# Chapitre 6

# Tâches, partitionnement et ordonnancement

## 6.1 Introduction

Un point essentiel pour obtenir des performances avec un ordinateur parallèle est d'utiliser des algorithmes adaptés au modèle de programmation et à l'architecture sous-jacente. Tout algorithme parallèle repose sur un ensemble de tâches qui traitent un ensemble de données. L'exécution concurrente de ces tâches par plusieurs processeurs implique certains choix quant à la manière de diviser un problème en sous-problèmes, et pour savoir sur quel processeur chacun sera assigné et à quel moment il devra démarrer. Ces questions sont directement liées à ce que l'on appelle le partitionnement, le placement et l'ordonnancement des tâches. Selon ces choix on aura des performances plus ou moins bonnes. Le but de ce chapitre est de mettre ces problèmes en évidence et de les discuter plus en détail.

## 6.2 Calcul de l'ensemble de Mandelbrot

Pour commencer cette discussion, nous allons illustrer le problème du découpage d'un problème en sous-problèmes dans le cas du calcul de l'ensemble mathématique de Mandelbrot.

Pour construire cet ensemble, il faut itérer la relation $z_{n+1} = z_n^2 + c$ où $z_i$ et $c$ sont des nombres complexes, et regarder pour quelles valeurs de $c$, $z_n \to \infty$ pour $n$ assez grand. En pratique, on itère cette relation jusqu'à ce que $\|z_n\|$ dépasse un seuil fixé (par exemple, la valeur 100) et on associe la valeur de $n$ au $c$ choisi.

On prend généralement $z_0 = 0$ et on considère un ensemble de $N$ valeurs $c = c_x + ic_y$ également réparties dans une région prédéfinie du plan complexe. En représentant, pour chacune des valeurs de $c$ une couleur correspondant au nombre d'itérations nécessaires pour dépasser le seuil choisi, on génère une image

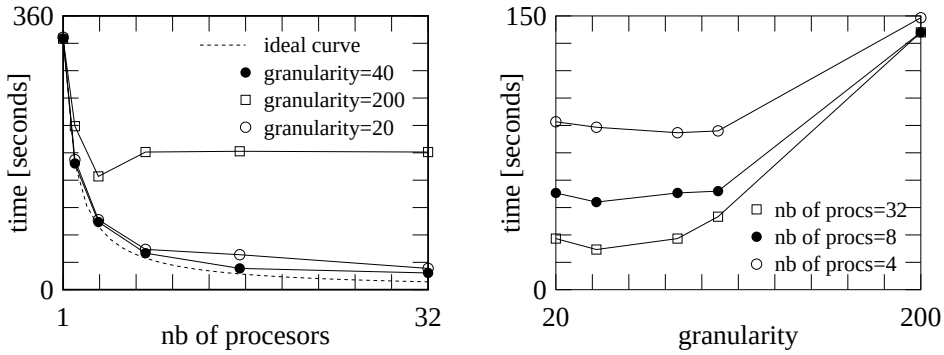


FIGURE 6.1 – *Performances obtenues pour le calcul de l'ensemble de Mandelbrot, en fonction de la granularité de la tâche et du nombre de processeurs. (Cas de la machine Archipel de Volvox, à 32 processeurs, 1992.)*

complexe, très spectaculaire, appelée ensemble de Mandelbrot.

Si l'on dispose d'une machine parallèle, on peut itérer simultanément la relation $z_{n+1} = z_n^2 + c$ pour toutes les valeurs de $c$ considérées. A première vue, ce problème est de type SIMD puisque, pour chaque valeur de $c$, on effectue la même opération. Cependant, le nombre d'itérations à effectuer est différent d'un point à l'autre : une fois que l'on a dépassé le seuil fixé pour un $c$ donné, on arrête le calcul. Dès lors, pour éviter d'avoir des processeurs inactifs, il est nécessaire qu'une fois son travail terminé, un processeur soit réaffecté à une autre valeur de $c$.

Ceci est possible avec une machine MIMD. Toutefois, le gain de performance que l'on observe n'est pas toujours optimum non plus, comme le montre la figure 6.1.

Les mesures ont été effectuées en faisant varier le nombre de processeurs collaborant au calcul de l'ensemble de Mandelbrot. Chaque processeur reçoit comme tâche le calcul d'une portion de l'espace des valeurs de $c$. Lorsqu'un processeur a terminé, un gestionnaire de tâches (ici une station de travail centrale) réalloue cette ressource de calcul à une nouvelle portion de l'image. Chaque tâche est caractérisée par la taille de ces portions d'image. On appelle ceci la **granularité** de la décomposition du travail et on la représente par un nombre $l$ donnant la taille en pixels de chaque bloc d'image. Ainsi, une granularité de 40 signifie que chaque processeur reçoit un carré de $40 \times 40$ pixels à calculer. Ce calcul est illustré sur la figure 6.2.

Les courbes obtenues sur la figure 6.1 montrent que les performances dépendent crucialement de la granularité des tâches et que le gain maximum n'est jamais obtenu (idéalement, on espère que le temps d'exécution d'une machine comprenant $N$ processeurs est inférieur d'un facteur $N$ à celui d'une machine ne comprenant qu'un seul de ces processeurs). Bien que ce problème n'implique aucune communication de données entre les tâches (et constitue ainsi un cas de

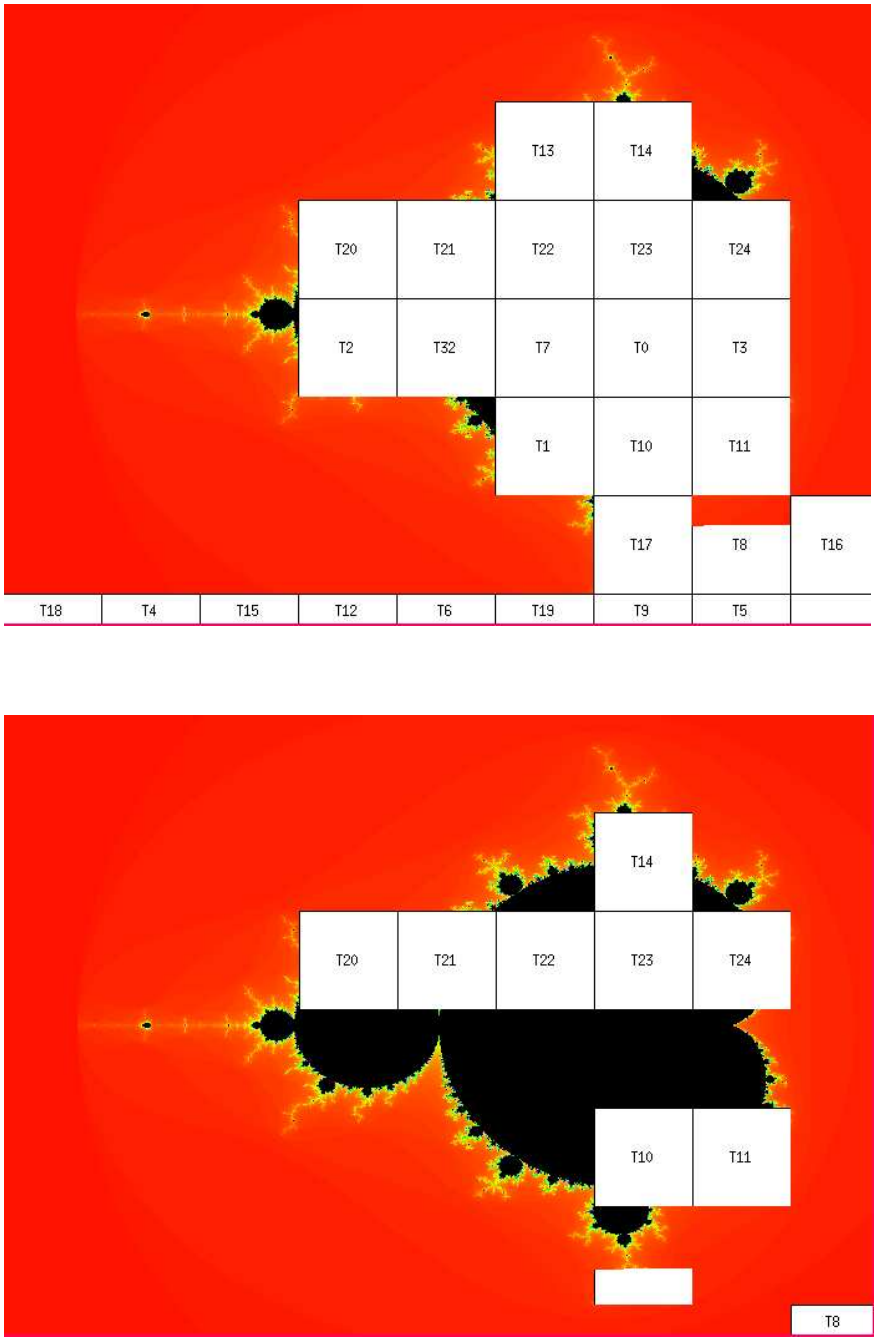


FIGURE 6.2 – *Calcul de l'ensemble de Mandelbrot avec la machine Archipel. Chaque carré désigne la portion de l'espace alloué au processeur indiqué (T0, T1,...,T32). La figure montre le calcul à deux étapes distinctes, illustrant la réaffectation dynamique des processeurs à de nouvelles tâches.*

parallélisme idéal), une saturation est observée, au delà de laquelle il est inutile d'ajouter encore d'autres processeurs. Ces résultats s'expliquent de la manières suivante :

