
- *Chaque fois qu'une tâche devient prête, elle est affectée à l'un des processeurs de la liste P_I. Si cette liste est vide, la tâche est insérée (en temps constant) dans la liste $R_{\prec_r}$ de tâches prêtes.*

Proposition 10 *[72] L'exécution de toute application ordonnancée par la politique d'ordonnancement $\mathcal{O}_1$ est telle que (les coûts d'ordonnancement et de communication sont pris en compte, chaque tâche est supposée allouer un volume borné de mémoire) :*

$$\begin{aligned} T_p &\leq \frac{T_1 + hC_1}{p} + \frac{q}{p}(T_\infty + hC_\infty) + h\frac{q}{p}O(\sigma) \\ S_p &\leq S_1 + qO(T_\infty + hC_\infty) \end{aligned}$$

Preuve. En reprenant la démarche utilisée dans [64, 72] et dans la preuve du théorème 3 page 118, l'ensemble des tops de l'exécution peut être partitionné en trois sous-ensembles :

1. A (activité) : pour tout top de A, tous les q processeurs de la machine sont en train soit d'exécuter une instruction de l'application soit d'effectuer un accès à distance. Nécessairement :
$$|A| \leq \frac{T_1 + hC_1}{q}$$
2. I (inactivité) : pour tout top de I, un au moins des q processeurs est inactif ($|P_I| \neq 0$). En utilisant la même technique que celle utilisée dans la preuve du théorème 3 page 118 il vient :
$$|I| \leq \text{durée sur une infinité de processeurs} \leq T_\infty + hC_\infty$$
3. Q (ordonnancement) : pour tout top de Q, un au moins des q processeurs est en train de manipuler les structures maintenues globalement. Les accès concurrents sont sérialisés en utilisant le verrou global. La durée de chaque manipulation est majorée par $O(h)$. Il y a au plus 2σ manipulations des structures, donc :
$$|Q| \leq O(h\sigma)$$

La durée sur q processeurs $T_q = |A| + |Q| + |I|$ est donc majorée par $\frac{T_1+hC_1}{q} + T_\infty + hC_\infty + O(h\sigma)$. En remarquant que la durée d'exécution sur les p processeurs réels de la machine est telle que $T_p \leq \left\lceil \frac{q}{p} \right\rceil T_q$, on obtient la borne annoncée.

L'ordonnancement précédent exécute les tâches en suivant au plus près l'ordre de « référence » : il y a donc, à chaque top, au plus $q-1$ tâches qui sont exécutées en avance sur cet ordre, donc, à tout instant, un maximum de $(T_\infty + hC_\infty)(q-1)$. Chaque tâche étant supposée allouer une quantité bornée de mémoire, on obtient la majoration de l'espace mémoire annoncée. □

6.4.2 Politique d'ordonnancement $\mathcal{O}_2$

Nous donnons dans cette section la définition de cette politique et évaluons théoriquement ses performances. Cet algorithme est, à la base, un algorithme glouton auquel on impose aux vols de « suivre » *a priori* l'ordre de « référence ». Cette technique est utilisée dans le langage Cilk et entièrement détaillée dans [16].

Définition 9 *La **politique d'ordonnancement** $\mathcal{O}_2$ consiste à exécuter de manière gloutonne les tâches prêtes en suivant au plus prêt l'ordre séquentiel de création des tâches en élargissant les contraintes de précédence (définition 6 page 69) à tous les accès en lecture, (directs et différés).*

L'implantation maintient de manière distribuée un graphe $G' \subset G$ représentant la portion créée et non encore terminée de G. Chacun des q processeurs possède un ensemble de branches du graphe qui a été développé localement. Les tâches sont ordonnées dans chaque branche selon l'ordre d'un parcours en profondeur d'abord du graphe.

L'ordonnancement est effectué comme suit. Chaque fois qu'un processeur p termine une tâche t, il peut être dans l'une des deux situations suivantes :

- *Une tâche t' suit t dans la branche et t' est prête.*
- *La tâche qui suivait t a été volée par le processeur p'.*

Dans la première situation, p prend t' et l'exécute. Dans la seconde, il va voir sur p' si le reste de la branche qui a été volé est exécutable (c'est-à-dire si la première tâche de cette branche est prête et si cette branche n'est pas celle en cours d'exécution sur p'). Si oui, toute cette branche est volée par p qui continue son exécution avec la première tâche de son vol. Dans tous les autres cas, p vole à un processeur quelconque une tâche prête (et tout le reste de la branche) puis l'exécute.

Les nouvelles contraintes de précédence imposées sur les tâches sont telles que si une tâche est prête, alors une exécution séquentielle de toute la descendance de cette tâche est possible sans synchronisation. C'est ce qui permet l'exécution en profondeur de toute portion du graphe.

Proposition 11 *L'exécution de toute application ordonnancée par la politique d'ordonnancement $\mathcal{O}_2$ est telle que (les coûts d'ordonnancement et de communication sont pris en compte) :*

$$\begin{aligned} T_p &\leq \frac{T_1 + hC_1}{p} + h\frac{q}{p}O(T_\infty + hC_\infty) \\ S_p &\leq qS_1 \end{aligned}$$

Preuve. Cet ordonnancement étant glouton, les performances en temps sont données par le théorème 3 page 118. Le coût de l'implantation de cet ordonnancement est de l'ordre du nombre de tâches volées par chaque processeur. Or chaque processeur ne peut effectuer plus de $T_\infty + hC_\infty$ requêtes de vol et la durée de chaque vol est constante avec un grande probabilité. Nous ne détaillons pas ici le calcul de cette probabilité, calcul qui est effectué dans [15].

L'ordonnancement précédent exécute les tâches en suivant au plus près l'ordre séquentiel de création des tâches. L'exécution de toute la descendance d'une tâche requiert donc un volume mémoire majoré par S_1. On vérifie aisément qu'il y a au plus q descendances qui sont exécutées en concurrence, un processeur arrêtant l'exécution d'une branche uniquement dans le cas où un autre processeur est en cours d'exécution sur la continuation de cette même branche. L'espace mémoire nécessaire à l'exécution peut donc être majoré par qS_1. Il est à noter que cette borne est globale et ne garantit absolument pas que la mémoire utilisée par chacun des q processeurs est majorée par S_1. □

6.5 Bilan

L'exécution d'une application sur une machine parallèle est modélisée par un graphe de flot de données. Ce graphe décrit les contraintes de précédence entre les tâches de calcul et les accès aux données. Étant donné un graphe, les politiques d'ordonnancement utilisées dans les langages de « haut niveau » garantissent les performances en temps et en mémoire de toute exécution parallèle de ce graphe. Le tableau 6.1 page 127 récapitule les garanties d'efficacité d'ordonnancement fournies par les modèles de coût associés aux langages étudiés.

Langage	**Restriction sur le graphe**	T_p **majoré par**	S_p **majoré par**
Cilk	série-parallèle	$\frac{T_1}{p} + O\left(T_\infty\right)$	pS_1
NESL	série-parallèle emboîté	$O\left(\frac{T_1}{p} + \log pT_\infty\right)$	$O(S_1 + p \log pT_\infty)$
Athapascan-1	description emboîté	$\frac{T_1+hC_1}{p} + \frac{q}{p}(T_\infty + hC_\infty) + h\frac{q}{p}O\left(\sigma\right)$	$S_1 + qO(T_\infty + hC_\infty)$
	série-parallèle emboîté	$\frac{T_1+hC_1}{p} + h\frac{q}{p}O(T_\infty + hC_\infty)$	qS_1

Tableau 6.1 *Garanties d'efficacité de Cilk, NESL et Athapascan-1.*
Les architectures visées sont des machines à mémoire partagée pour Cilk et NESL et les machines à mémoire distribuée pour Athapascan-1. Dans le cas des machines à mémoire distribuée, le délai d'accès à un bit de donnée distant est supposé majoré par h : ceci est effectué en émulant q processeurs virtuels sur les p processeurs réels de la machine. Notons que seul Athapascan-1 tient compte des communications dans la formule de coût : Cilk est conçu pour les machines de type SMP *et NESL les ignore.*

Les techniques d'ordonnancement en ligne de type « glouton » où les processeurs sont maintenus au maximum en activité permettent d'atteindre des efficacités en temps quasi-optimales, de l'ordre de $\frac{T_1}{p} + T_\infty$. Si l'efficacité en temps peut être obtenue relativement facilement, il n'en est pas de même pour l'efficacité en mémoire. Différentes variantes des algorithmes gloutons permettent de garantir une majoration de la consommation mémoire, le principe de base étant de suivre au maximum l'ordre séquentiel d'exécution. Le respect de cet ordre sur une machine à plusieurs processeurs introduit cependant un coût

supplémentaire d'ordonnancement. NESL par exemple suit de plus près l'ordre séquentiel que Cilk, ce qui lui permet de majorer l'utilisation mémoire par $S_1 + p \log p T_\infty$ tandis que Cilk majore par pS_1. Le gain est substantiel pour des applications pour lesquelles $T_\infty \leq S_1$, ce qui est le cas par exemple pour le produit de deux matrices $n \times n$ pour lequel $T_\infty = \log n$ et $S_1 = n^2$. Cet ordonnancement a cependant un coût $\log p$ fois plus important.

Dans le cas d'Athapascan-1, le coût de cette technique peut être de l'ordre de la taille σ du graphe, taille qui peut être de l'ordre de T_1 pour certaines applications (le calcul de Fibonacci à un grain fin par exemple). Il est possible, en imposant des contraintes plus fortes de précédence entre les tâches, d'atteindre les mêmes performances que celles garanties par le langage Cilk ($\mathcal{O}_2$).

Nous présentons dans le chapitre suivant l'implantation dans Athapascan-1 des deux politiques d'ordonnancement $\mathcal{O}_1$ et $\mathcal{O}_2$ permettant de contrôler la mémoire requise pour toute exécution.

7

Implantation et évaluation du contrôle de la consommation mémoire dans Athapascan-1

Ce chapitre étudie l'ordonnancement dans la version distribuée de la bibliothèque Athapascan-1. Les principales sections de ce chapitre traitent les points suivants :

- Le **module d'ordonnancement** d'Athapascan-1 qui est entièrement séparé du reste ce qui permet une programmation aisée de nouvelles stratégies (section 7.2 page 130).
- L'**implantation** des stratégies d'ordonnancement $\mathcal{O}_1$ et $\mathcal{O}_2$ (section 7.3 page 137).
- L'**évaluation** de ces deux stratégies sur un exemple simple de calcul d'une portion de l'ensemble de Mandelbrot (section 7.4 page 139).

7.1 Le problème de l'ordonnancement

Les applications s'exécutant sur une machine parallèle sont décomposées en tâches qui s'exécutent simultanément sur les différents processeurs de la machine. Une fois les tâches déterminées et créées, leur ordonnancement est généralement un des problèmes majeurs posés par l'utilisation d'une machine parallèle. La qualité de l'ordonnancement a une influence cruciale sur les performances de l'exécution, mais le calcul d'un « bon » ordonnancement des tâches implique un surcoût. En effet, notons W_1 le travail de l'application et W_o celui effectué par l'ordonnancement. Alors nécesssairement $T_p \geq \frac{W_1+W_o}{p}$ et donc supérieur à $T_p^{opt} + \frac{W_o}{p}$.

