État de l'art des langages parallèles

Paul-Jean CAGNARD*
Laboratoire d'informatique théorique
Département d'informatique
École polytechnique fédérale de Lausanne
CH–1015 Lausanne
cagnard@di.epfl.ch

Résumé

Dans ce rapport on présente divers langages parallèles en comparant leurs spécificités et leurs points communs. On présente aussi un ensemble de modèles de calcul associés aux langages décrits dans ce rapport.

Le nombre de langages parallèles développés étant considérable, on a volontairement restreint l'ensemble des langages, ainsi que des modèles de calcul retenus, dans le but de conserver des langages représentatifs des diverses familles existantes. Enfin, les langages et modèles de calcul présentés sont mis en relation avec le langage ParCeL-2 et son modèle de calcul, que l'on présentera également.

1 Introduction

L'histoire de l'informatique parallèle est intimement liée à celle du calcul scientifique à grande échelle et à celle de la simulation numérique de phénomènes physiques. L'objectif recherché est double : on souhaite effectuer des calculs toujours plus rapidement et résoudre des problèmes de plus en plus complexes. L'idée de faire coopérer de nombreux processeurs pour augmenter la puissance de traitement disponible est séduisante, mais en même temps elle pose un grand nombre de problèmes qui lui sont propres et qui n'existent pas lorsque l'on n'utilise qu'un seul processeur. Des problèmes algorithmiques nouveaux se posent car il faut trouver un moyen de répartir les calculs sur les différents processeurs à disposition. Des problèmes de programmation surgissent, comme on a besoin de langages de haut niveau pour la programmation séquentielle, on a besoin de nouveaux langages pour permettre d'exprimer les algorithmes parallèles avec un niveau d'abstraction satisfaisant.

De nombreux langages parallèles ont vu le jour. Parmi eux on trouve beaucoup d'extensions d'anciens langages, comme les diverses extensions parallèles de FORTRAN. Certains essaient d'être généraux, d'autres sont plutôt adaptés à certains types de problèmes particuliers. ParCeL-2 fait partie de cette dernière catégorie. Le domaine d'application visé est celui du « calcul cellulaire », concept générique qui regroupe les modèles de calcul réguliers à espace et temps discrets. Ce sont des modèles abondamment utilisés en simulation numérique et qui ont des caractéristiques communes avec les automates cellulaires. On trouve notamment dans ces modèles les automates cellulaires proprement dits, les méthodes de différences finies,

*Projet numéro 21-49519.96 financé par le Fonds National suisse de la Recherche Scientifique

les méthodes d'éléments finis et les méthodes de gaz en réseaux ou « lattice gas methods » [1] (FHP, lattice Boltzmann [2]). La propriété commune de ces modèles est de discrétiser le problème à résoudre ou le phénomène à simuler et surtout de discrétiser le temps. Le calcul procède ainsi par étapes successives qui représentent le passage du système observé dans une suite d'états aux instants $t + i\tau, i = 0, 1, ..., n$, où t est le temps initial et τ le pas de temps discret. Dans le cas des méthodes de gaz en réseaux, l'espace est divisé en une grille de petites cellules dans lesquelles peuvent se trouver des particules de gaz qui peuvent entrer en collision au cours du temps.

D'un point de vue informatique, la programmation de ce genre de modèles a été très souvent effectuée à l'aide de méthodes, de langages et de structures de données hérités de la programmation séquentielle. Dans le cas de l'exemple ci-dessus l'espace est alors représenté par un tableau à deux dimensions qui est parcouru à chaque pas de temps et mis à jour en fonction de son état au pas précédent. Le but de ParCeL-2 est d'offrir au programmeur le moyen de décrire le système à simuler d'une manière beaucoup plus intuitive et directe. Au lieu de devoir d'abord représenter le système à l'aide d'un tableau et d'en assurer la gestion au moyen des routines adéquates, et ensuite de devoir s'occuper de la parallélisation de son programme, consistant à répartir des segments de données sur les différents processeurs et à gérer les communications entre eux, le programmeur n'aura qu'à créer une cellule type en décrivant son comportement et son voisinage, et à donner la taille de l'espace, c'est-à-dire le nombre de cellules. Le compilateur et l'environnement d'exécution se chargent ensuite de créer et de répartir le nombre de processus nécessaire sur les processeurs effectivement disponibles, et d'assurer la gestion de l'évolution des différentes cellules et des communications entre elles.

Le langage ParCeL-1 décrit dans [3] constitue le point de départ du projet ParCeL-2. Il permet la description de cellules types et de leurs interconnexions, mais dans l'optique du connexionisme, ce qui entraîne un certain dynamisme dans le nombre de cellules à chaque pas et dans leurs interconnexions. En effet, des cellules peuvent être créées ou détruites, et les connexions peuvent être redirigées vers d'autres cellules, à chaque étape. ParCeL-2 cherche plutôt à étudier la mise en œuvre de grands réseaux de cellules réguliers et statiques.

Dans la suite de cet article on présentera différents modèles de programmation parallèles et notamment celui qui correspond à ParCeL-2. Ensuite on passera en revue divers langages parallèles et modèles de calcul intrinsèquement parallèles. Enfin on présentera en détail l'environnement d'exécution de ParCeL-2.