1. Granularité trop fine : Le gestionnaire des tâches passe son temps à réaffecter les processeurs à de nouvelles régions du plan. Le travail de chacun est trop court et la réallocation devient le "goulet d'étranglement" de l'application. Le système n'arrive même pas toujours à utiliser tous les processeurs disponibles (en particulier lorsque très peu d'itérations par point sont suffisantes).
2. Granularité trop grossière : Comme toutes les tâches ne sont pas de la même durée, un petit nombre de processeurs se voit allouer des régions de l'espace nécessitant un long travail, alors que les autres terminent très rapidement le reste de l'image, sans pouvoir être réutilisés. En d'autres termes, une petite fraction des processeurs disponibles font effectivement le travail et on ne gagne rien à ajouter de nouveaux processeurs dans le calcul. Il y a donc une granularité idéale pour ce problème, et selon le graphique, on voit qu'elle vaut environ 40.

Cet exemple suggère plusieurs solutions pour améliorer la vitesse d'exécution et s'approcher de la courbe de performance idéale. On pourrait découper le plan des valeurs de $c$ de manière inégale en un nombre de sous-régions identiques au nombre de processeurs utilisés, de sorte que chaque sous-région contienne la même quantité de travail. On aurait ainsi un équilibre des charges. Cette répartition n'est pas toujours facile à deviner et il peut être nécessaire d'exécuter le programme une fois afin de se rendre compte de la bonne décomposition. Dans le cas de l'ensemble de Mandelbrot, cela peut être une bonne approche au problème car, souvent, on désire en un deuxième temps zoomer une partie plus intéressante de l'image et on peut utiliser les calculs précédents pour améliorer la rapidité d'exécution de la suite du travail.

Une autre solution pour augmenter les performances de calcul est de resubdiviser dynamiquement les tâches les plus longues : lorsqu'il n'y a plus de travail pour un processeur, on peut essayer d'en enlever un peu à celui qui en aurait trop et de le redistribuer. Cela a l'avantage de ne pas nécessiter la connaissance des tâches les plus longues à l'avance, mais le désavantage d'avoir une gestion compliquée et intelligente de la répartition du travail. On reviendra plus loin sur certaines méthodes d'équilibrage dynamique des charges dans le paragraphe 6.7

## 6.3 Tâches et notion de dépendance

De façon très générale, un programme parallèle peut être vu comme un ensemble de tâches **concurrentes** dont certaines peuvent être exécutées simultanément alors que d'autres doivent être exécutées dans un ordre donné.

Formellement, une tâche $T_i$ est une suite d'opérations exécutée en séquentielle par un seul processeur. Par définition elle ne contient donc aucune communication avec d'autres processeurs. Une tâche est par ailleurs caractérisée par

1. Un ensemble $E_i$ de variables d'entrée qui sont nécessaires à l'exécution de la tâche.
2. Un ensemble $S_i$ de variables de sortie qui sont modifiées lors de l'exécution.

Une tâche peut être vue comme un opérateur de l'ensemble de ses entrées sur l'ensemble de ses sorties. On peut associer à chaque tâche un temps d'exécution $t_i$.

Deux tâches $T_i$ et $T_j$ sont dites **indépendantes** si elles ne modifient pas leur variables communes :

$$S_i \cap S_j = E_i \cap S_j = E_j \cap S_i = \{\}$$

Des tâches indépendantes peuvent être exécutées en parallèle. Si elles ne sont pas indépendantes, il faut préciser leur ordre d'exécution. Si $T_i$ doit être exécutée avant $T_j$, on note $T_i < T_j$.

La tâche $T_j$ est **consécutive** à $T_i$ s'il n'existe aucune autre tâche $T_k$ devant être exécutée entre $T_i$ et $T_j$.

Les dépendances entre tâches impliquent une relation dite de **précédence** entre elles qui permet la construction d'un graphe de tâches : un arc relie la tâche $T_i$ à la tâche $T_j$ si $T_j$ est consécutif à $T_i$. Le graphe de précédence donne une représentation d'un algorithme et constitue un outil de base pour en discuter la parallélisation.

## 6.4 Partitionnement

**Le partitionnement** est la division du problème à résoudre en sous-tâches qui seront assignées à des processeurs différents.

En général, le partitionnement n'est pas unique et la granularité idéale n'est pas connue à l'avance. C'est souvent un problème de compromis. Plus le problème peut être découpé en petits morceaux, plus le parallélisme à disposition sera grand (ce qui est favorable à l'obtention de performances). Par contre, si les tâches sont trop courtes, l'overhead de parallélisation (communications, synchronisations) devient important et les performances globales chutent. Le partitionnement est donc un problème d'optimisation qui revient à chercher le découpage en tâches qui minimise le temps total d'exécution. Ce problème n'est pas simple en général car l'overhead dépend de la manière dont le problème est partitionné.

Cette situation est illustrée dans la figure 6.3, pour le calcul de la somme de $N$ nombres sur $p$ processeurs, avec $N \geq p$. Ce calcul se réalise en répartissant $N/p$ valeurs par processeur. La somme se fait tout d'abord en additionnant ces $N/p$ valeurs sur chaque processeur, afin d'obtenir $p$ valeurs sur $p$ processeurs. Sur

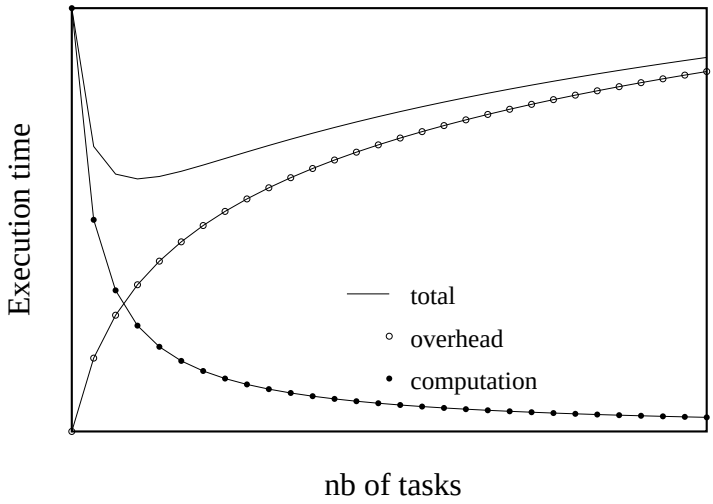


FIGURE 6.3 – *Temps d'exécution en fonction du nombre de tâches. Plus celui-ci est grand, plus le degré de parallélisation est grand et plus l'overhead de parallélisation devient important.*

une architecture en hypercube, il faut ensuite $\log_2 p$ communications et calculs pour sommer les $p$ nombres placés sur les $p$ processeurs. On peut donc séparer le temps total d'exécution en $N/p$ pour le temps de calcul proprement dit et $\log_2 p$ pour l'overhead de parallélisation. Pour $N$ fixé, le temps de calcul diminue si $p$ augmente alors que l'overhead augmente.

## 6.5 Placement et ordonnancement

**Le placement** est le choix de quel processeur exécutera quelle tâche. On cherche à distribuer le travail parmi les processeurs pour garantir qu'un maximum d'entre eux soient utilisés à tout moment, tout en minimisant l'overhead de communication. Le placement est un problème difficile même dans des cas simples : supposons qu'on ait des tâches $T_1$, $T_2$,...,$T_n$ à distribuer sur $p$ processeurs. Même si ces tâches sont indépendantes, mais de durées différentes, une répartition qui optimise l'équilibrage de charge est difficile à trouver.

**L'ordonnancement** détermine le moment où chaque tâche débutera. Une fois que le partitionnement et le placement sont décidés, il faut encore assurer que les différentes tâches s'exécutent dans le bon ordre ou encore qu'elles n'attendent pas inutilement longtemps avant de démarrer. Une tâche ne peut débuter que si toutes celles dont elle dépend sont achevées. Comme il est probable que plusieurs de ces tâches soient sur des processeurs différents, il faut des mécanismes de synchronisation inter-processeurs. Il faut notamment éviter les situations d'interblocage (*deadlock*) dues au fait que deux tâches (ou plus) attendent mutuellement

leur achèvement respectif avant de pouvoir se terminer.