Face à l'importance de ce problème de nombreuses stratégies d'ordonnancement ont été développées. Deux principaux types de stratégies sont à distinguer [24] : les stratégies de type **statique** où l'ordonnancement est calculé **avant** l'exécution et les stratégies de type **dynamique** où l'ordonnancement est cette fois calculé **au cours** de l'exécution. Les stratégies de type statique [60, 108] ont l'avantage de n'introduire aucun surcoût d'ordonnancement lors de l'exécution mais ne sont véritablement efficaces que dans le cadre des

applications régulières [58, 95] où le graphe des tâches de l'application ne dépend pas des conditions d'exécution. À l'opposé, les stratégies de type dynamique, où l'ordonnancement des tâches est calculé à la volée, permettent une bonne adaptation aux conditions d'exécution. Ces stratégies n'ont cependant qu'une vue partielle de l'application qui doit être ordonnancée et introduisent un surcoût à l'exécution.

La technique d'ordonnancement la plus efficace dépend hélas de l'application considérée [58, 18], et l'obtention d'une efficacité maximale passe donc par le codage de la stratégie d'ordonnancement dans l'algorithme de calcul. Cette programmation est cependant en général délicate et peut dans certains cas accroître significativement la complexité de la programmation. De plus, les stratégies simples et adaptatives, telles que les algorithmes gloutons présentés section 6.3.1.1 page 118 risquent d'être programmées et reprogrammées de nombreuses fois. C'est pourquoi les langages parallèles de « haut niveau » tentent d'offrir des outils permettant un ordonnancement efficace des tâches sans être obligé de se poser le problème de la programmation de la stratégie. Par exemple l'environnement Pyrros [60] pour un ordonnancement statique, ou les langages Cilk, NESL, Jade, Sisal [48] qui intègrent dans leur noyau une politique d'ordonnancement. Cette solution, si elle permet d'affranchir l'utilisateur du problème d'ordonnancement, s'avère limitée si les conditions d'exécution ne sont pas celles de prédilection de l'algorithme choisi pour faire la régulation : c'est entre autres le problème dont souffre la version distribuée [90] du langage Cilk. Une autre alternative est de séparer entièrement la politique d'ordonnancement de l'application [107, 106, 58, 34] et de proposer non pas une seule et unique stratégie d'ordonnancement, mais tout un ensemble : l'utilisateur décidera, éventuellement expérimentalement, quelle est la stratégie la mieux adaptée à sa situation. C'est la solution retenue par les environnements de programmation tels que Dynamo [102], PAC++-Givaro [57, 56] ou Athapascan-1.

Nous présentons dans la suite de ce chapitre tout d'abord l'environnement d'ordonnancement défini dans [29, 26, 25] constituant le module d'ordonnancement utilisé au sein de la bibliothèque Athapascan-1, puis l'implantation à l'aide de cet environnement des trois stratégies d'ordonnancement présentées au cours du chapitre 6 page 115 : une stratégie gloutonne permettant d'obtenir une durée d'exécution théoriquement quasi-optimale et les deux stratégies introduites au chapitre précédent, $\mathcal{O}_1$ et $\mathcal{O}_2$, permettant de contrôler théoriquement l'espace mémoire nécessaire à l'exécution. Ces stratégies seront comparées expérimentalement en fonction du volume mémoire nécessaire à l'exécution de l'application sur plusieurs processeurs.

7.2 Implantation des politiques d'ordonnancement en Athapascan-1

L'ordonnancement est assuré dans Athapascan-1 par un module séparé de la bibliothèque. Nous détaillons dans cette section les interfaces des quatre différents composants du module d'ordonnancement :

– Le générateur de tâches, constitué par la bibliothèque Athapascan-1.

– Le module d'ordonnancement, chargé d'affecter un site d'exécution à chaque tâche.
– Le module d'exécution, chargé d'exécuter les tâches qui lui sont confiées.
– Le module d'information de charge, chargé de fournir des informations relatives à l'état d'activité de la machine.

La figure 7.1 page 131 représente l'interaction de ces différents modules au sein d'une architecture distribuée. Le principe de base de cette distribution repose sur une réplication de chacun des modules sur chacun des nœuds de la machine. Chaque version locale d'un module n'interagit qu'avec les réplicats locaux des autres modules (par appels de fonctions) et les autres réplicats de son module situés sur d'autres sites (par échange de messages).

Nous présentons dans la suite tout d'abord le noyau d'exécution parallèle Athapascan-0 sur lequel reposent les implantations d'Athapascan-1 et du module d'ordonnancement, puis nous détaillons les quatre composants de ce module ainsi que leurs interactions.

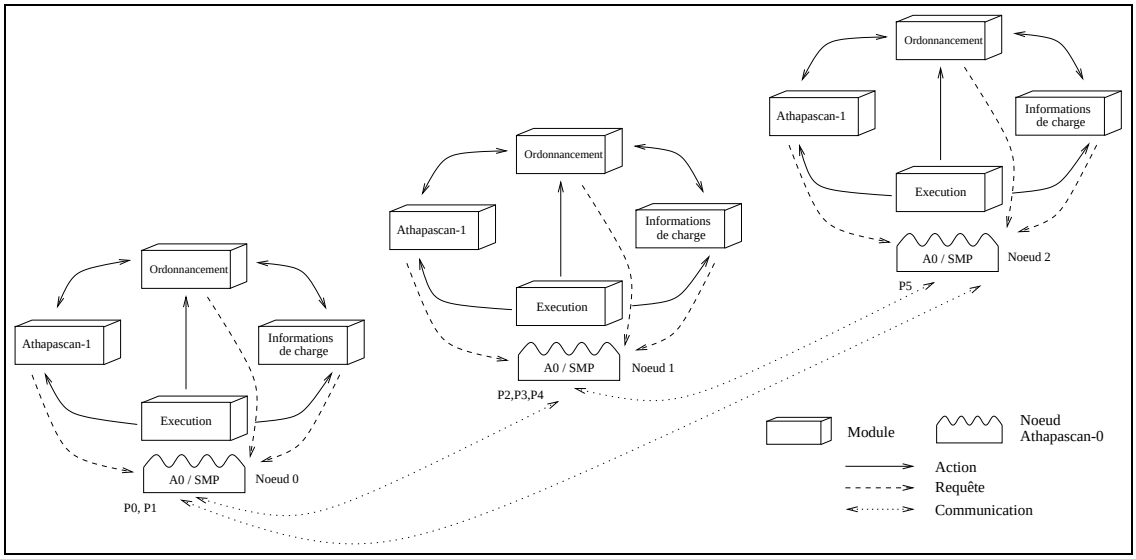

Figure 7.1 *Interaction des différents modules dans le cadre d'une machine distribuée. Les différents nœuds constituants la machine parallèle sont gérés par la bibliothèque Athapascan-0. Caque nœud peut contenir plusieurs processeurs physiques (ce qui permet d'exploiter les machines de type* SMP*). Chaque module est présent sur chacun des nœuds et les différents réplicats d'un même module peuvent communiquer au moyen des primitives de communications offertes par Athapascan-0. Le réplicat d'un module n'interagit qu'avec les représentants locaux des autres modules.*

7.2.1 Athapascan-0

Athapascan-0[1] [19, 61] est un noyau exécutif pour machines parallèles supportant la multiprogrammation légère (ou *multi-threading*). L'abstraction de la machine parallèle offerte par Athapascan-0 consiste en l'interconnexion par un réseau d'un ensemble de nœuds de calculs. Les primitives offertes par la bibliothèque permettent de créer des fils

[1] L'*URL* de la page officielle du projet est la suivante : `http://www-apache.imag.fr/software/ath0/`.

d'exécution de calcul (*threads*) sur chaque nœud et d'échanger des messages contenant des données entre des threads situés sur des nœuds différents.

Ce noyau exécutif permet d'exploiter quatre types de parallélisme pouvant être présents sur une machine parallèle :

- Le parallélisme inter-nœuds, chaque nœud exécutant une copie du noyau d'exécution Athapascan-0 et de l'application. Les nœuds progressent donc dans leurs exécutions indépendamment les uns des autres (mises à part les synchronisations dues aux communications).
- Le parallélisme intra-nœud, plusieurs fils d'exécution étant présents sur un même nœud. En effet, si ce nœud[2] possède plusieurs processeurs, ces fils d'exécution pourront être répartis entre ces processeurs par le système d'exploitation gérant ce nœud.
- Le parallélisme entre calculs et communications, en offrant des primitives asynchrones de communication. Ce type de parallélisme est également exploitable en utilisant des communications bloquantes et en s'appuyant sur le parallélisme intra-nœud (en utilisant alors plusieurs fils, certains calculant, d'autres étant synchronisés sur des communications).
- Le parallélisme entre les communications, si un fil d'exécution initie plusieurs communications non bloquantes ou si plusieurs fils d'exécution communiquent à partir du même nœud.

Cette bibliothèque constitue, en offrant exactement les primitives de multiprogrammation légère et de communication nécessaires, la couche de portabilité d'Athapascan-1. Il peut donc être installé sur toute machine possédant Athapascan-0 et un compilateur C++ supportant le mécanisme de classes « patrons »[3]. Athapascan-0 est implanté de telle manière que sa couche de portabilité est réduite à l'utilisation d'une bibliothèque de processus légers (généralement des threads POSIX [69]) et d'une bibliothèque de communications (généralement MPI [40]).

7.2.2 Le module générant les tâches : Athapascan-1

Au sein du système exécutif, le rôle de la bibliothèque Athapascan-1 est de créer des tâches. Ces tâches sont générées localement et font partie d'un graphe les reliant aux données partagées : ce graphe traduit les contraintes de précédence et de localité.

Bien que les tâches soient toujours générées localement, le graphe est réparti. Les tâches sont en effet placées sur les différents nœuds de la machine, selon les décisions du module d'ordonnancement. La gestion de la cohérence du graphe, le calcul des états de ses nœuds et la synthèse des données deviennent de ce fait des opérations distribuées.

[2] Un nœud Athapascan-0 est actuellement implanté comme un processus lourd UNIX. Si plusieurs processeurs physiques sont à la dispositions du système, le cas d'un SMP par exemple, les processus légers contenus dans le nœud seront répartis par le système. C'est la situation des nœuds 0 et 1 représentés figure 7.1 page 131.

[3] L'implantation d'Athapascan-1 utilise de manière intensive les classes « patrons », ou *template* [100], afin d'effectuer le maximum de l'analyse lors de la passe de compilation. Ces classes sont parfois mal supportées par les compilateurs. Les dernières versions publiques d'`egcs` semblent pourtant les supporter convenablement.

La gestion distribuée de ce graphe nécessite la coopération des différents réplicats du module de la bibliothèque. Cette coopération est effectuée au moyen des primitives de communications offertes par le noyau Athapascan-0.