2 Modèles de calcul parallèle

L'une des principales raisons souvent évoquée pour le manque de succès du parallélisme est l'existence de trop nombreux modèles de programmation ainsi que de trop nombreuses architectures de machines différentes, ce qui oblige les concepteurs de programmes à repartir de zéro à chaque portage de code sur une nouvelle machine. En programmation séquentielle, ce problème ne se pose pas car on dispose d'un modèle de programmation sur lequel tout le monde s'accorde, le modèle de von Neumann. Ce modèle reste suffisamment abstrait pour correspondre à de nombreuses architectures de machines différentes. Notamment, il ne décrit ni la gestion des périphériques, ni la mémoire virtuelle, ni la gestion du chargement de code segmenté, ni aucun des aspects qui sont très liés à une architecture et à un système particuliers, et qui feraient perdre sa généralité au modèle. Les langages de programmation séquentiels offrent tous une vision abstraite de la machine sur laquelle seront exécutés les

programmes. Cela offre l'avantage considérable au programmeur de pouvoir s'affranchir de la diversité des architectures des ordinateurs séquentiels.

De nombreux modèles pour la programmation parallèle ont été proposés. Tous ces modèles font très nettement la distinction entre des aspects antithétiques du parallélisme comme par exemple le calcul vectoriel ou non, la mémoire distribuée ou partagée et la communication entre processus par envoi de messages ou par mémoire partagée. Le principal obstacle à l'adoption d'un modèle de programmation parallèle unificateur comme celui de VON NEUMANN est qu'il doit satisfaire aux trois exigences suivantes [4]:

- refléter de manière fidèle les contraintes des machines existantes,
- avoir un large domaine d'application en ce qui concerne les machines existantes et à venir,
- permettre des prédictions de performance précises.

Pour l'instant, aucun modèle parallèle n'a atteint ce but.

Le domaine d'application privilégié de ParCeL-2 étant la programmation de calculs dans des réseaux de processus réguliers selon des modèles apparentés aux automates cellulaires, le modèle BSP, que l'on présentera en détail ci-dessous, apparaît particulièrement intéressant. Il satisfait pleinement aux deux dernières exigences ci-dessus mais pas entièrement à la première, notamment en raison de la difficulté de modéliser les communications dans les machines parallèles.

2.1 Modèle BSP

Le modèle BSP, *Bulk Synchronous Parallel*, est l'un des plus utilisés actuellement avec le modèle PRAM. On va le présenter ci-après de manière plus détaillée que les autres modèles pour deux raisons principales. Premièrement, cela va permettre de montrer que le modèle BSP est un bon candidat pour le rôle de l'équivalent dans le domaine calcul parallèle du modèle de VON NEUMANN dans le domaine du calcul séquentiel. Il répond particulièrement bien aux trois exigences formulées ci-dessus. Deuxièmement, le modèle d'exécution des calculs de ParCeL-2 correspond assez précisément au modèle d'exécution de BSP, comme on le verra plus loin.

2.1.1 Modèle architectural

Le modèle de calcul BSP [5] est présenté par ses concepteurs comme une tentative de fournir un modèle unificateur pour l'informatique parallèle comme l'est celui de VON NEUMANN pour le calcul séquentiel.

Le modèle BSP offre au programmeur une vision abstraite très simple de la machine sous-jacente. Cette machine abstraite, ou machine BSP, se compose de trois éléments (voir figure 1) :

- un ensemble de paires processeur-mémoire,
- un réseau de communication entre les paires, processeur-mémoire qui permet de transmettre des messages point à point,
- un mécanisme efficace permettant la synchronisation de tous les processeurs ou seulement d'une partie de ceux-ci.

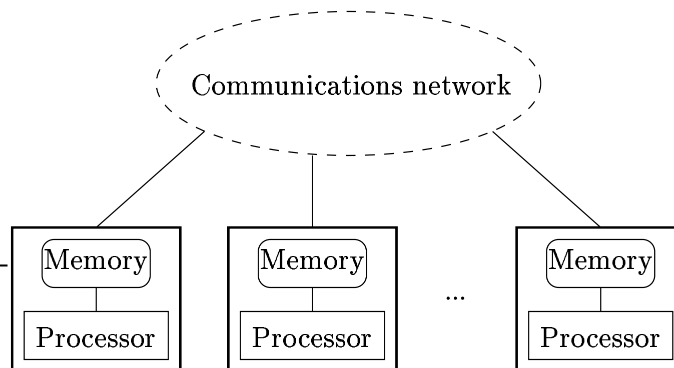

FIG. 1 – *Modèle architectural d'une machine BSP*

Il convient de noter que cette description n'inclut pas de fonctions de combinaison de résultats, de réplication ou de *broadcast*.

Parmi les systèmes parallèles existants qui correspondent à cette description et dont le modèle BSP, par conséquent, constitue une abstraction, on peut citer :

- les machines monoprocesseurs avec leur mémoire cache et principale,
- les réseaux de stations de travail utilisant PVM, MPI ou d'autres bibliothèques de passage de messages,
- les machines multiprocesseurs à mémoire distribuée,
- dans une certaine mesure, les machines multiprocesseurs à mémoire partagée.

Afin de pouvoir mesurer les performances d'une machine BSP, il est utile de définir plusieurs paramètres. Le nombre de processeurs est évidemment important, ainsi que la rapidité de ceux-ci. On peut alors définir un *step* comme l'unité de base de calcul, par exemple une opération en virgule flottante, et la vitesse d'une machine se mesure en *steps*/seconde.