Les critères qui définissent un bon placement et ordonnancement sont évidemment les performances obtenues. Il faut en particulier diminuer l'overhead du parallélisme et obtenir un bon **équilibre des charges** (ou **load balancing**) sur tous les processeurs.

Le choix d'un bon partitionnement, placement et ordonnancement pour une application donnée est en général un problème d'optimisation combinatoire NP-complet. En d'autres termes, il n'existe pas d'algorithme simple qui le résolve rapidement. Heureusement, la nature de l'application considérée offre souvent des solutions heuristiques acceptables.

## 6.6 La méthode des temps «au plus tôt» et «au plus tard»

Dans ce paragraphe, nous allons illustrer le problème du placement et de l'ordonnancement par un exemple simple pour lequel les temps de communications inter-processeurs sont négligeables (parallélisation à gros grain). De plus, nous supposerons connu le temps d'exécution de chaque tâche, ce qui correspond à une situation d'ordonnancement statique car ces temps ne changent pas durant l'exécution du programme.

La méthode dite des temps au plus tôt et au plus tard constitue une technique permettant d'analyser le graphe de précédence d'une application et de construire le placement et l'ordonnancement des tâches (qui n'est pas forcément l'optimum absolu).

Pour illustrer cette technique, considérons l'exemple décrit dans la figure 6.4. Les différentes tâches sont notées $T_i$ et les indications à côté de chaque tâche représentent la durée d'exécution. On commence par déterminer le premier moment où une tâche pourra débuter. Cela s'obtient en parcourant le graphe de haut en bas. La tâche $T_1$ est supposée commencer au temps $t = 0$. Au plus tôt, les tâches $T_2$ et $T_3$ pourront débuter au temps $t = 1$, soit dès que $T_1$ est terminée.

De même, les tâches $T_4$ et $T_5$ ne peuvent pas commencer avant que $T_2$ soit achevée, c'est-à-dire au plus tôt au temps $t = 3$. En poursuivant ce raisonnement, la dernière tâche $T_9$ ne pourra commencer qu'au temps $t = 14$. Donc, l'exécution optimum de ce programme sera telle que $T_9$ débutera à $t = 14$. Si $T_9$ débute plus tard, les performances ne seront pas les meilleures et il faut essayer d'éviter cela. Dans ce but, on cherche maintenant le dernier moment où chaque tâche peut commencer afin de garantir que $T_9$ démarre effectivement au bon moment. On observe ainsi que $T_7$ doit commencer au plus tard 7 secondes avant le temps optimal de $T_9$, soit au plus tard au temps $t = 7$. De même, la tâche $T_8$, ne durant que 4 secondes, doit débuter au plus tard au temps $t = 14 - 4 = 10$. En remontant de la sorte jusqu'à $T_1$ on détermine la table des temps « au plus

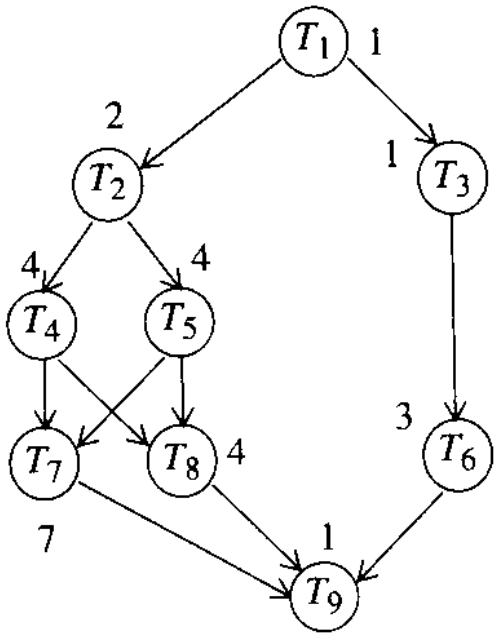


FIGURE 6.4 – *Graphe de précédence d'une application composée de plusieurs tâches (Moldovan, p. 385).*

tôt »et « au plus tard »indiquant le début de chaque tâche, comme le montre le tableau 6.1 (Moldovan, p. 385).

Comme on le voit, certaines de ces tâches ont ces deux temps identiques ($T_4$, $T_7$, par exemple), ce qui signifie qu'elles doivent impérativement débuter à un moment précis. Ce temps est d'ailleurs le même pour $T_4$ et $T_5$ ce qui indique qu'il faudra au moins deux processeurs si l'on ne désire pas introduire des délais supplémentaires et garder le rythme fixé.

Pour établir le placement final (et l'ordonnancement) correspondant à cette situation, on construit la table 6.2 dont chaque ligne correspond à une unité de temps (ici, on aurait 15 lignes, pour les temps $t = 0$ jusqu'à $t = 14$). Les colonnes correspondent aux processeurs à disposition. On place dans cette table les tâches dont le temps d'exécution est bien défini (earliest_ time=latest_ time), en partant de la première colonne. Si celle-ci est pleine, on remplit la colonne suivante, ce qui signifie qu'un autre processeur devient nécessaire. Une fois que toutes les tâches sont placées (on place en priorité les tâches pour lesquelles la différence latest_time-earliest_ time est minimum), le nombre de colonnes remplies indique le nombre de processeurs nécessaires. Ici, il en faut deux. On constate aussi que le premier processeur est utilisé à 100% alors que l'autre l'est 12 secondes sur 15 (soit 80%). Il y a donc un léger déséquilibre de charge.

Le speedup maximum qu'on peut espérer pour cette situation est facile à ob-

| Task | Earliest Time | Latest Time |
|---|---|---|
| 1 | 0 | 0 |
| 2 | 1 | 1 |
| 3 | 1 | 10 |
| 4 | 3 | 3 |
| 5 | 3 | 3 |
| 6 | 2 | 11 |
| 7 | 7 | 7 |
| 8 | 7 | 10 |
| 9 | 14 | 14 |

TABLE 6.1 – *Table des temps au plus tôt et au plus tard pour le graphe de la figure précédente.*

tenir de notre analyse : $T_9$ termine après 15 secondes avec 2 processeurs (ce qui est l'optimum pour cette décomposition en 9 tâches). Avec un seul processeur, le temps total serait la somme des temps de chaque tâche, soit 27 secondes. Le speedup maximum est donc 27/15=1.8, indépendamment du nombre de processeurs à disposition (cf loi d'Amdahl : le speedup est limité par la quantité de parallélisme à disposition).

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | $T_1$ | $T_2$ | $T_2$ | $T_4$ | $T_4$ | $T_4$ | $T_4$ | $T_7$ | $T_7$ | $T_7$ | $T_7$ | $T_7$ | $T_7$ | $T_7$ | $T_9$ |
| $P_2$ | idle | $T_3$ | idle | $T_5$ | $T_5$ | $T_5$ | $T_5$ | $T_6$ | $T_6$ | $T_6$ | $T_8$ | $T_8$ | $T_8$ | $T_8$ | idle |

TABLE 6.2 – *Placement et ordonnancement des tâches. Deux processeurs suffisent. Les cases marquées "idle" signifie que le processeur concerné est inactif.*

## 6.7 Equilibrage de charge

Comme on l'a vu déjà plusieurs fois, un déséquilibre de charge entre les processeurs est une cause importante de perte d'efficacité dans une application parallèle. On se souvient (paragraphe 4.2) que le speedup est donné par le **degré de parallélisme moyen**. S'il y a un fort déséquilibre de charge (load unbalance), certains processeurs seront inactifs et le degré de parallélisme diminuera. Une telle situation est illustrée dans la section 6.2, pour le calcul de l'ensemble de Mandelbrot. On constate dans cet exemple que seuls un petits nombre de processeurs travaillent pendant la durée complète de l'exécution. Le travail réalisé par de nombreux processeurs est immédiat et ne contribue pas à apporter un speedup intéressant. Dans une telle situation, il faut trouver des stratégies pour

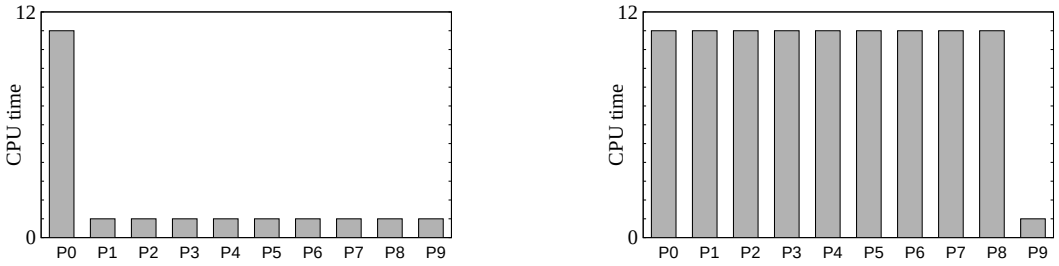


FIGURE 6.5 – Deux situations de déséquilibre de charge, avec les même temps $T_{par}$, mais un déséquilibre de nature très différente. A gauche, la situation est très mauvaise car presque tous les PE attendent sur le plus lent. A droit, seul un processeur attend. Le nombre de cycles perdus est bien moindre.

mieux partitionner le domaine de calcul ou bien il faut dynamiquement réallouer des parts de l'image aux processeurs qui ont terminé. Dans ce cas précis, un partitionnement basé sur un découpage cyclique (ou modulo) du domaine est une bonne solution.