7.2.3 Le module d'exécution

Le module d'exécution est composé d'un pool de processeurs virtuels. Ces processeurs sont implantés sous forme de fils d'exécution (*threads* Athapascan-0), ce qui permet d'exploiter le parallélisme intra-nœud et le recouvrement des communications par du calcul. Ces processeurs virtuels seront responsables de l'exécution effective des tâches.

7.2.3.1 Fonctionnement

Chaque processeur virtuel, autonome, exécute une boucle infinie, présentée figure 7.2 page 133, qui consiste à demander une tâche, l'exécuter puis... recommencer.

```
while( ! ended ) {
   t= ordo.get_task();
   if( t )
      t.execute();
   else
      ordo.wait();
}
```

Figure 7.2 *Boucle exécutée par tout processeur virtuel.*
Chaque processeur virtuel demande une tâche à exécuter, l'exécute, puis retourne au point de départ. Si aucune tâche n'est disponible le processeur se bloque en attente passive et sera réveillé lorsque le module d'ordonnancement aura de nouveau du travail à donner ou, au plus tard, à la fin de l'application.

L'existence de ce module d'exécution est due aux enseignements tirés d'un premier prototype de la bibliothèque Athapascan-1a [43, 27]. Dans ce prototype, dès qu'une tâche était prête et sur son site d'exécution, un fils Athapascan-0 était créé pour son exécution. Ce fonctionnement, simple et intuitif, impliquait cependant la création d'un grand nombre de fils d'exécution dont le coût était important. La gestion d'un nombre trop important de fils par Athapascan-0 était également problématique.

7.2.3.2 Interface avec la bibliothèque Athapascan-1

Il n'y a pas, à proprement parler, d'interaction directe entre ce module d'exécution et la bibliothèque. Cependant, les tâches ou les objets en mémoire partagée sont créés lorsque certaines instructions (`Fork`, `Shared`) du corps des tâches sont exécutées par les processeurs virtuels. C'est uniquement pour traduire ces créations d'objets qu'un lien a été représenté dans la figure 7.1 page 131.

7.2.3.3 Interface avec le module d'ordonnancement

L'interaction entre le module d'exécution et le module d'ordonnancement est de type *receiver initiated* : c'est lorsqu'un des processeurs virtuels devient inactif que le module

d'ordonnancement est contacté et doit fournir du travail, et non le module d'ordonnancement qui affecte les tâches dès qu'elles sont prêtes aux processeurs virtuels.

Dans la boucle représentée figure 7.2 page 133 l'interaction avec le module d'ordonnancement se fait à travers les deux fonctions `ordo.get_task()` et `ordo.wait()`. La première demande une nouvelle tâche à exécuter lorsque le processeur vient de finir l'exécution d'une tâche et la seconde est une synchronisation qui intervient lorsqu'aucune tâche n'est prête à être exécutée. Le fil d'exécution sera réveillé soit par le module d'ordonnancement si une tâche vient à être disponible, soit par le système si l'application est terminée.

7.2.4 Module d'ordonnancement

Le module d'ordonnancement est constitué d'un certain nombre de groupes, chacun d'entre eux implantant une politique d'ordonnancement. Un groupe représente donc cette politique particulière au sein du module d'ordonnancement. Chaque groupe est autonome et seul responsable de l'ordonnancement des tâches qu'il contient. Les interactions avec les autres modules, représentés figure 7.3 page 135, sont décrites dans la suite de cette section.

7.2.4.1 Fonctionnement

Les tâches sont fournies aux groupes par la bibliothèque Athapascan-1 dès leur création. Par défaut toute tâche fille est donnée au même groupe que sa mère. Conformément à la politique qui lui est associée, chaque groupe décide du nœud sur lequel chaque tâche doit être exécutée et, au niveau de chaque nœud, quelle tâche doit être donnée au module d'exécution lorsqu'un processeur virtuel demande du travail. Le module d'ordonnancement se comporte donc comme un filtre entre le module de génération du graphe constitué par la bibliothèque Athapascan-1 et le module d'exécution.

7.2.4.2 Interface avec la bibliothèque Athapascan-1

Les interactions possibles entre un groupe d'ordonnancement et la bibliothèque sont les suivantes :

- Lorsqu'une tâche est créée au niveau de la bibliothèque, elle est confiée au groupe responsable de son ordonnancement (`new_task`). Le choix du groupe est effectué par l'utilisateur, comme illustré figure 7.4 page 135. Par défaut, la tâche créée est placée dans le même groupe que la tâche créatrice.
- Chaque fois que l'état d'une tâche évolue, le groupe auquel elle appartient est averti (`new_state`). Cette fonction est appelée en particulier lors de la terminaison de la tâche. Voir le tableau 3.1 page 58 pour une définition des différents états et la figure 3.3(a) page 59 pour leurs possibles évolutions.
- Le groupe peut explorer la portion locale du graphe des tâches. Cette exploration se fait à partir d'une des tâches dont il est possible d'obtenir les données partagées

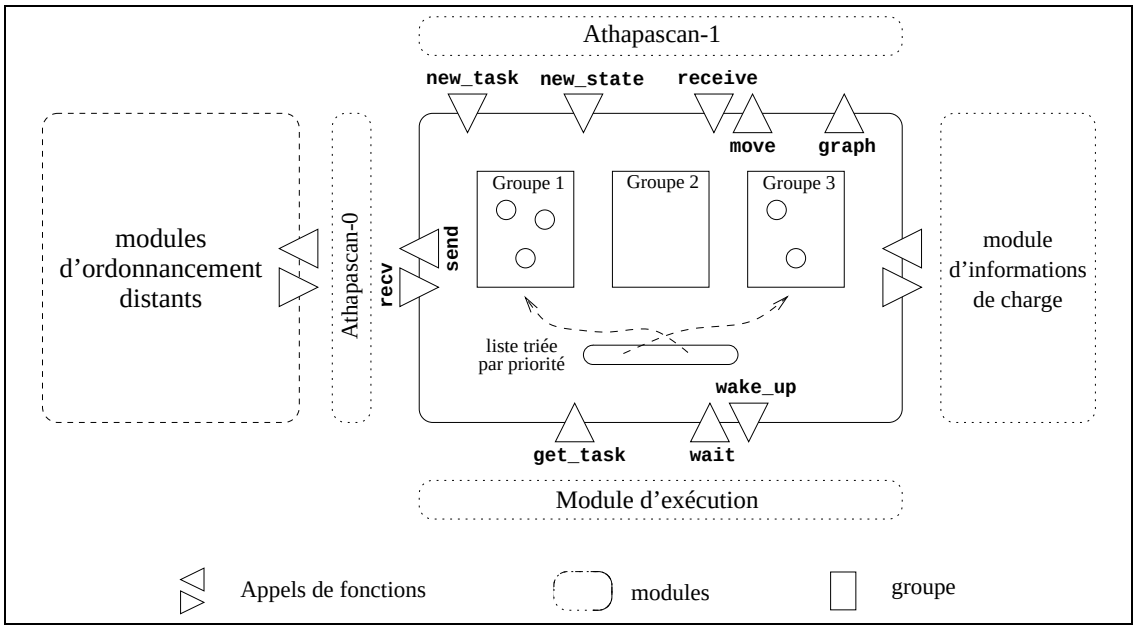

Figure 7.3 *Interfaces du module d'ordonnancement.*
Le module d'ordonnancement est en interaction locale avec la bibliothèque Athapascan-1, le module d'exécution et le module d'informations de charge. Il est également en relation avec les autres réplicats du module d'ordonnancement grâce aux primitives de communications offertes par Athapascan-0.

```
{
   ...
   policy::group grp;
   a1_set_default_group( grp );
   Fork< t >( <parametres> );
   ...
}
```

Figure 7.4 *Changement du groupe d'ordonnancement des tâches créées.*
Toutes les tâches créées après le changement du groupe par défaut, donc en particulier t, seront placées dans le groupe associé à la politique d'ordonnancement $policy$. L'utilisateur peut changer autant de fois le groupe par défaut de la tâche créatrice qu'il le désire.

accédées. Les contraintes de localité peuvent donc être analysées par la politique d'ordonnancement. À partir de ces données, il est également possible d'obtenir la liste des tâches locales les accédant ainsi qu'une liste des sites sur lesquels d'autres tâches les accèdent également (le graphe étant distribué par rapport aux données, section 4.2). Une série de fonctions est associée à toutes ces actions possibles et est représentée sous la dénomination générique `graph` dans la figure 7.3 page 135.

– Si le groupe décide de déplacer une tâche vers un autre site, il soumet une requête de déplacement (`move`) à la bibliothèque. La tâche sera déplacée, ainsi que ses données, vers le site destinataire. Lors de la réception sur le site distant, le réplicat du groupe sera averti de l'arrivée de cette tâche (`receive`). Le groupe ne peut gérer tout seul le déplacement des tâches car dans ce cas la cohérence du graphe doit être maintenue.

7.2.4.3 Interface avec le module d'exécution

Lorsque le module d'exécution demande une tâche à exécuter (`get_task`), le module d'ordonnancement transmet cette requête au groupe de plus forte priorité. Si la liste des groupes possédant au moins une tâche exécutable est vide, le module retourne une tâche vide (ce qui aura pour effet de placer le processeur virtuel dans un état d'attente, comme spécifié figure 7.2 page 133). Lorsqu'un des groupes qui ne possédait pas de tâche exécutable en possède soudain une (suite à la création ou la réception d'une tâche, l'évolution d'un état), ce groupe est inséré dans la liste en respectant sa priorité. Si un processeur virtuel s'était stoppé dans le module d'ordonnancement (`wait`), alors ce processeur est réveillé (`wake_up`) ce qui lui permet de tenter à nouveau d'obtenir une tâche à exécuter.

7.2.4.4 Interface avec les autres réplicats

Un groupe peut communiquer avec son homologue situé dans un autre réplicat du module d'ordonnancement au moyen des primitives de communications offertes par Athapascan-0. Les interactions entre ces groupes dépendent donc de la politique d'ordonnancement implantée : pour un placement de type aléatoire où les tâches sont affectées aux nœuds arbitrairement au moment de leur création aucune interaction n'est nécessaire ; pour d'autres, comme les algorithmes gloutons basés sur une technique de vol de travail, des requêtes de vol doivent être émises.

Il est à noter que deux groupes d'ordonnancement différents n'ont aucune interaction directe entre eux. Le seul lien possible est à travers le module d'information de charge qui permet à un groupe d'avoir accès à des informations fournies par un autre groupe.

7.2.5 Module d'information de charge

Le module d'information maintient une série d'indices permettant de caractériser l'état de charge de la machine à un instant donné. Les données maintenues par ce module sont classiquement fournies par le module d'information en ce qui concerne les données « systèmes » telles que la charge de la machine, la durée d'exécution des tâches, ou par le module d'ordonnancement en ce qui concerne les données « applicatives » telles que le nombre de tâches restant à exécuter.