La capacité et la rapidité du réseau de communication sont également deux caractéristiques importantes pour la performance d'une machine. Afin de faciliter la comparaison entre différents systèmes, on mesurera la performance du réseau de communication en unités de vitesse de calcul. Le coût d'une synchronisation entre les processeurs peut être mesuré en fonction du nombre de *steps* qui auraient pu être exécutés pendant le temps nécessaire à la synchronisation. Cela permet de distinguer les systèmes qui fournissent une synchronisation rapide, sur lesquels peu d'opérations auraient pu être exécutées pendant une synchronisation, de ceux qui offrent une synchronisation très lente par rapport à leur puissance de calcul. En général, on peut espérer de meilleures performances de la part d'un système qui possède un faible valeur de ce paramètre.

De la même manière, pour estimer le taux de transfert du réseau entre les processeurs, on définit le coût en termes de *steps* par mot de données transmis. Cela donne le rapport entre la rapidité de calcul du système et sa rapidité de communication. Plus ce quotient est petit, plus la machine est performante, car elle offre un bon équilibre entre la vitesse de calcul et la vitesse de communication, ce qui permet d'obtenir plus facilement des performances évolutives (*scalable*).

On en arrive ainsi à la définition proprement dite des quatre paramètres du modèle BSP :

- p : le nombre de processeurs de la machine,
- s : la rapidité du processeur, exprimée en *steps*/seconde,
- l : le coût, en nombre de *steps*, d'une barrière de synchronisation,
- g : le coût, en *steps*/mot, de l'envoi de données.

On peut remarquer qu'il n'y a en fait que trois paramètres indépendants : p, l et g. De plus ces paramètres représentent des caractéristiques moyennes du systèmes et leur valeur est obtenue à partir de mesures lors de l'exécution de programmes de test, ils ne correspondent donc généralement pas aux données maximales fournies par les constructeurs.

La valeur de g permet d'estimer le temps nécessaire aux échanges de données entre processeurs. Si le nombre maximal de mots qui arrivent sur un seul processeur lors d'un échange est h, on estime que gh *steps* auraient pu être exécutés pendant l'échange.

À titre d'exemple, on donne ci-dessous quelques valeurs tirées de [6].

- Un réseau de stations de travail dont les processeurs ont une performance soutenue de 20 MFlops peut avoir des valeurs de g de l'ordre de quelques centaines de Flops par mot transféré et des valeurs de l comprises entre 10'000 et 100'000 Flops.
- Une machine comme le Cray T3E, dont les nœuds ont une performance soutenue de 45 MFlops, aura des valeurs de g comprises entre 1,5 et 2,5 Flops/mot, et des valeurs de l de quelques centaines de Flops.
- Une Silicon Graphics Power Challenge avec des nœuds d'une performance soutenue de 75 MFlops offre des valeurs de g de l'ordre de 10 Flops/mot et de l de l'ordre de 1000 Flops.

Plus la valeur de g tend vers 1 et plus la valeur de l devient petite. On dispose alors d'un système très extensible. Pour un même problème, on pourra ajouter de nombreux processeurs sans faire chuter brutalement les performances, car la latence du réseau étant faible, l'accès à des données distantes ne prend pas un temps prohibitif par rapport à l'accès à des données locales. Dans le cas contraire, on a un ensemble de processus à grain très fin qui communiquent énormément, et comme les communications prennent beaucoup plus de temps que les accès à la mémoire locale, ce qui a un effet désastreux sur les performances.

En résumé, on peut dire que le modèle BSP offre l'avantage non négligeable de mieux correspondre aux architecutures de machines réelles que ne le fait par exemple le modèle PRAM [4, 7, 8]. Il permet en outre de bonnes prédictions de performances au moyen de 4 paramètres qu'il est aisé d'obtenir pour une machine particulière.

2.1.2 Modèle de calcul

On a présenté ci-dessus le modèle BSP en tant que modèle architectural d'ordinateur. On va maintenant le présenter comme un modèle de calcul, ce qui intéresse plus directement ParCeL-2. On ne considère plus les processeurs disponibles sur la machine utilisée, mais des processeurs virtuels, $P_1, \ldots, P_n$, qui seront exécutés sur la machine réelle. Le nombre n de processeurs

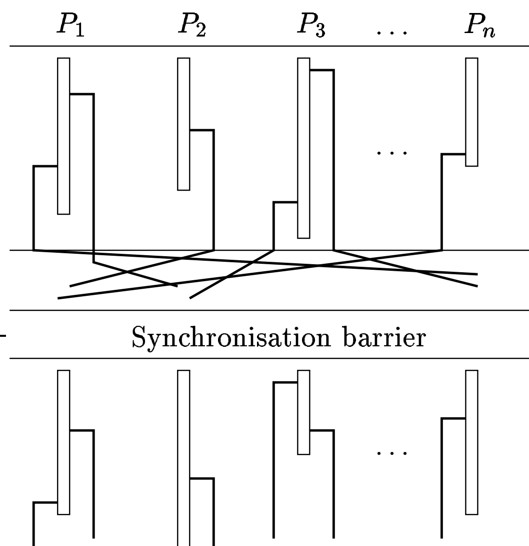

FIG. 2 – *Exécution d'un programme dans le modèle de calcul BSP*

virtuels est indépendant du nombre de processeurs réels disponibles, il peut notamment être inférieur ou largement supérieur.