La figure 6.5 illustre deux cas de déséquilibre de charge, caractéréisés par des temps d'exécution parallèles $T_{par}$ identiques, déterminés par le processeur le plus chargé. On se rend bien compte que la situation de gauche est très mauvaise car un grand nombre de cycles machine sont gaspillés par les 9 PE qui attendent $P_0$. Par contre, dans l'image de droite, la situation est bien moins dramatique car un seul processeur est trop peu chargé.

### 6.7.1 Métrique de déséquilibre de charge

Il est donc important de proposer des métriques qui mesurent le déséquilibre de charge et dont la valeur reflétent la sévérité de la situation.

Une mesure de l'équilibre de charge est donnée par la valeur de l'efficacité $E$ lorsque les communications et autres sources d'overhead peuvent être négligées. Une efficacité de 100% implique alors un équilibre parfait alors qu'une efficacité de 20% signifie que, globalement, seuls 20% des processeurs ont travaillé.

Cela peut se voir directement par le raisonnement suivant. Soit $T_{seq}$ le temps d'exécution séquentiel du programme. Idéalement, le temps d'exécution parallèle devrait être $T_{seq}/p$, si chaque processeur hérite d'une portion identique du travail. On peut donner comme mesure de l'équilibre de charge, l'écart à 1 du rapport entre ce temps idéal et le temps du processeur le plus chargé (qui est d'ailleurs aussi le temps $T_{par}$ de l'exécution parallèle)

$$\Delta = 1 - \frac{(T_{seq}/p)}{T_{par}} = 1 - \frac{T_{seq}}{pT_{par}} = 1 - E$$

où $E$ est l'efficacité définie dans l'équation (5.5). La grandeur $\Delta$ ainsi définie varie entre 0 (pas de déséquilibre) et 1 (déséquilibre maximum). Dans le cas de la figure 6.5, en supposant que $T_{seq}$ est la somme de temps de chaque PE, on a respectivement pour les images de gauche et de droite, les valeurs

$$\Delta = 1 - 20/(10 \times 11) = 0.818 \qquad \Delta = 1 - 100/(10 \times 11) = 0.09$$

ce qui reflète bien la différence des deux cas.

Il y a de nombreuses autres mesures, qui quantifient le déséquilibrage de charge indépendamment d'une référence au temps séquentiel, ce qui est raisonnable sachant qu'il y a en général un overhead de parallélisation (en particulier les communications).

Par exemple si $T_i$ dénote le temps de calcul du processeur $P_i$ pour réaliser sa part d'un travail $W_{par}$ donné, la variance des $T_i$

$$V = \frac{\sum_{i=1}^{p}(T_i - \mu)^2}{p} \in [0, \infty]$$

est une mesure du déséquilibre, où

$$\mu = \frac{1}{p}\sum_{i=1}^{p} T_i$$

est la moyenne des temps $T_i$ et $p$ le nombre de processeurs. On notera que $\mu = T_{moy}$ donne le temps CPU d'une situation parfaitement équilibrée.

Si la variance $V$ est nulle, l'équilibrage est parfait. Mais plus $V$ est grand, plus le déséquilibre est important. Cependant, cette métrique n'est pas toujours très selective. Sur la figure 6.5 de gauche on a $T_{moy} = \mu = 2$ et $V = \mathrm{Var(T_i)} = 9.0$. A droite on a $T_{moy} = \mu = 10$ mais aussi $V = \mathrm{Var(T_i)} = 9.0$. La variance des temps d'exécution ne détecte donc pas la sévérité du déséquilibre de charge. Une amélioration serait de normaliser $V$ par $\mu$.

Une autre mesure de déséquilibre de charge qu'on va utiliser par la suite est donnée par

$$u = m - \mu \in [0, \infty] \tag{6.1}$$

où

$$m = \max_i(T_i) \qquad \text{et, comme avant,} \qquad \mu = T_{moy} = \frac{1}{p}\sum_{i=1}^{p} T_i$$

Une valeur nulle de $u$ indique que $\max(T_i) = T_{moy}$, ce qui exprime un équilibrage parfait. A l'opposé, une valeur élevée indique un déséquilibrage substantiel. Pour les cas de la figure 6.5, on obtient respectivement

$$u = 11 - 2 = 9 \qquad u = 11 - 10 = 1$$

ce qui indique bien que le premier déséquilibre est plus grave que le deuxième.

En supposant comme ci-dessus qu'on peut négliger l'overhead dû aux communications, on peut poser que $m = \max_i(T_i) = T_{par}$ et $\mu = T_{moy} = (\sum_i^p T_i)/p = T_{seq}/p$. Il en résulte que $u$ vaut dans ce cas

$$u = T_{par} - T_{seq}/p = T_{par}\,(1 - E) = \Delta \times T_{par}$$

Dans le reste de ce chapitre, on va considérer différentes situations de déséquilibre de charge et présenter des stratégies de répartition du travail permettant d'assurer que chaque processeur reste occupé le plus longtemps possible. Nous commencerons par le cas statique et étudierons ensuite le cas dynamique, plus compliqué. Il n'y a pas de méthode universelle qui résoud tous les cas. Souvent, le problème d'équilibrage de charge revient à résoudre un problème d'optimisation difficile pour lequel seules des heuristiques sont utilisables. Par conséquent, la suite de ce chapitre a seulement pour but d'illustrer les situations courantes et de proposer des solutions possibles.

## 6.8 Equilibrage de charge statique

.

Dans ce paragraphe, on suppose que le temps de calcul de chaque partitionnement possible est calculable et qu'il ne varie pas durant l'exécution du programme.

Le problème de l'équilibrage des charges est alors un problème de partitionnement et de placement qui peut, en principe, se résoudre avant exécution. La méthode des temps au plus tôt et au plus tard présentée au paragraphe 6.6 est un exemple de méthode de solution pour les problèmes à granularité plutôt grossière et un graphe de tâche irrégulier. Ci-dessous, nous décrivons d'autres approches liées à la recherche d'une répartition des données équitables.

Le cas le plus courant dans un programme SPMD est que le partitionnement est équivalent à un découpage du domaine de calcul en sous-domaines ("domain decomposition", en anglais). Souvent le temps de calcul est proportionnel au nombre de données du sous-domaine. Pour avoir équilibrage de charge, il suffit donc d'avoir un découpage en sous-domaines de taille égale. Une première difficulté est que souvent le domaine ne se divise pas exactement par le nombre de processeurs, à l'instar de l'exemple donné au paragraphe 5.2.

Mais si on néglige ce problème d'arithmétique, la prochaine difficulté est que, souvent, le domaine de calcul est géométriquement irrégulier. On peut penser à un plan de ville où les calculs ne doivent se faire que dans les régions entre les bâtiments. C'est notamment le cas lorsqu'on étudie comment le vent souffle dans les rues d'une ville ou encore si l'on modélise le trafic des véhicules.

### 6.8.1 «Space filling curves»

Il est en principe possible de numéroter tous les points du domaine de calcul, de les parcourir séquentiellement et d'en attribuer un nombre égal à tous les processeurs. Le parcours séquentiel d'un domaine en deux ou trois dimensions est un problème standard de l'informatique. On le fait couramment lorsqu'on décrit un tableau 2D par un indice unique qui parcourt les lignes du tableau les unes après les autres.

Un tel mapping s'exprime facilement si on considère un domaine à deux dimensions, par exemple de taille $2^b \times 2^b$ dont les coordonnées $(x, y)$ sont exprimées en binaires

$$x = x_b x_{b-1} \dots x_1 \qquad y = y_b y_{b-1} \dots y_1$$

On peut construire

$$z = y_b y_{b-1} \dots y_1 x_b x_{b-1} \dots x_1 \tag{6.2}$$

par concaténation des expressions de $y$ et $x$. Si on parcourt $z$ de 0 à $2^{2b} - 1$ et qu'on ré-exprime $x$ et $y$ à partir de $z$ on traverse le domaine ligne par ligne.