Ce module peut, à partir de ces données, synthétiser des informations qui seront utilisées par les politiques du module d'ordonnancement [46, 49] : par exemple, l'estimation de la durée d'exécution d'une tâche sur laquelle sont basés les algorithmes développés dans [36].

L'interaction de ce module avec les autres dépend de l'implantation de la politique d'ordonnancement qui définira les informations dont elle a besoin. Il n'est en effet pas raisonable de maintenir *a priori* un ensemble d'informations si elles ne sont pas utilisées vu le coût de leur mise à jour. De plus deux politiques d'ordonnancement ne requièrent que rarement des informations similaires.

Il est à noter que pour les stratégies d'ordonnancement que nous proposons ce module n'est pas utilisé.

7.3 Implantations de quatre algorithmes d'ordonnancement dans Athapascan-1

Nous présentons dans cette section l'implantation, au sein du module d'ordonnancement présenté dans le début de ce chapitre, de quatre stratégies d'ordonnancement :

- Un algorithme de placement arbitraire.
- Un algorithme glouton présenté section 6.3.1.1 page 118.
- L'algorithme $\mathcal{O}_1$ présenté section 6.4.1 page 124.
- L'algorithme $\mathcal{O}_2$ présenté section 6.4.2 page 126.

7.3.1 Algorithme de placement arbitraire

Chaque processeur maintient une liste locale de tâches prêtes gérée de manière LIFO. Chaque fois qu'une nouvelle tâche est créée, elle est arbitrairement envoyée à un processeur qui sera le site de son exécution. L'implantation de cet algorithme au sein du module d'ordonnancement est la suivante :

- `new_task` : la tâche est envoyée arbitrairement à un processeur de la machine.
- `new_state` : si une tâche passe à l'état prêt, elle est insérée en tête de la liste locale et les processeurs virtuels bloqués dans le module d'ordonnancement sont réveillés. Toutes les autres modifications d'état sont ignorées.
- `receive` : si la tâche reçue est prête à être exécutée alors elle est insérée en tête de la liste locale. Sinon, elle est ignorée (on attend qu'elle soit prête pour la prendre en considération).
- `get_task` : la tâche située en tête de la liste est retournée. Si la liste était vide, une tâche vide est retournée (ce qui aura pour effet de bloquer le processeur virtuel).
- De plus, chaque fois qu'une tâche est ajoutée dans une liste vide (`new_state` ou `receive`) tous les processeurs virtuels qui se sont bloqués dans le module d'ordonnancement (`wait`) sont réveillés (`wake_up`).

7.3.2 Algorithme glouton

La stratégie implantée est celle présentée section 6.3.1.1 page 118 et qui consiste à maintenir les processeurs au maximum en activité. Le principe de base est de maintenir une liste de tâches prêtes, liste dans laquelle les processeurs virtuels iront chercher du travail lorsqu'ils seront inactifs. Cette liste est gérée de manière distribuée : chaque nœud maintient une liste locale dans laquelle il insère les tâches prêtes qui ont été créées sur le nœud. Lorsqu'un processeur virtuel demande une tâche à exécuter, cette tâche est prise dans la liste locale. Si cette liste locale est vide, une requête de vol est émise vers un nœud choisi arbitrairement. Ce nœud émettra alors une tâche prête ou retransmettra la requête vers un autre nœud si sa liste est également vide. L'implantation de cet algorithme au sein du module d'ordonnancement est la suivante :

– `new_task`, `receive` : si la tâche est prête à être exécutée alors elle est insérée en tête de la liste locale. Sinon, elle est ignorée.
– `new_state` : si une tâche passe à l'état prêt, elle est insérée en tête de la liste locale et les processeurs virtuels bloqués dans le module d'ordonnancement sont réveillés. Toutes les autres modifications d'état sont ignorées.
– `get_task` : la tâche située en tête de la liste est retournée (la liste est de type LIFO). Si la liste était vide, une tâche vide est retournée et une requête de vol est émise (`send`) vers un autre nœud choisi au hasard.
– `recv` : Lors de la réception d'une requête de vol, une tâche prise arbitrairement dans la liste des tâches prêtes est envoyée (`move`). Si la liste était vide, la requête est retransmise (`send`) vers un autre nœud.
– De plus, chaque fois qu'une tâche est ajoutée dans une liste vide tous les processeurs virtuels qui se sont bloqués dans le module d'ordonnancement (`wait`) sont réveillés (`wake_up`).

7.3.3 Algorithme $\mathcal{O}_1$

L'algorithme $\mathcal{O}_1$, présenté section 6.4.1 page 124, consiste à suivre au plus près l'ordre de « référence » défini sur l'ensemble des tâches. L'implantation de cet ordre est effectué de manière centralisée par le groupe qui ne prend en compte les nouvelles tâches créées qu'à la terminaison de la tâche créatrice : chaque tâche maintient donc la liste de ses tâches filles. L'implantation de cet algorithme au sein du module d'ordonnancement est la suivante :

– `new_task` : la tâche est insérée en queue de la liste des filles de la tâche créatrice.
– `new_state` : si une tâche passe à l'état terminé sur le processeur maître, la liste de ses filles vient remplacer dans la liste locale la tâche créatrice : la liste reste donc triée. Si une tâche passe à l'état terminé sur un autre processeur, la liste des filles est émise (`send`) vers le processeur maître. Si une de ces tâches est prête, les processeurs virtuels éventuellement bloqués sur le processeur maître sont réveillés. Si des processeurs avaient émis des requêtes de demande de travail qui n'avaient pas été honorées, le processeur maître tente alors de les satisfaire. De même si une tâche déjà insérée dans la liste locale sur le processeur maître passe à l'état prêt.
– `receive` : Seuls les processeurs autres que le maître peuvent recevoir des tâches. La tâche est conservée pour exécution. Cette tâche reçue est la conséquence d'une requête faite au processeur maître.
– `get_task` : sur le processeur maître, la première tâche prête de la liste locale est retournée. Sur les autres processeurs, si une tâche a été reçue (`receive`) elle est retournée ; sinon une requête de demande de travail est émise (`send`) vers le processeur maître.
– `recv` : seul le processeur maître reçoit des messages venant des autres réplicats du module. Si le message représente la fin d'une tâche, les tâches filles sont insérées

dans la liste locale. Si c'est une demande de travail, la première tâche prête de la liste est émise.

7.3.4 Algorithme $\mathcal{O}_2$

L'algorithme $\mathcal{O}_2$, présenté section 6.4.2 page 126, constitue un intermédiaire entre l'algorithme $\mathcal{O}_1$ et la stratégie gloutonne basée sur une technique de vol de travail. Cet algorithme peut être vu comme l'utilisation locale sur chaque nœud de la stratégie $\mathcal{O}_1$, les nœuds étant reliés conformément à la stratégie gloutonne par des requêtes de vol de travail. Il est à noter qu'il y a toujours un des processeurs qui suit l'ordre de « référence » $\prec_r$: à tout instant t, en notant t_i la tâche en cours d'exécution sur p_i à l'instant t, toutes les tâches inférieures à $\min_{\prec_r}(t_i)$ ont été exécutées.

Chaque processeur maintient une liste locale triée selon l'ordre de « référence » dans laquelle il puise en tête les tâches à exécuter. Lorsque qu'il rencontre une tâche non prête il abandonne cette liste et part voler la fin d'une autre liste. Lorsque la dernière tâche d'une liste est terminée, le processeur tente de poursuivre son exécution avec ce qui lui reste de la portion de liste qui lui avait été volée (et qui constitue donc les tâches suivant cette dernière tâche exécutée selon l'ordre de « référence »). L'implantation de cet algorithme au sein du module d'ordonnancement est la suivante :

- `new_task` : la tâche est insérée en queue de la liste des filles de la tâche créatrice.
- `new_state` : si une tâche passe à l'état terminé la liste de ses filles vient remplacer dans la liste locale la tâche créatrice : la liste locale reste donc triée.
- `receive` : il y a réception de toute une portion de liste de tâches suite à une requête de vol : ces tâches constituent la nouvelle liste locale.
- `get_task` : la première tâche de la liste locale est retournée. Si cette tâche n'est pas prête, il y a émission d'une requête de vol vers un processeur choisit arbitrairement. Si la liste est vide, le processeur émet une requête de vol en essayant d'abord de récupérer la fin (qui lui avait été nécessairement volée) de cette liste afin de continuer l'exécution (c'est ce qui permet de majorer le volume de mémoire nécessaire à l'exécution).
- `recv` : la réception d'une requête de récupération de liste volée est effectuée en retournant ce qui reste de la liste qui avait été volée, sauf si cette liste est celle en cours d'utilisation. Une requête de vol est traitée en retournant une portion de liste : une tâche prête est choisie arbitrairement parmi toutes les listes possédées par le site, puis toutes les tâches suivant cette tâche prête sont envoyées au processeur inactif.

7.4 Évaluations

Nous étudions dans cette section l'évaluation des quatre stratégies d'ordonnancement présentées dans la section précédente. Les performances de ces stratégies sont comparées en fonction du volume de mémoire nécessaire à l'exécution sur une machine parallèle.

Cette section est organisée comme suit : nous présentons tout d'abord l'application qui nous sert à comparer les quatre stratégies, puis nous présentons les paramètres des évaluations, puis la collecte et la synthèse des résultats et enfin nous étudions, pour chaque stratégie, le volume mémoire consommé lors d'une exécution particulière.

7.4.1 Expérimentation

L'application considérée pour comparer les différentes stratégies d'ordonnancement implantées consiste à calculer une portion de l'image de l'ensemble de Mandelbrot. Cette application a été choisie d'une part pour la simplicité du parallélisme généré (de type diviser pour paralléliser) et d'autre part pour son côté « visuel ».

Cette application, bien qu'extrêmement simple et d'intérêt restreint, nous a cependant permis d'évaluer rapidement et très simplement le comportement vis-à-vis de l'ordre d'exécution des tâches de toutes les politiques d'ordonnancement que nous implantions.

7.4.1.1 Algorithme

L'algorithme consiste en une découpe récursive de la zone à calculer. Une fois un certain seuil atteint, une tâche de visualisation, destinée à afficher la portion de l'image calculée, sera créée par zone ; le calcul des valeurs de chaque zone sera effectué également par une découpe du calcul récursive, chaque portion de zone calculée étant accumulée (après détermination de la couleur à associer à chaque pixel) dans la donnée qui sera lue par la tâche d'affichage. Le graphe de tâches correspondant à cette application est illustré figure 7.5 page 140.