Un programme BSP se compose simplement d'une suite de *supersteps*, voir figure 2. Ces *supersteps* sont exécutés sur des processeurs virtuels qui seront placés sur les processeurs réels de la machine utilisée. Un *superstep* est lui-même constitué de trois étapes successives : une étape de calcul, une étape de communication entre processeurs lors de laquelle les données requises sont transférées entre les processeurs, et une barrière de synchronisation qui attend que les communications soient terminées, qui rend les données transférées accessibles dans les mémoires locales, et qui, lorsqu'elle est terminée, autorise le début du *superstep* suivant. Lors des phases de calcul, des accès à des données distantes peuvent être faits. Les requêtes concernant ces données distantes ne sont pas bloquantes, mais les données concernées ne seront disponibles que lors de la prochaine phase de calcul. L'accès proprement dit aux données distantes a donc lieu lors des phases de communication exclusivement.

Cette approche de la programmation parallèle peut s'appliquer à tous les types de machines parallèles : les architectures à mémoire distribuée, les machines multiprocesseurs à mémoire partagée et les réseaux de stations de travail. De plus le modèle de la mémoire offert au programmeur est consistant avec celui des machines actuelles à accès non uniforme à la mémoire (NUMA). Des expériences ont montré [9] que l'exécution d'un grand nombre de barrières de synchronisation n'est pas forcément aussi pénalisant en termes de performances qu'on le croit généralement. En outre, le fait de séparer les calculs et les communications permet d'envisager de router les communications plus efficacement, car on peut regrouper les blocs de données à destination d'un processeur réel particulier.

2.2 Autres modèles

Après avoir présenté le modèle BSP ci-dessus, on va présenter quelques autre modèles de calcul parallèles en les comparant avec le modèle BSP.

2.2.1 Task-channel

Le modèle *task-channel*, introduit dans [10], cherche à permettre au programmeur d'écrire des programmes modulaires. Pour cela, deux abstractions élémentaires nécessaires aux programmes parallèles sont définies : les *tâches* et les *canaux*. Il est particulièrement intéressant dans le cas de ParCeL-2, car combiné au modèle BSP, il se rapproche du modèle de calcul de ParCeL-2 dans lequel, comme on le verra plus loin, les processus sont exécutés de manière globalement synchrone et les communications entre processus ont lieu par l'intermédiaire de canaux.

Les tâches et les canaux sont des abstractions qui correspondent au modèle architectural général d'un ordinateur parallèle, et à un ensemble de processus en exécution concurrente qui nécessitent des communications entre eux. Les caractéristiques du modèle *task-channel* sont les suivantes :

1. Un programme parallèle est composé de plusieurs tâches qui sont exécutées concurremment. Le nombre de tâches peut varier durant l'exécution de programme.

2. Une tâche encapsule un programme séquentiel et de la mémoire locale. Il s'agit en fait d'une machine de von Neumann virtuelle. Un ensemble de ports d'entrée et de sortie constitue l'interface entre une tâche et son environnement.

3. Une tâche peut accomplir quatre types d'actions en plus de lire et d'écrire dans sa mémoire locale. Elle peut envoyer des messages sur ses ports de sortie, lire des messages sur ses ports d'entrée, créer de nouvelles tâches ou se terminer.

4. Un envoi de message est asynchrone, ce qui signifie qu'il se termine immédiatement. Une réception de message est synchrone, c'est-à-dire qu'elle bloque l'exécution d'une tâche jusqu'à ce qu'un message soit disponible.

5. Une paires de ports d'entrée et de sortie peut être connectée pour former un *canal*. Les canaux peuvent être créés et détruits. Une référence à un canal peut être incluse dans les messages, ce qui permet de modifier les connexions dynamiquement.

6. Les tâches peuvent être réparties sur les processeurs physiques de différentes manières. Cependant, les différentes répartitions n'affectent pas la sémantique d'un programme donné. En particulier, plusieurs tâches peuvent être affectées à un seul processeur. Dans le cas général peut aussi envisager de répartir une seule tâche sur plusieurs processeurs, mais cette possibilité n'est pas considérée par ce modèle de calcul.

La notion de tâche permet de représenter le concept de localité. Les données peuvent être locales ou distantes pour une certaine tâche. La notion de canal permet de spécifier les dépendances entre les données de différentes tâches.

Les tâches représentent donc un fragment de code qui peut être exécuté séquentiellement sur un processeur. L'optimisation des communications entre tâches est rendue possible par la notion de canal. En effet, si deux tâches qui partagent un canal sont situées sur deux processeurs différents, la connexion est établie à l'aide de communications entre les processeurs. Si deux tâches sont situées sur le même processeur, un mécanisme plus efficace peut être employé pour la communication entre elles, tel que des variables partagées par exemple.

En outre, puisque les tâches utilisent toujours des canaux pour communiquer, quel que soit le processeur sur lequel elle sont exécutées, le résultat d'un programme parallèle ne dépend

pas de la répartition des tâches sur les différents processeurs disponibles. Cette invariance permet de concevoir des algorithmes sans se soucier du nombre de processeurs disponible lors de l'exécution.