Mais il est souvent souhaitable de choisir un parcours qui préserve au mieux la localité des données au sens que des données proches dans la représentation $z$ soient aussi proches dans les coordonnées $(x, y)$. Cela permet d'exploiter au mieux la mémoire cache et de créer des domaines ayant un meilleur rapport surface sur volume dans le cas d'un partitionnement sur plusieurs processeurs. Un parcours linéaire qui «remplit» efficacement un domaine en $D$-dimensions est appelé en anglais une *Space filling Curve*.

La courbe dite **de Morton** est une modification de l'équation (6.2) qui permet un parcours unidimensionnel dans domaine à $d$-dimention et qui préserve mieux la localité que le choix précédent. En deux dimensions, la courbe de Morton est construite en alternant les bits de $y$ et ceux de $x$

$$z = y_b x_b y_{b-1} x_{b-1} \dots y_2 x_2 y_1 x_1 \tag{6.3}$$

Le tableau suivant illustre cette construction pour un domaine $4 \times 4$.

| | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 0000 | 0001 | 0100 | 0101 |
| 01 | 0010 | 0011 | 0110 | 0111 |
| 10 | 1000 | 1001 | 1100 | 1101 |
| 11 | 1010 | 1011 | 1110 | 1111 |

La figure 6.6 montre le parcours ainsi obtenu sur des domaines plus grands. On observe l'invariance d'échelle du parcours à mesure que le domaine croît en taille. On constate aussi que, le plus souvent, la localité spatiale est préservée.

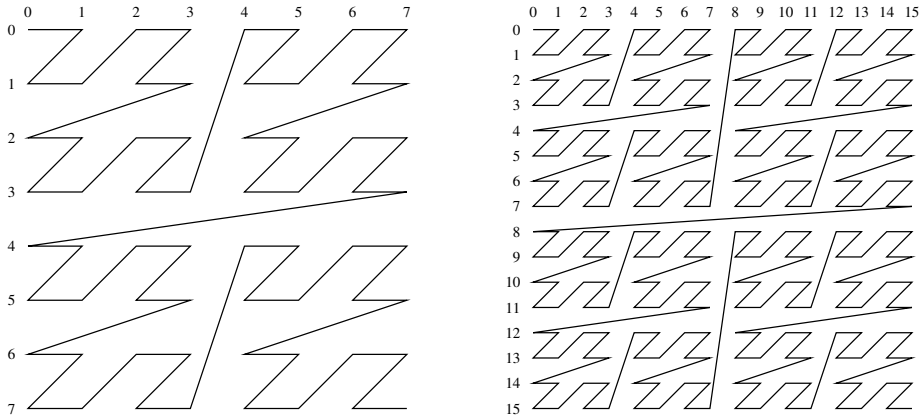


FIGURE 6.6 – Le parcours correspondant à la courbe dite de Morton dont le mapping est donné par l'équation (6.3).

Cependant, un des exemples le plus célèbre de «space filling curve» est donné par la **courbe de Hilbert** [1] qui est illustrée sur la figure 6.7. On notera cependant que la relation (voir https ://en.wikipedia.org/wiki/Hilbert_curve) entre la position sur la courbe et la coordonnée $(x, y)$ correspondante n'est pas si simple que dans les exemples précédents. Cette relation est aussi difficile à inverser.

On peux aussi utiliser une courbe de Hilbert pour partitionner un domaine irrégulier. Sur la figure 6.8 un partitionnement équilibré des points à calculer en quatre sous-domaines est indiqué par les quatre couleurs. L'irrégularité du domaine est ici obtenu en supposant que la résolution de la grille de calcul varie en certains points. En conséquence, sur cet exemple, il y a 238 points à répartir sur quatre processeurs, à raison d'environ de 60 points pour chacun d'eux.

### 6.8.2 Bissection récursive

Une autre façon de partitionner un domaine de calcul est d'utiliser la technique dite de **bissection récursive** qui consiste à diviser par une ligne (ou un plan) le domaine de calcul en 2 parties contenant chacune un travail identique et de procéder récursivement sur chacun des morceaux ainsi obtenus, en altérnant les découpage dans les $d$ dimensions du domaine. On obtient alors un partionnement contenant une puissance de 2 de sous-domaines. Cette procédure est illustrée sur la figure 6.9 où l'on voit un domain 2D irrégulier dont une partie est exclue du calcul (le rectangle gris). Ce domaine est itérativement divisé en deux parties de taille égale (au mieux de ce que la géométrie permet), par des découpes verticales alternant avec des découpes horizontales. L'exemple de la figure comprend au final quatre étapes et produit ainsi 16 sous-domaines. La tailles des sous-domaines est

1. `https://en.wikipedia.org/wiki/Hilbert_curve`

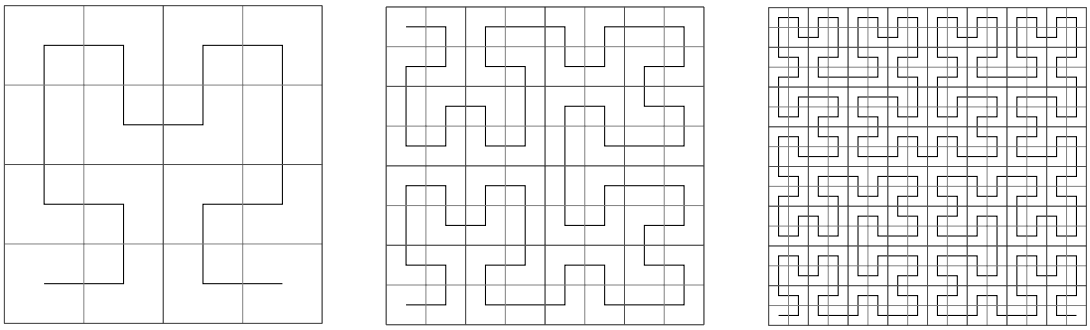

FIGURE 6.7 – Courbe de Hilbert qui parcourt un domain de calcul à deux dimensions, dont la résolution $m \times m$ augmente de gauche à droite, avec $m = 4, 8, 16$, respectivement. Le mapping entre l'indice $i$ qui parcourt la courbe de Hilbert et les points $(x, y)$ du domaine est donné sur `https://en.wikipedia.org/wiki/Hilbert_curve`. Ce mapping suppose que le nombre total de points est une puissance de 2.

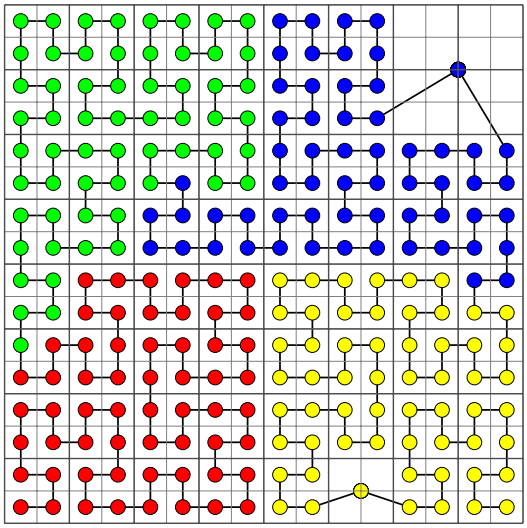

FIGURE 6.8 – Courbe de Hilbert dans un domaine dont la résolution varie d'une région à l'autre. Il y a dans cet exemple 238 points à calculer. Ce nombre est ici divisé en 4 pour réaliser une partition équilibrée sur 4 processeurs. Les couleurs indiquent l'appartenance des point à un même processeur.

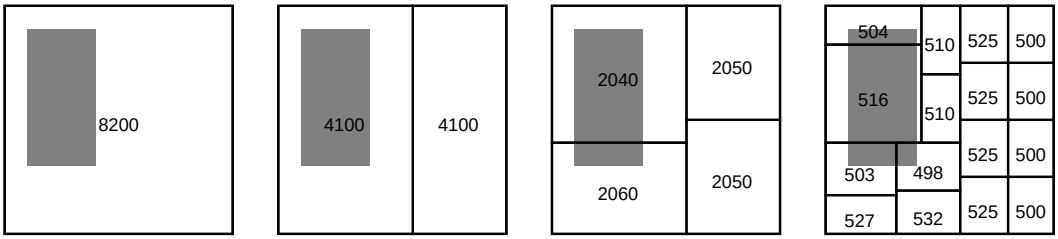


FIGURE 6.9 – Exemple de bissection récursive sur un domaine de calcul irrégulier, caractérisé par le rectangle grisé dans lequel aucun calcul n'est nécessaire. Le nombre total de point à calculer est 8200, au lieu de 10'000 pour le domaine complet. De gauche à droite, on voit l'effet des découpages successifs et la taille des sous-domaines obtenus,

donnée par les nombres indiqués sur la figure. Il y a des variations de tailles inévitables mais globalement le déséquilibre sera moindre que si on avait découpé le domaine de façon régulière.