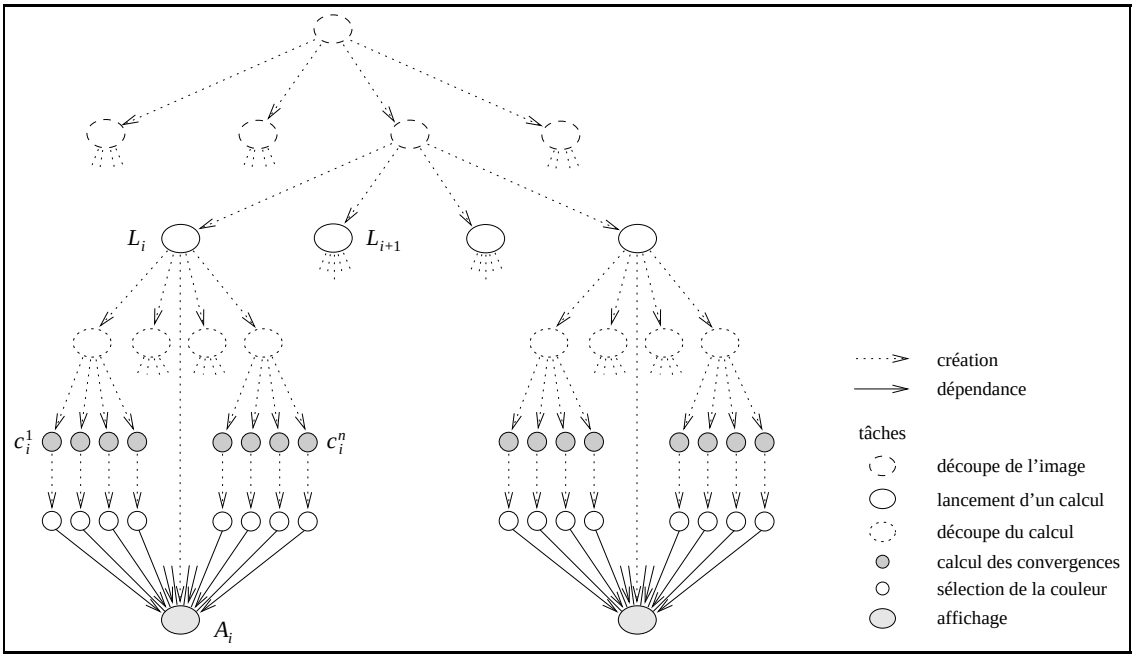

Figure 7.5 *Graphe de tâches du calcul et de la visualisation de l'ensemble de Mandelbrot.*

Nous notons dans la suite $L_1, \ldots, L_N$ les tâches de lancement de calcul, $A_1, \ldots, A_N$ les tâches d'affichages et $c_i^1, \ldots, c_i^n$ les tâches de calcul qui seront associées à la tâche A_i. Les contraintes de précédence entre ces tâches et l'ordre de « référence » sont les suivants :

$$\begin{aligned} \forall i \in [1..N], \forall j \in [1..n] \quad &: \quad L_i \prec c_i^j \prec A_i \\ \forall i \in [1..N] \quad &: \quad L_i \prec_r c_i^1 \prec_r \ldots \prec_r c_i^n \prec_r A_i \prec_r L_{i+1} \end{aligned}$$

L'espace mémoire S_1 est la somme des espaces mémoire alloués par tout ensemble $\{c_i^1, \ldots, c_i^n\}$ de tâches de calcul, cet espace mémoire étant désalloué par la tâche d'affichage A_i. Puisqu'il y a au plus N groupes de tâches de calculs :

$$S_1 \leq S_p \leq NS_1$$

7.4.1.2 Conditions d'évaluations

Toutes les évaluations effectuées dans cette section ont été effectuées sur un quadri-processeurs de type SMP (ceci permet de communiquer via la mémoire par appels de fonction, ce qui facilite considérablement l'implantation des stratégies). Les caractéristiques précises des exécutions sont rassemblées dans le tableau 7.1 page 141. La zone calculée est $(-0.07, 0.74) \times (-0.03, 0.68)$: ce choix est essentiellement le fruit du hasard, nous recherchions juste une portion de l'ensemble qui présentait certaines zones de non convergence (représentées en noir) afin d'introduire une certaine irrégularité dans la durée des tâches de calcul.

Caractéristiques des exécutions	
taille de l'image (pixels)	500×500
zone calculée	$(-0.07, 0.74) \times (-0.03, 0.68)$
nombre d'itérations maximum	100
nombre total de tâches	2405
dont tâches de découpe de l'image	$1 + 4 = 5$
dont tâches d'affichage	$4 * 4 = 16$
dont tâches de découpe de calcul	$16 * (4 * (1 + 4)) = 320$
dont tâches de calcul	$16 * 16 * 4 = 1024$
mémoire requise pour une zone	$\approx (500 * 500/1024) * 4$ o ≈ 976 o
T_1	≈ 29 s
T_∞	≈ 0.04 s
S_1	≈ 78 ko
p	de 1 à 4 IBM SP 604e 332 MHz

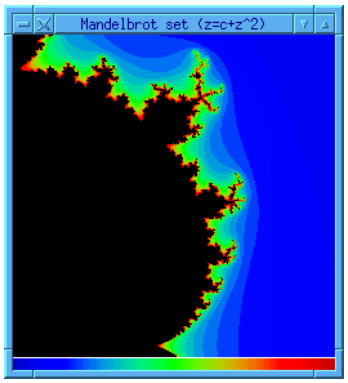

Tableau 7.1 *Conditions d'évaluation des différentes stratégies d'ordonnancement.*
Le tableau à gauche contient les caractéristiques de l'application utilisée pour comparer les différentes stratégies. Pour indication, l'image à droite représente le résultat d'une exécution.

Les courbes présentées dans les figures 7.8 page 145, 7.9 page 146, 7.10 page 147, 7.11 page 148 et 7.13 page 149 ont toutes la même légende : l'ordonnée représente le volume mémoire réservé[4] par l'application (somme des volumes alloués non encore désalloués) après chaque nouvelle action (allocation ou libération) sur cette mémoire. Ces actions sont numérotées et représentées en abscisse.

[4]Ces données ont été obtenues en définissant une version tracée des opérateurs C++ `new`, `new[]`, `delete` et `delete[]` et représentent donc toutes les allocations mémoire effectuées par Athapascan-1 et

7.4.1.3 Résultats

Synthèse Les résultats présentés dans le tableau 7.2 page 142 représentent la durée, le volume mémoire nécessaire et la nombre de messages effectués pour chacune des quatre politiques d'ordonnancement arbitraire, gloutonne, $\mathcal{O}_1$ et $\mathcal{O}_2$. Les mesures de temps ont été effectuées en désactivant le mécanisme de traçage des opérations mémoire. Le nombre de messages représente le nombre d'appels de fonction d'un processeur vers un autre (les implantations et les exécutions ont été effectuées sur un *SMP* à mémoire partagée). Les durées sont des moyennes sur 20 exécutions dont on a retiré les 2 extrêmes (les mesures sont stables). Les courbes de consommation mémoire présentées, quant à elles, ne concernent qu'une exécution particulière.

p	S_p (ko)	$\frac{S_p}{S_1}$	T_p (s)	$\frac{T_1}{T_p}$	m	S_p (ko)	$\frac{S_p}{S_1}$	T_p (s)	$\frac{T_1}{T_p}$	m
	arbitraire					**glouton**				
1	77.6	1	29.2	1	–	77.5	1	29.2	1	–
2	548.7	7.07	14.8	1.97	1200	146.0	1.88	14.6	1.99	80
3	871.7	11.23	10.0	2.91	1600	209.2	2.70	10.1	2.90	180
4	808.9	10.42	9.5	3.08	1800	284.8	3.68	9.5	3.07	240
	$\mathcal{O}_1$					$\mathcal{O}_2$				
1	78.0	1	29.2	1	–	78.2	1	29.3	1	–
2	84.5	1.08	14.9	1.96	2400	143.2	1.83	14.7	1.99	80
3	100.1	1.28	10.1	2.90	3200	189.7	2.43	10.0	2.91	180
4	105.3	1.35	9.7	3.02	3700	238.7	3.05	9.2	3.17	240

Tableau 7.2 *Évaluation numérique de différentes stratégies d'ordonnancement pour le calcul de Mandelbrot.*
Cette table est une synthèse chiffrée des différentes courbes présentées. On prend pour valeur de S_1 le volume mémoire nécessaire à l'exécution sur un processeur. Le nombre de messages engendrés m est à considérer comme un ordre de grandeur, ce nombre dépendant de l'exécution ; c'est un moyenne sur une vingtaine exécutions. $p = q$ représente le nombre de processeurs physiques (qui coïncide ici avec le nombre de processeurs virtuels d'Athapascan-1) de la machine.

Nous pouvons noter que l'accélération est proche de p pour $p \leq 3$, mais pas pour $p = 4$. Cela est dû à la présence d'un démon de communication activé par Athapascan-0 et qui vole du temps de calcul à l'application. Cela revient à dire, lorsque nous n'activons que 3 processeurs virtuels pour le module d'exécution d'Athapascan-1 ($p = 3$) qu'en fait 4 processeurs physiques peuvent être exploités du fait de la présence de ce démon. Lorsque $p = 4$ le système doit ordonnancer 5 processus légers sur les 4 processeurs de la machine, d'où la perte d'efficacité. Ce problème est largement étudié dans [20].

Nous remarquons également que les stratégies gloutonne et $\mathcal{O}_2$ semblent être légèrement plus rapide (au moins sur 2 processeurs). Ce phénomène est masqué par le fait que l'implantation a été effectuée sur une machine de type *SMP*, mais il est certainement accentué aur une machine à mémoire distribuée où la durée des messages est bien plus importante : les stratégies arbitraire et $\mathcal{O}_1$ seront donc largement désavantagées.

l'application (l'allocateur propre d'Athapascan-1 était désactivé). Les allocations effectuées par Athapascan-0 (utilisant `malloc`) ne sont donc pas comptées (mais sont constantes). De même la STL (*Standard Template Library*) n'a pas été configurée pour utiliser les opérateurs C++ `new` et `delete`. Les structures de gestion des stratégies d'ordonnancement ne sont donc pas prises en compte. Mais ce volume mémoire est fonction du nombre de tâches (la consommation mémoire de ces objets est comptée) donc ne modifie pas l'allure des courbes présentées.

Forme des courbes Les courbes présentées dans les figures 7.8 page 145, 7.9 page 146, 7.10 page 147, 7.11 page 148 et 7.13 page 149 présentent toutes un phénomème de « dents ». Ce phénomène s'explique de la manière suivante (nous considérons pour simplifier l'exécution sur un processeur d'un ordonnancement arbitraire, figure 7.8 page 145 par exemple).

Remarquons tout d'abord qu'il y a 16 dents : chacune de ces dents correspond à une tâche d'affichage A. Cette tâche d'affichage a été créée par une tâche de lancement de calcul L qui a créé 4 tâches de découpe de calcul (disons d_1, d_2, d_3, d_4). Chacune de ces tâches va donner naissance à 4 nouvelles tâches de découpe qui vont créer chacune 4 tâches de calcul de convergence (disons c_1, c_2, c_3, c_4) et 4 tâches de sélection de la couleur (disons s_1, s_2, s_3, s_4). Les contraintes de précédence impliquent que la tâche A sera exécutée après toutes ces tâches de découpe, de calcul de convergence et de sélection de couleur. En général, toutes ces tâches sont exécutées en respectant l'ordre séquentiel de création ce qui explique la régularité dans les formes des courbes. Les figures 7.6 page 143 et 7.7 page 144 illustrent ce phénomène.