Une tâche peut être vue comme une entité qui encapsule des données et du code ayant accès à ces données. Les ports par lesquels entrent et sortent les messages peuvent être vus comme une interface d'accès au code et aux données de cette tâche. On dispose ainsi d'un moyen de conception modulaire de programmes parallèles, similaire à un modèle orienté objet dans lequel l'héritage est absent et les méthodes sont remplacées par des canaux pour spécifier les interactions entre objets.

2.2.2 PRAM

Le modèle PRAM est un modèle classique très utilisé en algorithmique parallèle pour analyser la complexité des applications. Ce modèle est caractérisé par les paramètres suivants [11, 4]:

- un ensemble non borné de processeurs, $p_0, p_1, \ldots$
- une mémoire locale non bornée, associée à chaque processeur, dont les emplacements sont notés $ml_0, ml_1, \ldots$
- une mémoire globale non bornée que l'on suppose accessible en une unité de temps par les processeurs et dont les emplacements sont notés $M_0, M_1, \ldots$

Un programme est composé d'une séquence finie d'instructions étiquetées. L'exécution d'une instruction prend toujours une unité de temps. Diverses politiques peuvent être adoptées en ce qui concerne l'accès à la mémoire par les processeurs de manière concurrente ou exclusive en lecture ou en écriture, ce qui donne quatre types possibles de machine PRAM.

Le modèle PRAM a permis le développement d'algorithmes parallèles indépendants des architectures de machines à une époque où celles-ci changeaient souvent. Cependant, si certains algorithmes PRAM ont pu être traduits en algorithmes efficaces pour mémoires distribuées, il n'en est de loin pas toujours le cas, surtout pour les algorithmes dont les communications dépendent fortement des données. Notamment, contrairement au modèle BSP, le modl̀e PRAM ne permet pas de prendre en compte la latence et la bande passante du réseau de communication entre processeurs. De plus, il apparaît comme extrêmement difficile de construire des machines parallèles conformes au modèle PRAM.

2.2.3 LogP

Le modèle LogP tire son nom des quatre paramètres qui le caractérisent: L, o, g et P. Il s'agit d'un modèle asynchrone. On en trouve une description dans [12]. Ce modèle apporte la notion d'acceptation d'un message par le réseau de communication. Un processeur peut se trouver dans l'un des trois états suivants: exécuter un calcul local, recevoir un message, ou soumettre un message au réseau de communication à destination d'un autre processeur. L'envoi ou la réception d'un messages prennent o unités de temps, où o est appelé l'*overhead.* Pendant ce temps un processeur ne peut pas faire autre chose. P représente le nombre de processeurs. Les deux autres paramètres, L et g, caractérisent le comportement du réseau de communication. L est une borne supérieure pour la latence du réseau lors du transfert

d'un message contenant un seul mot ou un petit nombre de mots. L'intervalle de temps minimum entre deux transmissions ou réceptions consécutives est appelé le *gap* et correspond au paramètre g. L'inverse de g représente la bande passante disponible entre deux processeurs.

Une étude comparée de LogP et de BSP a été faite dans [13]. Il en ressort que l'on peut simuler LogP sur BSP , mais avec un ralentissement certain, et que le modèle BSP apparaît comme plus général et plus simple à utiliser que le modèle LogP.

2.2.4 LogGP

Ce modèle introduit dans [14] se présente comme une extension du modèle LogP. Alors que ce dernier fournit les caractéristiques du réseau de communication pour des messages courts de taille fixe, LogGP ajoute un paramètre G qui est défini comme le *gap per byte* ou le temps pris par chaque octet d'un long message. L'inverse de G caractérise la bande passante disponible pour de longs messages. Le quotient g/G est une très bonne mesure du gain obtenu en regroupant les données dans de longs messages. Le principal intérêt du modèle LogGP est qu'il permet de modéliser l'avantage qu'il y a à transférer de gros messages plutôt que de petits messages, ce qui va dans le même sens que BSP qui exécute toutes les communications en même temps, à la fin des *supersteps*, de manière à pouvoir regrouper les différentes données en messages de taille plus importante.

2.2.5 Block distributed memory

On peut trouver une présentation de ce modèle, que l'on abrégera en BDM, dans [15]. En plus de tenir compte de la latence lors des accès à la mémoire et de la bande passante du réseau de communication, ce modèle permet de prendre en compte la localisation spatiale des données. Il autorise en outre le placement initial des données et la recherche des données en mémoire à l'avance (*pipelined prefetching*). Quatre paramètres définissent ce modèle : p, τ, σ et m. Le paramètre σ peut être omis sans perte de généralité. Le paramètre p désigne le nombre de processeurs, chacun d'eux étant vu comme une machine à accès aléatoire (RAM) dans laquelle l'accès à la mémoire prend une unité de temps. Chaque processeur est connecté á un réseau de communication. Les données sont transférées à l'aide de messages point à point, chaque message étant un paquet contenant m mots d'emplacements *consécutifs* dans la mémoire locale. Le modèle de programmation étant celui d'une mémoire partagée, l'accès à des données distantes implique la préparation d'un paquet contenant la requête, l'injection de ce paquet dans le réseau, la réception de ce paquet par le processeur visé, et finalement l'envoi d'un paquet contenant m emplacements consécutifs comprenant la valeur demandée au processeur ayant soumis la requête. Le paramètre τ est le temps d'attente maximum nécessaire, ou latence, pour qu'un processeur reçoive le paquet contenant les données distantes demandées, et σ est la vitesse à laquelle un processeur peut lire ou écrire un mot sur le réseau. Ainsi l'accès à un emplacement de mémoire distant coûte $\tau + m\sigma$.