### 6.8.3 Partitionnement de graphe

La méthode de la bissection récursive génèrent souvent des sous-domaines dont les longueurs de frontière sont très inégales. Donc, il n'y a pas de garantie que le découpage ainsi obtenu soit optimal en terme de communications, d'autant que certains processeurs auront peut-être leurs sous-domaines voisins éclatés parmi beaucoups d'autres processeurs.

De façon générale, le problème de construire des partitions de tailles identiques dont les frontières sont les plus petites possibles est équivalent à celui du **partitionnement de graphe**. Chaque point de calcul est connecté à d'autres selon les spécificités de la méthode numérique utilisée (p. ex. les 4 voisins nord, sud, est, ouest dans la résolution de l'équation de Laplace). Cela permet de représenter le domaine de calcul selon un graphe que l'on cherche ensuite à partitionner en sous-graphes, contenant tous le même nombre de sommets (pour obtenir l'équilibrage de charge) et minimisant le nombre d'arcs qui relient les sous graphes entre eux (pour avoir des communications efficaces). Des bibliothèques comme *Metis*, accessible sur le web, permettent d'obtenir des partitionnement de domaines irréguliers très satisfaisant. La figure 6.10 illustre le résultat obtenu par Metis pour le découpage d'un domaine irrégulier en huit morceaux

Finalement, dans le cas où aucune communication inter-processeur n'est nécessaire, une technique très simple peut-être utilisée, même si on ne connait pas les temps de calculs associés à chaque point du domaine. On peut alors utiliser une répartition des données **cyclique** (ou modulo), comme discutée au paragraphe 7.3. Dans ce partitionnement, chaque processeur contient des points provenant de

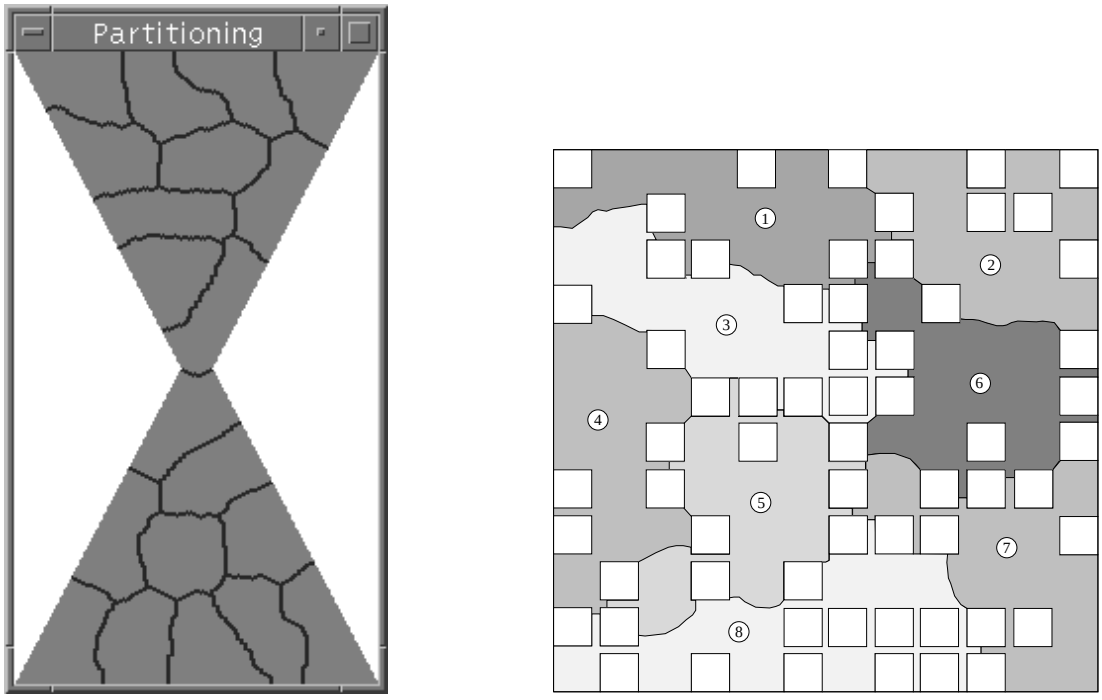


FIGURE 6.10 – *Partitionnement (avec Metis) de domaines irréguliers en sous-domaines afin d'équilibrer les charges et minimiser les communications. A droite, les carrés blancs indiquent les régions absentes du domaine de calcul. (Images : Alexandre Dupuis.)*

toutes les régions du domaine de calcul et, statistiquement, cela assure que le travail et bien réparti. Dans le cas de la recherche de motif dans une image, cette approche simple est souvent efficiace, à condition que chaque processeur ait une copie complète de l'image.

## 6.9 Equilibrage de charge dynamique

### 6.9.1 Problématique

Le cas de l'équilibrage de charge dynamique est plus compliqué car il se fait en cours d'exécution. Il est nécessaire lorsqu'on ne sait pas estimer les temps de calcul associés aux différentes tâches du problème, ou lorsque le partionnement doit être modifié dans un calcul itératif car la durée des tâches changent d'une itération à l'autre.

On peut illustrer une telle situation par le problème du trafic routier. Statiquement, on peut découper un réseau routier de façon que chaque processeur ait la même longueur de route ou le même nombre d'intersections. Mais le calcul du trafic dépend aussi du nombre de véhicules sur chaque tronçon de route, ce qui change au cours du temps et aussi en fonction des bouchons qui peuvent potentiellement se former, obligeant alors les véhicules à choisir d'autres parcours.

Les surcharges possibles induites dans un processeur par un autre utilisateur sont un autre facteur qui cause un déséquilibre de charge temporel dans une système parallèle multi-utilisateur.

Par rapport au cas statique, il y a maintenant de nouveaux enjeux :

— Il faut **détécter** un possible déséquilibre de charge lors de l'exécution. Il y a donc une partie du programme qui doit mesurer les charges respectives de chaque processeur ou les temps de calculs entre chaque points de synchronization.

— Il faut **décider** selon un critère donné si le déséquilibre de charge mesuré doit être traité ou non. Il peut être trop faible pour justifier un repartitionnement qui sera forcément coûteux en performance et pourrait masque le benéfice attendu. Ou bien, le déséquilibrage de charge est fluctuant est il n'est pas clair comment le corriger de façon durable.

— Finalement, il faut un mécanisme de **migration** de tâches ou de **re-répartition** des données entre les processeurs qui permet de diminuer le déséquilibre mesuré. Ceci s'obtient en appliquant les partitionnements proposés dans la section 6.8.

Pour chacun des trois éléments ci-dessus, on peut choisir une **stratégie globale** (ou centralisée) qui fait jouer un rôle particulier à un processeur donné qui se charge de comparer les temps d'exécution, de les analyser selon certains critères heuristiques et ensuite de repartitionner tout le calcul. C'est la solution la plus simple car toute l'information est rassemblée en un seul processeur. Mais c'est

aussi la moins scalable et celle qui va créer le plus de congestion.

Une **stratégie locale** est aussi envisageable pour diminuer les coût d'overhead de la détection et du traitement du déséquilibre de charge. Dans une telle situation, les proceseurs se comparent à leur voisins immédiats et décident entre eux qui est le plus chargé. Comme ces processeurs doivent de toute façon communiquer aux points de synchronization, l'overhead d'échanger en plus un temps de calcul ainsi qu'éventuellement une partie du sous-domaine de calcul reste petit. Malheureusement, une telle stratégie peut être lente à converger si un processeur a des contraintes contradictoires entres ses differents voisins. De plus, si la charge varie dans le temps plus vite que l'algorithme ne converge, il n'y aura jamais d'équilibre de charge global.

### 6.9.2 Exemples dans le cas itératif

A titre d'exemple de cette approche, la figure 6.11, gauche illustre un cas de résolution sur 4 processeurs de l'équation de la chaleur sur un domaine irrégulier, dû à des régions (les rectangles blancs) où la température est imposée et pour lesquelles aucun calcul n'est nécessaire.

L'équilibrage dynamique de la charge s'obtient par un mouvement progressif des frontières entre les sous-domaines, au cours des premières itérations du calcul. Dans le cas présent, ceci est facile à mettre en oeuvre car à chaque itération, les processeurs ont besoin des valeurs de températures des colonnes du bord des sous-domaines voisins. Il est donc possible, lors de la communication de ces colonnes d'échanger aussi la charge des processeurs respectifs. Cette charge est simplement estimée par le nombre de points de grille associés à chaque processeur. Si l'un de ces processeur a une charge inférieure à sont voisin, il gardera la colonne que ce dernier lui aura transmise dans la phase de communication, agrandissant ainsi son sous-domaine au profit de son voisin.