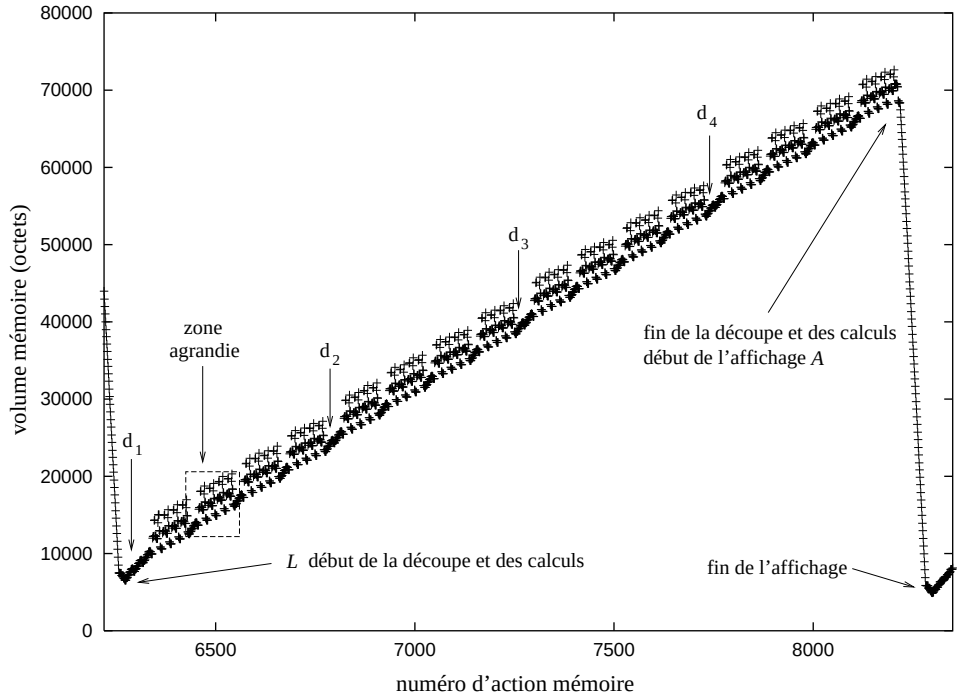

Figure 7.6 *Une « dent » de consommation mémoire.*
Cette dent correspond à l'évolution du volume mémoire nécessaire à l'application entre le début d'exécution d'une tâche de lancement de calcul et la tâche d'affichage A correspondant à ce lancement. On peut voir les 4 tâches principales de découpe et, pour chacune des tâches finales (créées par les principales), on peut distinguer les 4 tâches de calcul de convergence. La figure 7.7 page 144 est un agrandissement d'une de ces tâches finales.

La différence de pente entre les deux pans de la « dent » est due au fait que la mémoire est beaucoup plus manipulée lors de la phase de calcul que lors de la phase d'affichage : pour l'affichage, toutes les zones de calculs sont libérées les unes après les autres, tandis

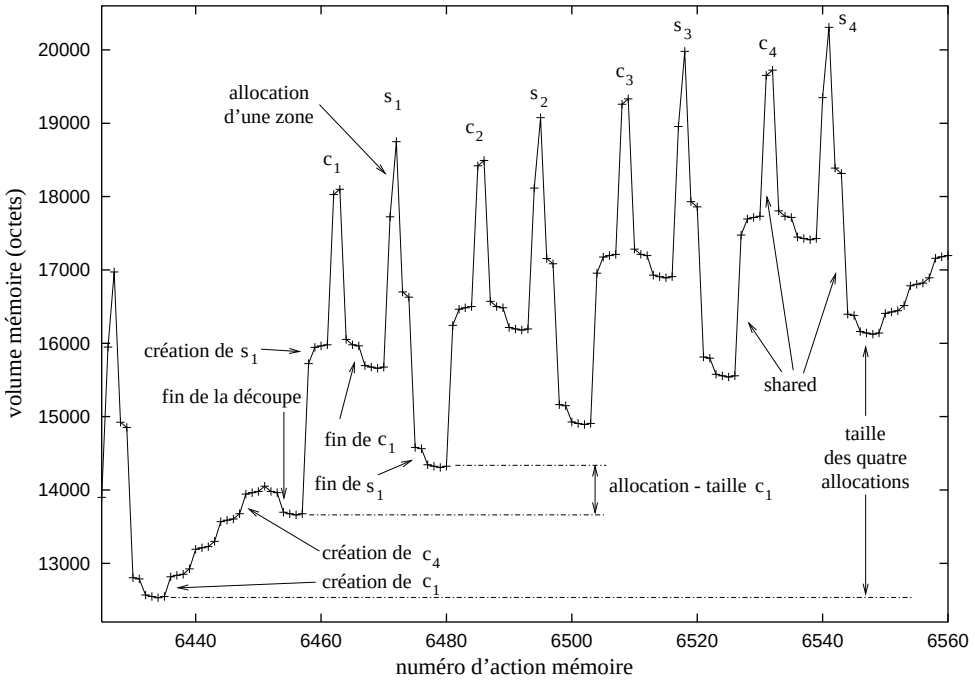

Figure 7.7 *Agrandissement de la figure 7.6 page 143 sur une tâche finale de découpe de calcul. On peut apercevoir les* 4 *tâches* c_i *de calcul de convergence et les* 4 *tâches* s_i *de sélection de couleur qui leurs sont associées.*

que dans la phase de calcul les zones sont allouées, des tâches sont créées, des tâches se terminent, des données partagées sont instanciées, *etc.*, ce qui ralentit la vitesse d'évolution de la mémoire (vitesse par rapport au numéro d'action sur la mémoire).

7.4.2 Algorithme de placement arbitraire

La figure 7.8 page 145 illustre le volume mémoire nécessaire à l'exécution de l'application considérée section 7.4.1 page 140. La durée d'exécution et le nombre de messages générés sont transcrits dans le tableau 7.2 page 142. On peut remarquer que l'utilisation de plusieurs processeurs accroît dramatiquement le volume mémoire nécessaire. Cela est dû aux exécutions tardives des tâches d'affichage qui ont pour effet de libérer la zone allouée.

Les piètres performances concernant le volume mémoire nécessaire à l'exécution s'expliquent par le fait que l'exécution des tâches d'affichage, celles qui libèrent la mémoire, peuvent être exécutées très tardivement. Par exemple, le placement suivant peut apparaître sur deux processeurs p_0 et p_1 : supposons que p_0 possède toutes les tâches de découpe de l'image, c'est donc lui qui décide du placement des tâches de lancement L_i. Il peut décider de les placer toutes sur p_1, mais à un rythme tel qu'entre deux tâches L_i reçues p_1 ait le temps d'exécuter toutes les tâches c_n^i mais pas de commencer A_i. Alors L_{i+1} sera mise en tête de la liste des tâches prêtes et A_i ne pourra être exécutée qu'après l'exécution de toutes les tâches engendrées par L_{i+1}. L'ordre d'exécution des tâches sur

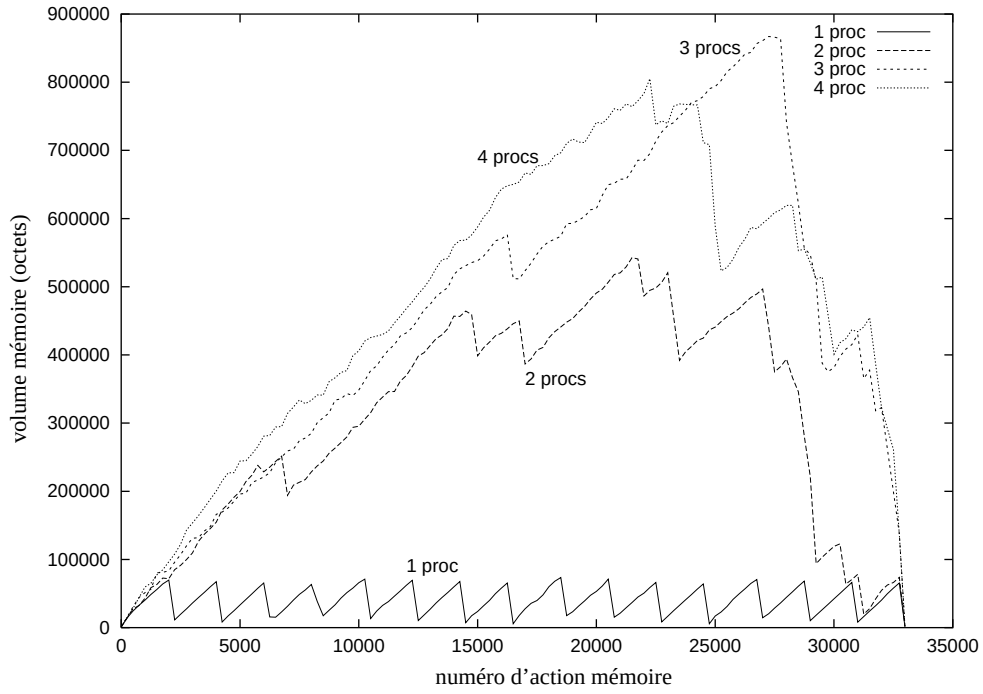

Figure 7.8 *Volume mémoire nécessaire à l'exécution selon un placement arbitraire des tâches. Les conditions d'expérimentation sont celles présentées à la section 7.4.1.2 page 141.*

p_1 est alors :

$$L_1 < c_1^1 < \ldots < c_1^n < L_2 < \ldots < L_N < c_N^1 < \ldots < c_N^n < A_N < A_{N-1} < \ldots < A_1$$

Cet ordre d'exécution mène à une consommation mémoire $S_p = NS_1$ qui est le maximum que cette application puisse consommer.

Le nombre de messages engendrés est également important : un message par tâche avec une probabilité de $1 - \frac{1}{p}$. De plus, aucune notion de localité entre les tâches est respectée, comme en témoigne la figure 7.14 page 150.

7.4.3 Algorithme glouton

La figure 7.9 page 146 illustre le volume mémoire nécessaire à l'exécution de l'application présentée section 7.4.1 page 140. La durée d'exécution et le nombre de messages générés sont transcrits dans le tableau 7.2 page 142. On peut remarquer que l'utilisation de plusieurs processeurs accroît le volume mémoire nécessaire d'une manière semble-t-il linéaire. Le nombre de messages engendrés est faible, ce qui traduit une bonne propriété de localité de l'algorithme.