Le modèle de communication de BDM correspond à celui de LogP. La principale nouveauté réside dans la prise en compte de la proximité spatiale des données en transmettant des paquets de taille m comprenant les données souhaitées.

2.2.6 Modèle d'acteurs

Le modèle d'acteurs a d'abord été proposé de manière complète et rigoureuse par Gul Agha [16, 17, 18], suite à des travaux de Carl Hewitt dans le domaine de l'intelligence artificielle. Ce

modèle est motivé par la volonté de pouvoir créer des *sociétés d'experts*. Un acteur est défini comme une entité ayant une adresse à laquelle on peut envoyer des messages, et contenant une action à accomplir en correspondance avec les messages reçus. Les données locales d'un acteur, des adresses d'autres acteurs, sont appelées ses *accointances*. Les instructions qui régissent le comportement d'un acteur sont appelées son script. Un acteur est activé dès qu'il reçoit un message et est suspendu le reste du temps. L'adresse d'un acteur peut être envoyée comme message à un autre acteur. Les actions accomplies par un acteur lors de la réception d'un message sont essentiellement de trois types : envoyer des messages, créer de nouveaux acteurs et définir son comportement pour le message suivant. Les acteurs se transforment ainsi de message en message en changeant d'*accointances* et même de script. Le parallélisme provient de l'envoi de messages asynchrones, non bloquants à l'émission. Un acteur peut envoyer plusieurs messages à la suite et ainsi activer plusieurs autres acteurs en parallèle. En outre, un acteur peut traiter plusieurs messages en parallèle. Dès qu'il reçoit un message et que son nouveau comportement est spécifié, il peut commencer un traitement. Pour que le traitement en parallèle puisse avoir lieu, il faut qu'un acteur soit capable de préciser son nouveau comportement avant la fin du traitement en cours. Les acteurs qui ne peuvent le faire sont appelés acteurs *séquentialisés*.

Un mécanisme important a été introduit par Gul Agha, celui de délégation. Un acteur qui reçoit un message peut déléguer le travail à accomplir à un autre acteur, appelé son délégué. Ce dernier effectue la tâche requise et répond directement à l'acteur ayant envoyé le message original. Le délégué d'un acteur est une accointance et peut donc changer au cours de l'évolution de cet acteur.

Finalement, on peut remarquer que dans ce modèle, tout est ramené au concept primitif d'acteur. Par exemple, une addition est réalisée par l'envoi d'un message contenant les deux nombres à additionner à un acteur « + ». De plus, l'envoi de messages utilise des acteurs messagers qui vont effectivement transmettre les messages. Seuls certains acteurs de bas niveau ne sont pas constitués d'acteurs, ce sont par exemple les nombres et les messagers. La volonté de tout transformer en acteurs rend le modèle très pur mais aussi très lourd à implanter.

3 Modèle de calcul de ParCeL-2

Dans cette section on va décrire le modèle de calcul de ParCeL-2 plus en détail. On pourra constater qu'il emprunte des concepts à plusieurs modèles présentés ci-dessus.

3.1 Modèle de calcul

Les principaux concepts du modèle de calcul de ParCeL-2 se retrouvent soit dans le modèle BSP soit dans le modèle task-channel. Du modèle BSP on reprend le fonctionnement cyclique des programmes, et du modèle Task Channel on reprend l'idée de l'utilisation de canaux pour les communications entre processus. Le langage ParCeL-2 étant destiné à la description de programmes massivement parallèles réguliers dont les processus ont un grain fin, on fera l'hypothèse que le nombre de processus est beaucoup plus grand que le nombre de processeurs disponibles dans la machine sur laquelle le programme est exécuté.

Le type d'applications visé par ParCeL-2 conduit à choisir un modèle de calcul statique. C'est-à-dire que le nombre de processus présents dans un programme est constant et connu au moment de la compilation. Il n'est donc pas possible de créer de nouveaux processus lors de l'exécution d'un programme. En revanche, il est possible d'optimiser la gestion des processus

par rapport au cas dynamique. La topologie de communication est également statique et définie à la compilation. Cela signifie que les communications auront toujours lieu entre les même processus au cours d'un programme. En particulier, il n'est pas possible de créer de nouveaux liens de communications entre processus ni de modifier les liens existants en cours d'exécution.

3.1.1 Calculs

Un programme ParCeL-2 est composé de plusieurs processus exécutés de manière concurrente. L'exécution d'un processus consiste en une alternance répétée jusqu'à la fin du programme entre phases de calcul et de communication, comme c'est le cas dans le modèle BSP. Une barrière de synchronisation est placée entre la phase de calcul et la phase de communication, cf. figure 2. Il n'est pas obligatoire que tous les processus attendent à la barrière de synchronisation à chaque cycle de l'exécution ; une barrière de synchronisation peut avoir lieu parmi des sous-ensembles de processus, ce qui permet d'éviter l'attente du passage de la barrière de synchronisation par des processus qui seraient plus lents.

Chaque processus dispose, afin d'effectuer ses calculs, d'une mémoire locale à laquelle il est le seul à pouvoir accéder. Il n'existe pas de mémoire partagée par les différents processus. Le seul moyen d'échange d'information est l'utilisation des canaux de communication, comme on va le voir ci-dessous.