Sur la figure 6.11 (gauche) on voit, en couleur, le champ de température après convergence du calcul. On voit aussi le partitionnement initial en quatre sous-domaines de taille égale (lignes blanches verticales) et le partitionnement final (lignes noires) Sur la partie droite de la figure, on voit l'évolution des 3 frontières séparant les 4 sous-domaines, selon l'algorithme d'équilibrage dynamique décrit ci-dessus. On constate une convergence rapide de la position idéale de ces frontières. Pour ce résultat, il faut que la tolérance de différence de travail entre deux sous-domaines soit au moins de 100 sites, correspondant à la taille d'une colonne. Si on veut forcer un écart plus faible, les frontières se mettront à osciller comme le montre la figure 6.12 où un échange de colonne se fait dès que la différence de taille des sous-domaines adjacents dépasse 50 points de grille.

Notons qu'on peut évidemment combiner les approches globales et locales ce qui, selon les problèmes, permet d'exploiter le meilleur de chacune des stratégies.

On va maintenant présenter un exemple dans lequel le domaine de calcul s'agrandit au cours du temps. Il s'agit ici de calculer le front d'injection d'un

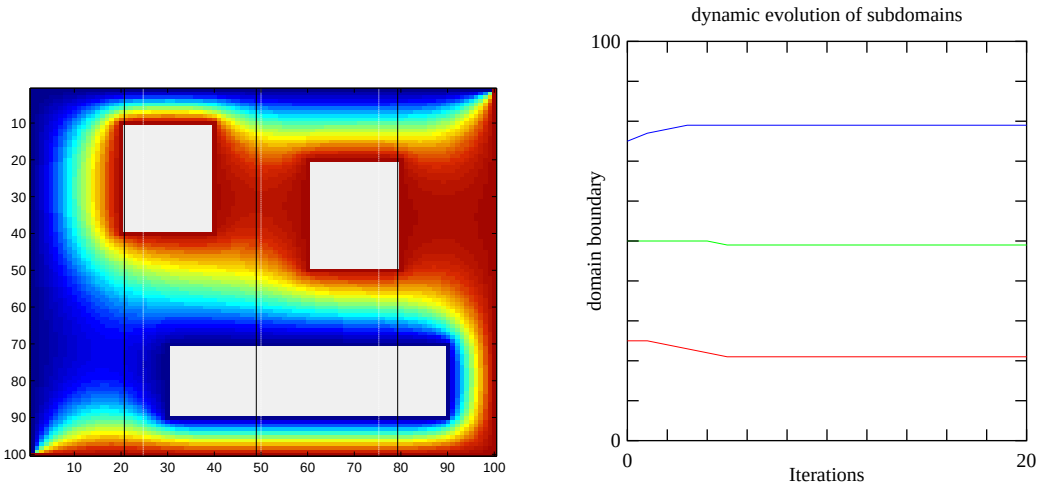


FIGURE 6.11 – Equilibrage dynamique de la taille des sous-domaines dans le cas de la résolution de l'équation de la chaleur. Les sous-domaines s'agrandissent ou se rétrécissent après chaque communication, en gardant ou non la colonne de sites communiquée par ses voisins (Simulations : Christophe Charpilloz).

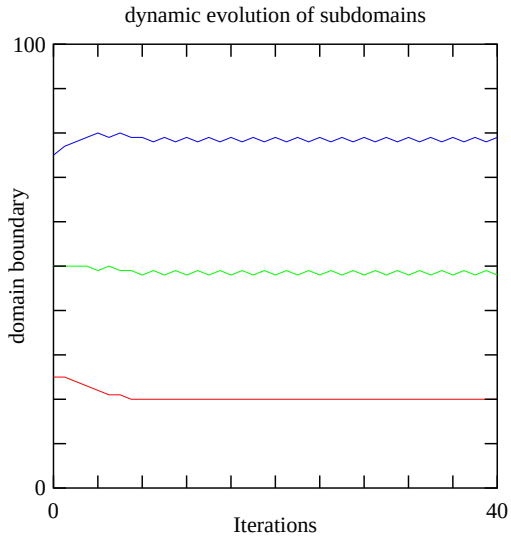


FIGURE 6.12 – Situation identique au cas de la figure précédente, mais avec une exigence trop forte sur l'égalité de la taille des sous-domaines (Simulations : Christophe Charpilloz).

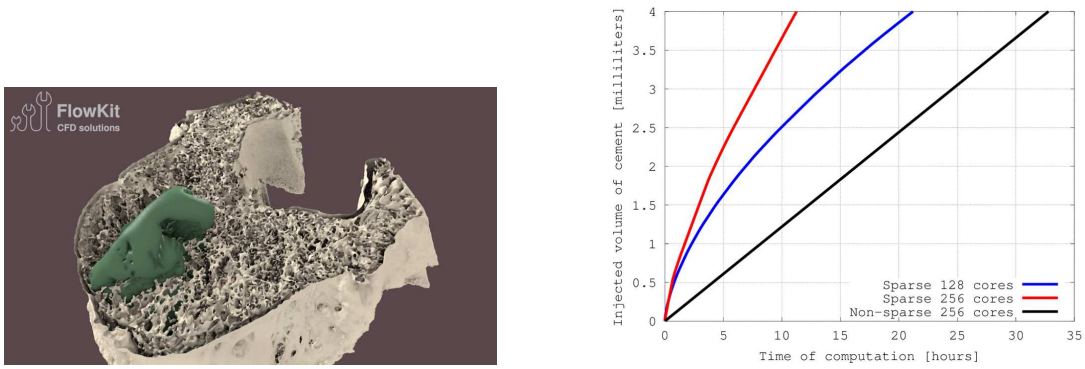


FIGURE 6.13 – **Gauche :** Simulation de l'injection de ciment dans une vertèbre fracturée. **Droite :** performance avec (noté *sparse* sur la figure) et sans (noté *non-sparse* sur la figure) équilibrage de charge dynamique. (Simulations et image : Jonas Latt, logiciel Palabos).

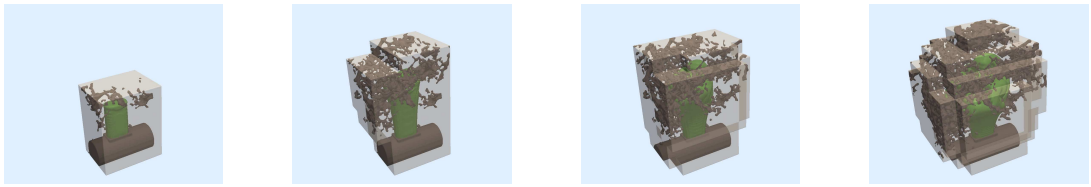

FIGURE 6.14 – A mesure que la simulation avance, on élargit la région sur laquelle les calculs ont lieu (Simulations : Jonas Latt, Palabos).

ciment dans une vertèbre fracturée, comme sugéré à la Fig. 6.13 (gauche).

La question à laquelle les médecins souhaitent répondre est la quantité exacte de ciment qu'il faut injecter dans la vertèbre. S'il n'y an a pas assez, la réparation ne se fera pas, s'il y en a trop, le ciment pourrait déborder de la vertèbre et causer des dommages aux vertèbres voisines. La simulation numérique permet donc d'évaluer au mieux la quantité idéale de ciment. Il faut une image de la vertèbre endommagée et connaître le point d'injection. Au début de la simulation, seule la région proche du point d'injection change. Les processeurs qui sont chargés de calculer les zones éloignées de ce point d'injection, n'ont rien à faire avant que le front de ciment ne les atteignent. Pour cette raison, un équilibrage de charge dynamique est souhaitable. Ici on repartitionne le domaine à mesure que le calcul avance, comme l'illustre la Fig. 6.14. A chaque étape d'équilibrage de charge, les processeurs se partagent un domaine qui grandit car il suit le front d'injection. Sur la Fig. 6.13 (droite) on peut voir le gain de temps de calcul qui découle de cet équilibrage de charge. En comparant les courbes rouge et noir, on voit que l'équilibrage de charge dynamique a permis un gain de temps d'un facteur 3 environ.

Dans le paragraphe suivant, nous discutons plus en détail le cas d'une ap-

plication **non-itérative** dans laquelle chaque proceseur qui a terminé le travail qui lui était initialement alloué essaye d'obtenir un supplément de la part des processeurs encore occupés.