Cependant, cette stratégie peut mener à un ordonnancement des tâches tel que la consommation mémoire soit maximale, c'est-à-dire $Sp = NS_1$. En effet, soit une exécution sur 3 processeurs, p_0, p_1 et p_2 telle qu'au moins une phase de découpage du calcul intervienne entre les tâches L_i et les tâches c_i^j. Supposons que p_0 possède toutes les tâches L_i dans sa liste de tâches prêtes. Il exécute en premier L_N (la liste est gérée de manière

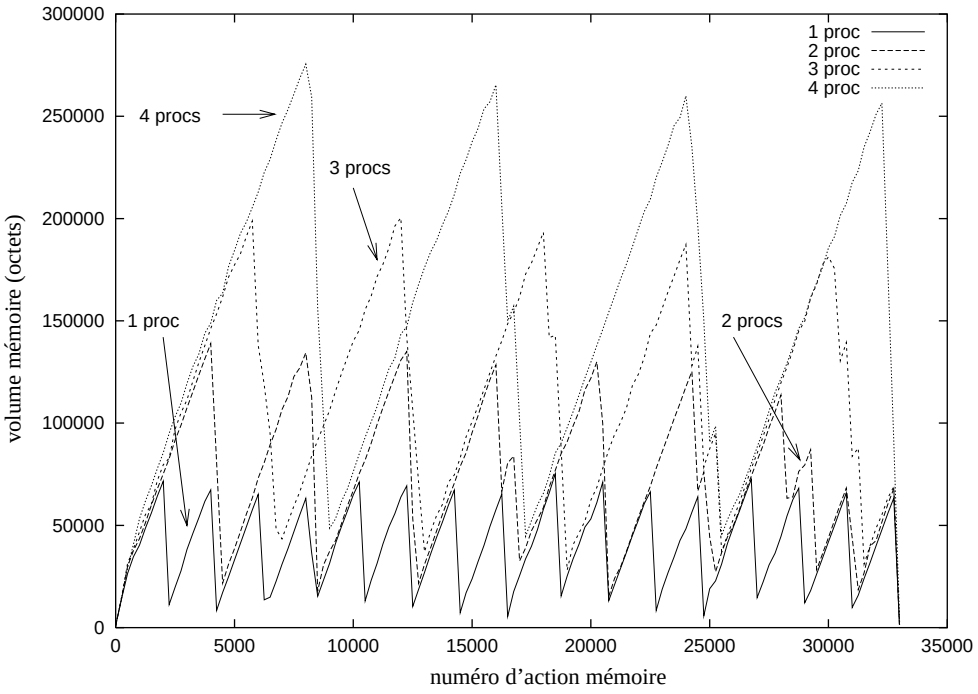

Figure 7.9 *Volume mémoire nécessaire à l'exécution selon un algorithme glouton. Les conditions d'expérimentation sont celles présentées à la section 7.4.1.2 page 141.*

LIFO). p_1 vole une tâche de découpe de calcul à p_0 et p_2 vole une tâche de calcul, disons c_1^1 à p_1. Compte tenu des contraintes de précédence entre les tâches, la tâche A_N ne pourra s'exécuter que lorsque l'exécution de c_1^1 aura été terminée et que p_0 en aura été averti. Or il se peut que p_2 ait plus de mal à contacter p_0 qu'à voler des tâches à p_1 (les liens de communications ne sont pas forcément les mêmes). Ce scénario pouvant être répété pour toute tâche L_i, il vient que p_2 va bloquer l'exécution sur p_0 de toutes les tâches A_i en possédant toutes les tâches c_i^1. Le volume mémoire nécessaire à cette exécution est donc égal à NS_1.

7.4.4 Algorithme $\mathcal{O}_1$

La figure 7.10 page 147 illustre le volume mémoire nécessaire à l'exécution de l'application présentée section 7.4.1 page 140. La durée d'exécution et le nombre de messages générés sont transcrits dans le tableau 7.2 page 142. On peut remarquer que l'utilisation de plusieurs processeurs n'accroît que très peu le volume mémoire nécessaire. Cela est dû au fait que l'ordre d'exécution des tâches suit au plus près l'ordre de « référence ». L'implantation centralisée de cet ordre sur un processeur maître génère un nombre important de messages : deux messages (un pour l'exécution distante et un pour notifier la terminaison) par tâche avec une probabilité de $1 - \frac{1}{p}$.

Compte tenu du goulot d'étranglement que représente la centralisation du calcul de l'ordre de « référence » sur un seul processeur maître, cette stratégie ne peut être envisagée que sur un nombre relativement restreint de processeurs. Une autre solution est de

distribuer le calcul de l'ordre. C'est ce que fait la stratégie $\mathcal{O}_2$ suivante.

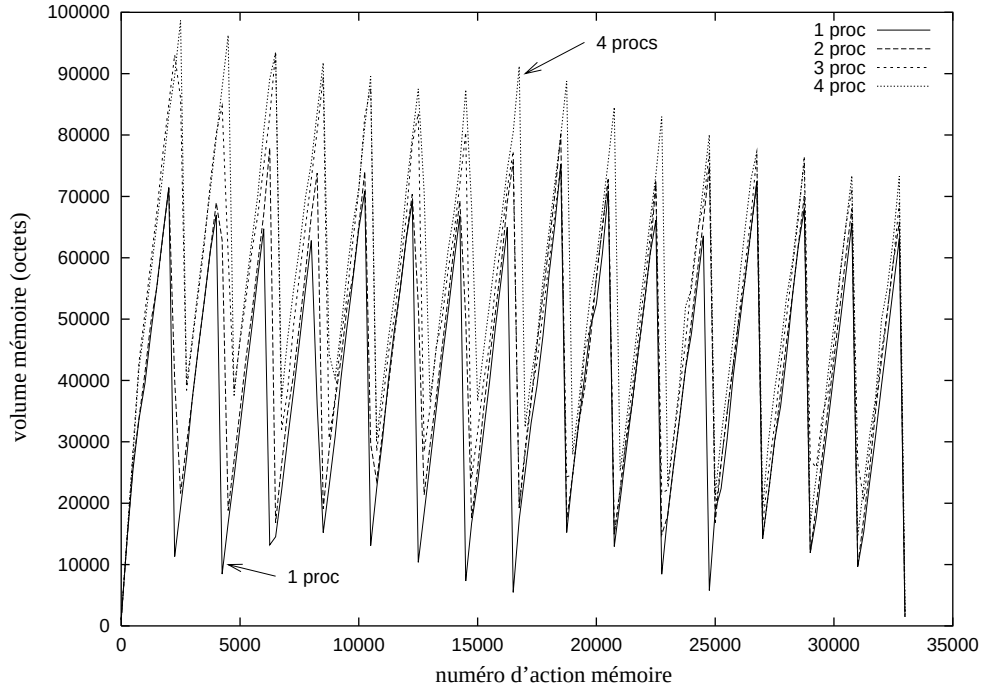

Figure 7.10 *Volume mémoire nécessaire à l'exécution selon l'algorithme $\mathcal{O}_1$.*
Les conditions d'expérimentation sont celles présentées à la section 7.4.1.2 page 141.

7.4.5 Algorithme $\mathcal{O}_2$

La figure 7.11 page 148 illustre le volume mémoire nécessaire à l'exécution de l'application présentée section 7.4.1 page 140. La durée d'exécution et le nombre de messages générés sont transcrits dans le tableau 7.2 page 142. On peut remarquer que l'utilisation de plusieurs processeurs accroît le volume mémoire nécessaire mais dans la limite de pS_1. Le nombre de messages engendrés est faible, ce qui traduit une bonne propriété de localité de l'algorithme.

7.4.6 Comparaison

La figure 7.12 page 148 trace la mémoire consommée par chacune des stratégies en fonction du nombre de processeurs. On peut remarquer que les stratégies $\mathcal{O}_1$ et $\mathcal{O}_2$ respectent les bornes théoriques préditent, à savoir $S_1 + ...$ et pS_1.

La figure 7.13 page 149 retrace, avec cette fois une même échelle pour les ordonnées, le volume mémoire utilisé lors de l'exécution sur 3 processeurs suivant la stratégie d'ordonnancement utilisée. Cette courbe permet de constater à quel point l'ordre d'exécution des tâches (calculé par la stratégie d'ordonnancement) influence le volume de mémoire nécessaire à l'exécution.

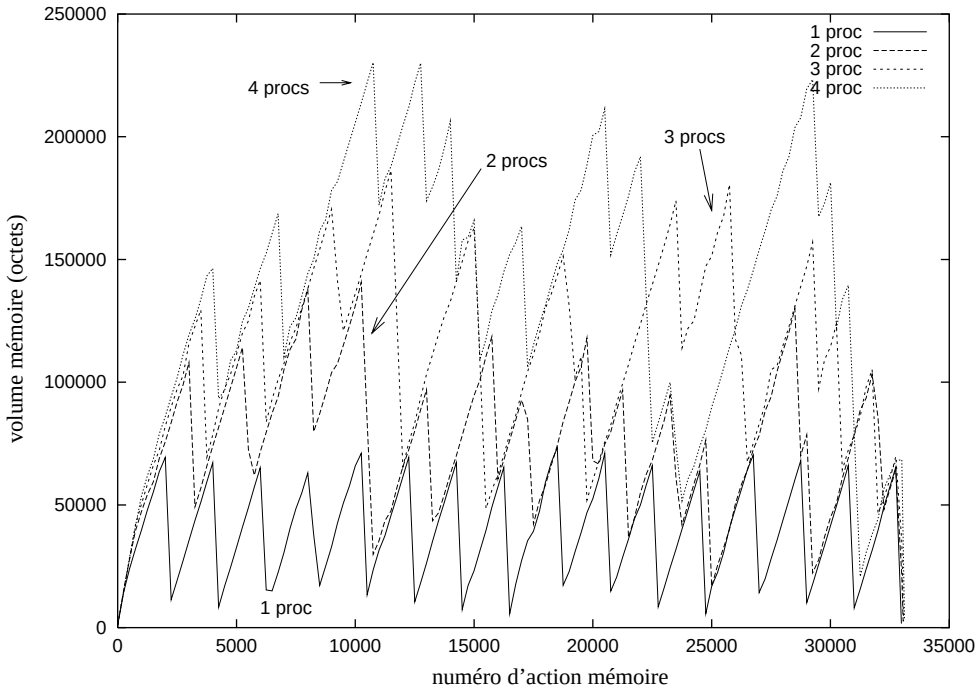

Figure 7.11 *Volume mémoire nécessaire à l'exécution selon l'algorithme $\mathcal{O}_2$.*
Les conditions d'expérimentation sont celles présentées à la section 7.4.1.2 page 141.