3.1.2 Communications

Les différents processus créés doivent pouvoir communiquer entre eux si l'on veut que les programmes effectuent des calculs parallèles intéressants, sinon on se retrouverait en présence de plusieurs programmes séquentiels totalement indépendants. Dans le langage ParCeL-2, le seul moyen de communication offert est le canal de communication. Un canal de communication peut être vu comme un lien entre deux processus le long duquel des données seront envoyées. Un canal est caractérisé par plusieurs propriétés : un type, qui correspond au type des données transmises dans ce canal; une période, dont l'inverse indique la fréquence maximale à laquelle de nouvelles données peuvent être transmises par rapport à la période de synchronisation du processus dans laquelle le canal considéré débouche; une sémantique de transmission des données que l'on décrira ci-après.

Il existe trois sortes de canaux de communication : les canaux de type point à point, ou 1-1, qui relient un processus à un autre, les canaux de type 1-n, qui relient le point d'entrée du canal à n points de sortie dans différents processus, et enfin les canaux de types n-1 qui relient n points d'entrée provenant de plusieurs processus à un seul point de sortie. Avec les deux premières sortes de canaux, la sémantique des communications est la suivante : une seule valeur peut-être transmise dans le canal par cycle du processus émetteur; si ce dernier écrit plusieurs valeurs dans le canal au cours d'un cycle, seule la dernière sera effectivement envoyée. Une valeur envoyée dans un canal reste disponible au point de sortie de ce canal tant qu'une autre valeur n'a pas été transmise. On remarquera que l'on peut considérer les canaux 1-n comme n canaux 1-1 dans lesquels les mêmes valeurs sont transmises aux mêmes instants. Les canaux de type n-1 posent le problème de l'accès concurrent à une zone de mémoire partagée. En effet, différents processus vont envoyer des valeurs différentes dans leur point d'entrée du canal qui débouche sur un seul point de sortie. Il existe pour cette raison deux manières différentes de communiquer avec ces canaux. Selon la première, toutes les valeurs transmises

sont disponibles au point de sortie du canal dans une structure de donnée correspondant à un ensemble de valeurs du type de données du canal, sans ordre particulier. Le point de sortie se comporte alors comme une boîte aux lettres. Selon la deuxième manière, il est possible d'associer un opérateur au canal qui combinera les différents valeurs transmises en une seule, du même type que celui du canal.

Il est à noter que l'on adopte un modèle de canal unidirectionnel. Si l'on souhaite une communication bidirectionnelle, il sera nécessaire de créer deux canaux, un dans chaque sens.

4 Langages parallèles

Dans ce qui suit on va examiner différents langages parallèles correspondant à divers modèles de calcul afin de situer ParCeL-2.

Les langages parallèles peuvent être regroupés en quatre familles générales :

1. Les extensions de langages séquentiels existants. Le langage parallèle consiste alors en l'introduction de quelques mots-clefs supplémentaires qui permettent d'exprimer le parallélisme et les autres concepts introduits dans l'extension du langage original. Le langage Jade [19] ainsi que, dans une certaine mesure ParCeL-2, font partie de cette classe.

2. Les langages utilisant des bibliothèques. Dans ce cas, le programmeur écrit toujours un programme dans un langage séquentiel, et ce sont les routines de bibliothèques externes qui s'occupent de paralléliser l'exécution. L'expression du parallélisme est ainsi masquée au programmeur. On trouve dans cette catégorie des langages comme Eiffel dans le cas du projet EPEE [20].

3. Les langages à parallélisme implicite. Il s'agit de langages séquentiels existants dans lesquels on laisse le compilateur s'occuper entièrement de la parallélisation d'un programme. Le programmeur n'a pas à se soucier de la parallélisation des instructions de son programme, ni de la répartition des données. On trouve dans cette famille des langages dérivés de langages fonctionnels, comme LISP ou ML, ou de langages déclaratifs comme PROLOG. Dans le cas des langages fonctionnels on cherche à exécuter en parallèle l'évaluation des arguments des fonctions, ce qui ne donne qu'un nombre de processus parallèles relativement restreint, de l'ordre de 2 ou 3, car il est rare d'utiliser des fonctions ayant un grand nombre d'arguments. Dans le cas de langages déclaratifs, on cherche à exécuter en parallèle la résolution de sous-buts, ou sous-clauses. Dans [21] on trouve des exemples de cette famille; D-LISP est un dérivé de LISP, Concurrent Clean est un dérivé de Clean, Haskell est un langage fonctionnel dont il existe une version parallèle, et enfin, Strand est un dérivé de PROLOG.

4. Les langages entièrement nouveaux. Ils peuvent parfois s'inspirer de langages existants, mais souvent ils introduisent leur propre syntaxe, leurs propres mots-clefs, ainsi qu'une sémantique unifiée, inédite. Ce sont souvent des langages créés en même temps qu'un modèle de calcul parallèle donné. C'est le cas de ParCeL-1, des langages d'acteurs, de Cellang et de CuPit-2 par exemple.

4.1 BSP-L

BSP-L est un langage expérimental utilisant le modèle de calcul BSP. On en trouve une description dans [5]. Son but est de fournir un environnement de test pour des constructions

syntaxiques afin de pouvoir étendre des versions parallèles de FORTRAN, de C ou d'autres langages parallèles de haut niveau. Il est utilisé principalement par l'environnement H-BSP dont le but est de fournir environnement de programmation parallèle pour des logiciels portables fonctionnant selon le modèle BSP. Il a pour but de rester très orienté vers les applications matricielles.