### 6.9.3 Critère de rééquilibrage

Comme discuté précédemment, une partie importante du rééquilibrage de charge dynamique est de décider si un re-partitionnement est nécessaire à un moment de l'exécution. On considère encore ici une application itérative et on va dériver un critère, basé sur le temps CPU des itérations précédente, qui indiquera qu'il est opportun de rééquilibrer les charges. Ce critère est obtenu par un calcul de minimisation du temps $T_{par}$ qui suppose un **principe de persistence**, à savoir que la cause de déséquilibre de charge va continuer suffisamment longtemps tout au long de l'exécution.

Soit $m(t)$ le temps CPU de l'itération $t$ de notre application itérative. Ce temps est, comme on l'a souvent répété, le temps CPU du processeur le plus lent. Si on exécute $N$ iterations, on a que le temps total sera

$$T_{par} = \int_0^N m(t)\ dt$$

où, pour des simplifications d'algèbre, on a remplacer la somme des $m(t)$ par une intégrale. On peut aussi écrire cette équation en y ajoutant et retranchant les temps moyen $\mu(t)$ de chaque itération

$$T_{par} = \int_0^N m(t)\ dt = \int_0^N [m(t) - \mu(t)]ds + \int_0^N \mu(t)dt \tag{6.4}$$

On va maintenant supposer qu'aux itérations $t_i = t_{i-1}+\tau_{i-1}$, $i = 1, \ldots, n$, on effectue un réquilibrage de charge dont le coût CPU vaut $C_i$ pour le re-partitionnement des données. Selon l'équation (6.1), on introduit

$$u_i(t) = m_i(t) - \mu_i(t)$$

le déséquilbre de charge observé durant les itérations $t \in [t_{i-1}, t_i]$. La figure 6.15 illustre un scénario d'équilibrage aux itérations $t_i$. Après chaque équilibrage, on s'attend à ce que $u_i$ soit nul ou presque, selon que la nouvelle répartition de charge soit paraite ou non. On écrit alors

$$T_{par} = \int_0^N \mu(t)\ dt + \sum_{i=1}^{n} \left( \int_0^{\tau_i} u_i(t)dt + C_i \right) \tag{6.5}$$

On notera que la fonction $u_i$ dépend à priori du moment auquel l'équilibrage se fait. Cela rend très difficile la découverte des $t_i$ qui minimisent $T_{par}$. Il y a un cas où l'équation (6.5) peut être minimisée. C'est en supposant que tous les intervalles

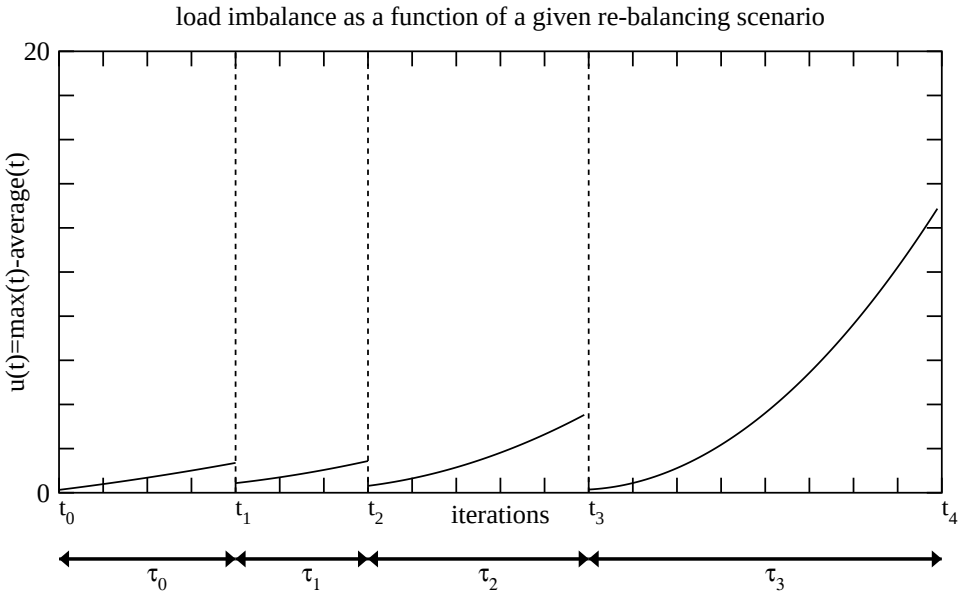


FIGURE 6.15 – Illustration des fonctions $u_i$ qui décrivent l'augmentation du déséquilibre de charge suite aux re-partitionnements effectués aux itération $t_i$.

entre rééqquilibrage sont identiques et correspondent à $\tau_i = \tau$ itérations, et que de plus la fonction $u_i = u$ est toujours la même (principe de persistence) et que $C_i = C$. On obtient donc que

$$T_{par} = \int_0^N \mu(t)dt + \frac{N}{\tau}\left(\int_0^\tau u(t)\ dt + C\right) \tag{6.6}$$

La valeur optimale de $\tau$ s'obtient en posant

$$\frac{\partial T_{par}}{\partial \tau} = 0$$

On a que

$$\frac{\partial T_{cpu}}{\partial \tau} = -\frac{N}{\tau^2}\left(\int_0^\tau u(t)dt + C\right) + \frac{N}{\tau}u(\tau) = 0 \tag{6.7}$$

dont la solution est

$$\tau u(\tau) - \int_0^\tau u(t)dt = C \tag{6.8}$$

Cette relation est implicite et tant que $u$ n'est pas connu, on ne peut pas déterminer $\tau$ a priori. Cependant, en pratique il faut considérer l'équation (6.8) comme des termes à calculer à la volée, à chaque itération, en utilisant la valeur de $u$ mesurée. Comme l'indique la figure 6.16, l'itération $\tau$ pour laquelle la surface du

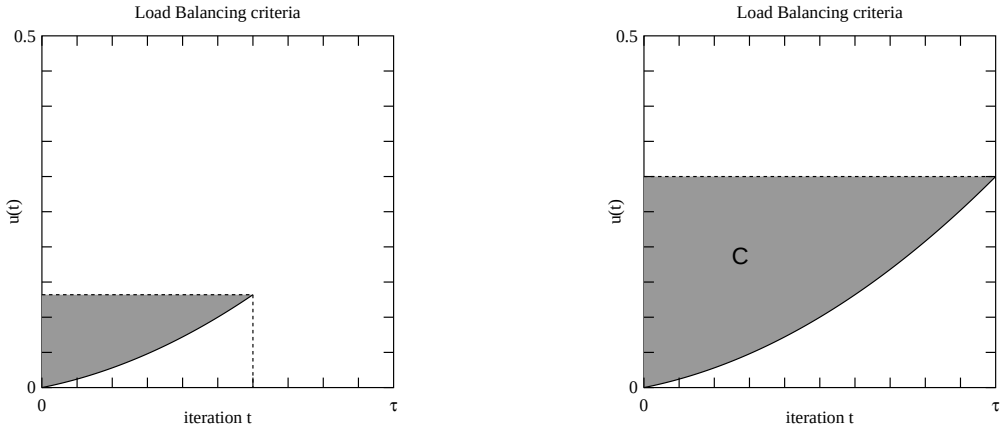


FIGURE 6.16 – Illustation du critère de rééquilibrage de charge donné par l'équation (6.8). A gauche, la surface hachurée n'atteint pas encore la valeur critique $C$. Il ne faut donc pas rééquilibrer à ce stade. A droite, on atteint une itération $\tau$ pour laquelle le critère est vérifié. Il faut don rééquilibrer les charges.

rectangle $\tau \times u(\tau)$ soustraite de la surface sous la courbe $u(t)$ atteint la valeur $C$, il vaut la peine de rééquilibrer les charges.

On remarque aussi que, même si l'équation (6.8) a été dérivée sous des hypothèses précises, elle s'applique de façon identique dans tous les cas. Cependant, dans le cas général, il n'y aura pas de garantie que la valeur de $\tau$ obtenue sera obtimale.

### 6.9.4 Cas non-itératif

Dans le cas d'applications non-itératives, chaque sous-domaine n'est calculé qu'une seule fois et la stratégie est différente que dans le cas itératif où l'on peut ajuster progressivement la taille de ces derniers entre les itération consécutives.

Pour commencer, nous allons envisager une situation symétrique (ou presque) telle que tous les processeurs jouent un rôle identique. Chacun reçoit initialement un ensemble de tâches (task pool), selon un critère quelconque et en commence l'exécution.

Lorsqu'un processeur a terminé le travail qui lui était initialement attribué et qu'il devient inactif, on obtient un déséquilibre de charge parmi les processeurs. Cela conduit à une dégradation des performances. Une répartition dynamique du travail (*dynamic load balancing*) doit être envisagée : un processeur inactif envoie une requête à un autre processeur du système pour obtenir du travail. Si ce dernier a suffisamment de travail, il en distribura la moitié au demandeur. S'il ne dispose pas assez de travail, il retournera une réponse négative car l'overhead