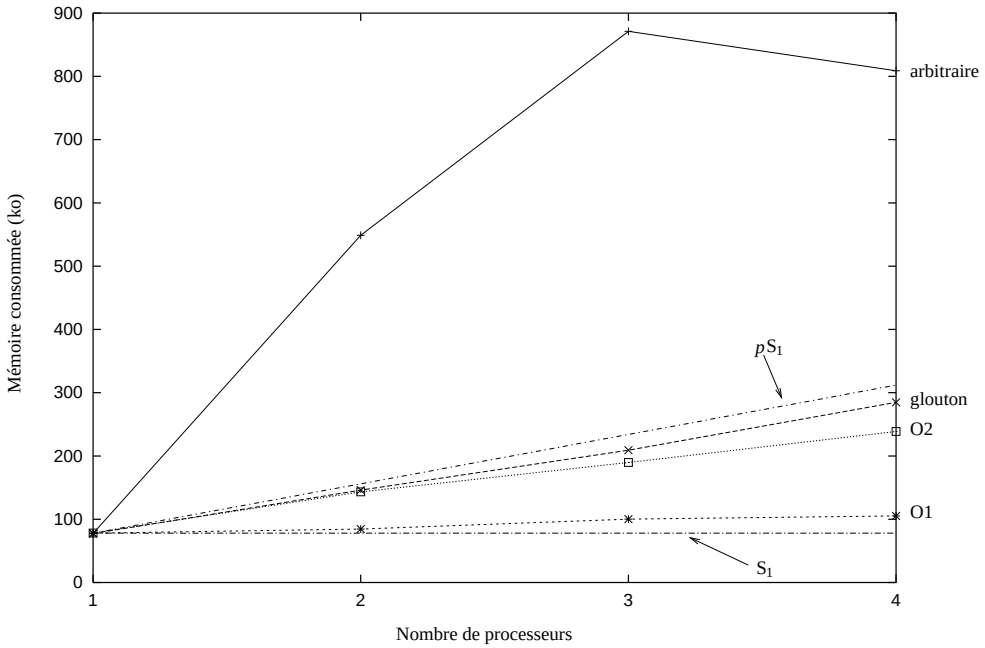

Figure 7.12 *Consommation mémoire en fonction du nombre de processeurs et de la stratégie d'ordonnancement utilisée.*
Les courbes S_1 et pS_1 sont particulières et utilisées dans les majorations théoriques.

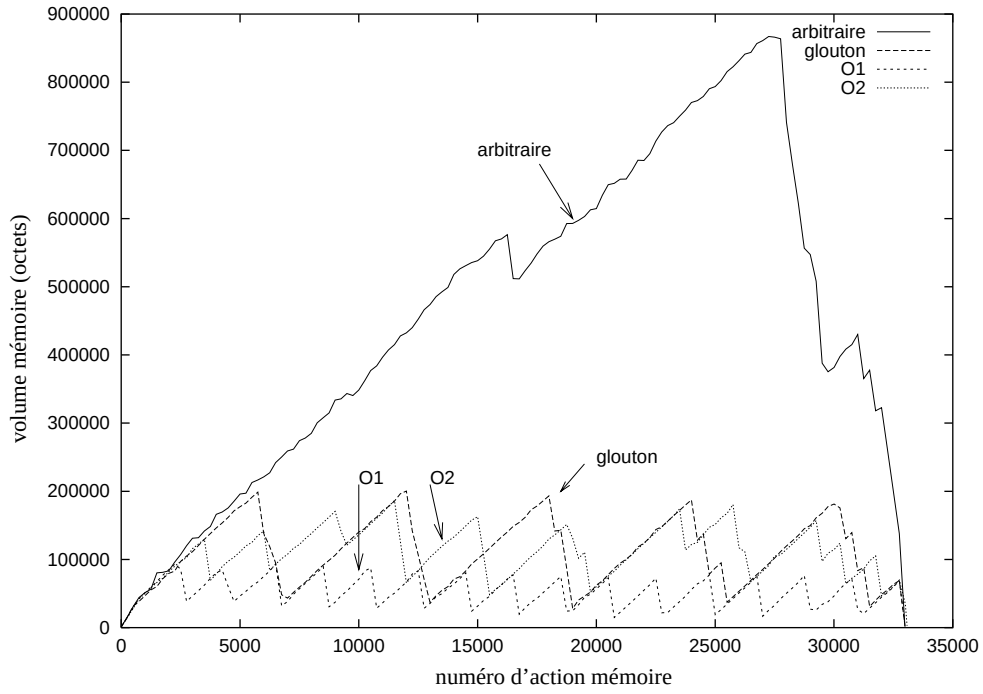

Figure 7.13 *Comparaison du volume mémoire nécessaire à l'exécution sur* 3 *processeurs selon la politique d'ordonnancement.*

La figure 7.14 page 150 illustre pour chacune des politiques d'ordonnancement, sur une exécution particulière effectuée sur 2 processeurs, l'ordre de calcul des différentes zones de l'image.

La première chose que l'on remarque est la répartition des couleurs : pour les stratégies arbitraires et $\mathcal{O}_1$ les zones de couleurs uniformes sont petites, tandis que pour les deux autres stratégies ces zones sont de plus grosse taille. Cela est dû à la propriété de localité de ces deux dernières : lorsqu'une tâche est exécutée sur un nœud, il y a de forte chance pour que toute sa fratrie soit également exécutée sur ce même nœud (à moins qu'il y ait eu vol). Cette propriété n'est pas vraie pour les stratégies arbitraires et $\mathcal{O}_1$. Ceci est directement corrélé au nombre de messages qui est plus important, tableau 7.2 page 142. En effet, une tâche correspond à un message dans le cas de ces stratégies tandis que pour les deux autres un message correspond à un gros bloc.

Une seconde chose à remarquer est la forme de la zone blanche, c'est-à-dire la zone non encore calculée : dans le cas arbitraire cette zone est relativement décousue, tandis que pour les trois autres cette zone reste compacte : la consommation mémoire est directement liée à ce critère.

Une dernière remarque concerne la différence entre la stratégie gloutonne et la stratégie $\mathcal{O}_2$. Ces deux stratégies se ressemblent fortement, mais la différence fondamentale est la suivante : dans le cas de la politique $\mathcal{O}_2$ il y a toujours un processeur virtuel qui exécute une tâche selon l'ordre de référence, ce qui n'est pas le cas pour la première. C'est cette différence qui permet de majorer le volume mémoire nécessaire à l'exécution dans le cas

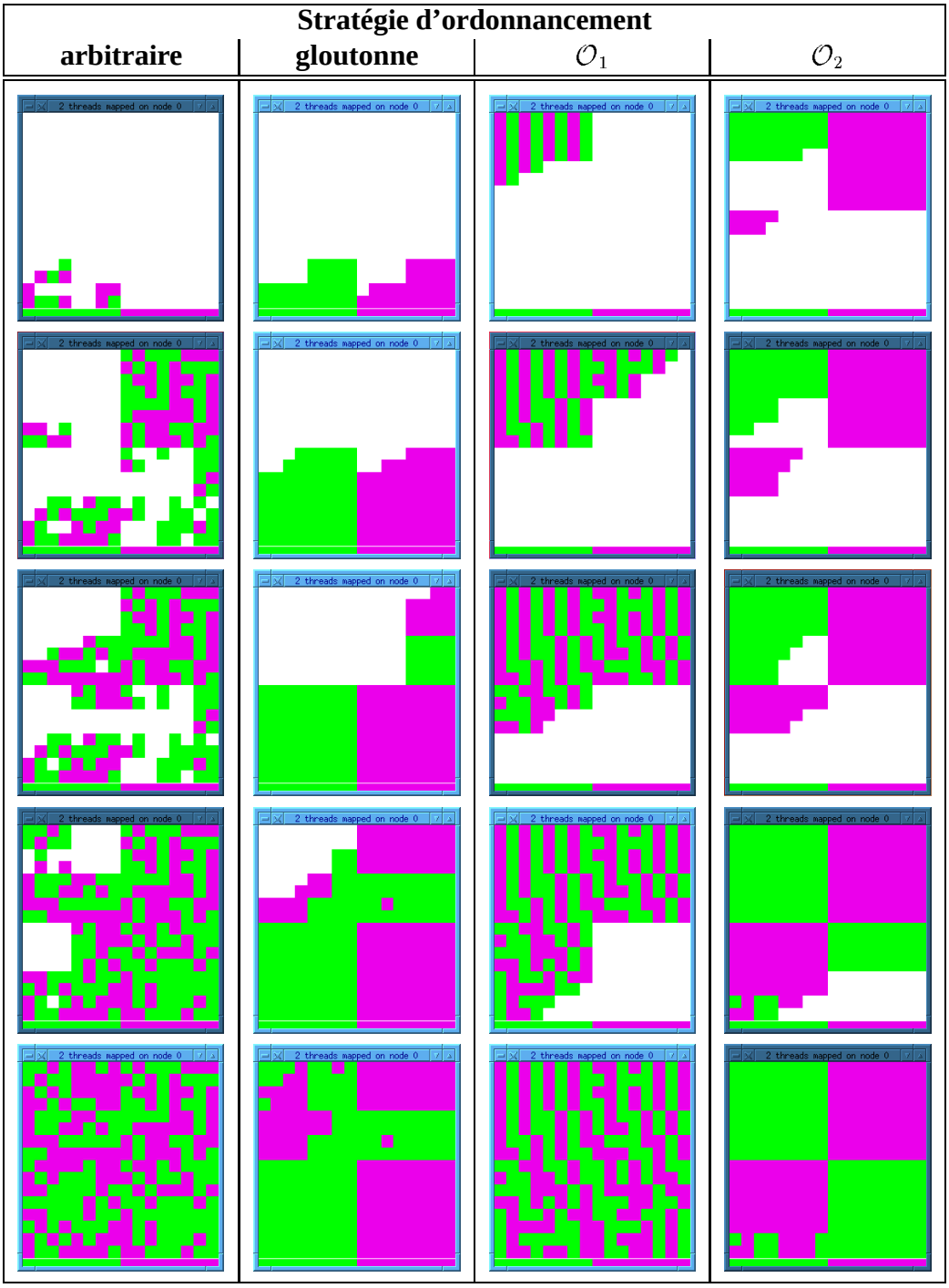

Figure 7.14 *Visualisation de l'ordre d'évaluation des tâches pour les différentes stratégies d'ordonnancement.*
L'exécution a eu lieu sur deux processeurs. Sont colorées les zones de l'image qui ont été calculées (une couleur par processeur) : le temps s'écoule de haut en bas. Les figures sur une même horizontale n'ont aucun lien particulier.

d'$\mathcal{O}_2$ contrairement à la stratégie gloutonne.

7.5 Conclusion

Nous avons étudié dans ce chapitre l'implantation et l'évaluation dans Athapascan-1 de deux politiques d'ordonnancement permettant un contrôle de la consommation mémoire.

L'ordonnancement est assuré par un module séparé de la bibliothèque Athapascan-1. Ce module facilite l'implantation d'une grande variété de stratégies, ce qui permet d'adapter la politique d'ordonnancement à la situation traitée (application, machine, contraintes mémoire, *etc.*).

Les évaluations effectuées en comparaison avec deux autres stratégies, simples mais ne contrôlant pas l'utilisation mémoire, qui sont un placement arbitraire des tâches et une stratégie gloutonne, montrent que la consommation mémoire des deux stratégies proposées sont conformes aux prédictions annoncées, comme illustré figure 7.12 page 148. Cependant, le nombre de messages généré (dû au caractère centralisé de l'algorithme) par la politique $\mathcal{O}_1$ qui est la plus performante en mémoire restreint son utilisation à une machine possédant peu de processeurs et un réseau rapide. En relâchant certaines contraintes sur l'ordre d'exécution des tâches, c'est-à-dire en ne suivant plus d'aussi près l'ordre de « référence », il est possible d'implanter une version distribuée de cet algorithme : $\mathcal{O}_2$.