4.2 GPL ou GL

GPL a d'abord été développé sous le nom de GL [22]. Il a ensuite été développé dans le cadre d'un projet européen ESPRIT GEPPCOM sous le nom de GPL. Il s'agit d'un langage ayant pour modèle de calcul le modèle BSP. Il est possible d'utiliser dans un programme GPL des fragments de code C existants. L'exécution parallèle de parties de programmes est indiquée par la présence de blocs spécifiques dans un programme séquentiel. Il n'est pas possible de créer plus de processus parallèles qu'il n'y a de processeurs à disposition.

Le développement de ce langage en est resté au stade de l'ébauche, les personnes à l'origine de ce projet ont préféré se concentrer sur le développement d'une bibliothèque de routines pour C et FORTRAN permettant d'écrire des programmes selon le modèle BSP, ainsi que de bibliothèques de routines de plus haut niveau, permettant par exemple de manipuler des tableaux, au moyen de leur bibliothèque BSP.

4.3 Opal

Le langage Opal [23] utilise le modèle BSP comme modèle de calcul. La syntaxe utilisée est celle d'Ada avec quelques extensions pour permettre l'expression de *supersteps*. Le nombre de processus parallèles est statique. Opal offre la notion de processus et celle de groupe. Les groupes permettent de rassembler les processus en une entité. Les processus sont exécutés de manière synchrone à l'intérieur d'un groupe, et les groupes sont exécutés de manière asynchrone entre eux. Les divers processus sont exécutés de manière cyclique selon le modèle BSP, et ils communiquent entre eux au moyen de variables partagées. Toute modification d'une variable partagée n'est effective qu'à la fin d'un *superstep*, conformément au modèle BSP. Les groupes de processus ne partagent pas de mémoire et communiquent par un mécanisme de rendez-vous analogue au mode de communication entre tâches d'Ada. Les rendez-vous entre processus ont lieu conceptuellement entre groupes, mais ils ont en fait lieu entre deux processus appartenant à deux groupes différents. La distinction entre les variables locales et les variables partagées en Opal est moins évidente que la distinction entre les variables locales et les canaux de communication du modèle task-channel lors de l'écriture d'un programme et peut entraîner des erreurs subtiles.

Le but principal d'Opal est de fournir un langage parallèle modulaire. Les modules doivent pouvoir être écrits comme en Ada, avec une interface disponible aux utilisateurs des modules, et un corps qui est caché. Puisque le modèle de calcul d'Opal est le modèle BSP, il ne serait pas possible de fournir uniquement les interfaces de modules lorsqu'ils sont composés eux aussi de *supersteps* en ayant des processus synchronisés de la même manière. En effet, le nombre de *supersteps* nécessaires à l'exécution d'une routine serait alors inconnu et l'écriture d'un programme BSP avec plusieurs processus synchronisés deviendrait impossible. C'est pourquoi la notion de groupes de processus a été introduite. Un module externe appartient toujours à un groupe de processus qui lui est propre. Puisque les groupes sont exécutés de manière asynchrone entre eux, on peut cacher le corps des modules externes.

4.4 OCCAM-3

OCCAM-3 [24] est un langage parallèle qui succède à OCCAM et OCCAM-2. Il permet la déclaration de processus parallèles qui sont exécutés de manière asynchrone. Les divers processus communiquent entre eux au moyen de canaux. Dans les premières versions du langage, les canaux ne permettaient de connecter qu'un seul processus à un autre. L'une des innovations d'OCCAM-3 est la possibilité de créer des modules réutilisables. L'interface entre un processus et un module se fait aussi au travers de canaux de communication. Afin de permettre à ces modules d'être utilisés par plusieurs processus, les canaux ne sont plus limités uniquement au lien entre deux participants dans une communication. Un moyen de partager les canaux a donc été introduit, il rend possible la connexion de plusieurs processus à un module.

4.5 CARPET

CARPET et le langage suivant, Cellang, sont deux langages intéressants dans le domaine d'application visé par ParCeL-2, car ils permettent de décrire des programmes utilisant les automates cellulaires comme modèle de calcul. L'intérêt de fournir des langages parallèles utilisant les modèles d'automates cellulaires provient d'une part du fait que ces modèles offrent un cadre conceptuel très intéressant pour la simulation de la dynamique de phénomènes complexes, et d'autre part du fait que les automates cellulaires sont des modèles intrinsèquement massivement parallèles. CARPET (CellulAR Programming EnvironmenT) [25] est le langage de programmation utilisé dans l'environnement CAMEL (Cellular Automata environMent for systEms modeLing) [26]. Dans cet environnement, un ensemble de macro-cellules sont exécutées sur les processeurs disponibles, un processeur ne pouvant recevoir qu'une macro-cellule. Un des processeurs est le maître et exécute un processus contrôleur. Chaque macro-cellule est chargée de l'exécution d'une partie des cellules de l'automate cellulaire complet. Un système de communication permet l'échange de données entre les cellules. CAMEL fournit également une interface graphique pour la configuration d'un programme, ainsi que pour l'observation et la modification lors de l'exécution des paramètres de la simulation d'un système.

Le langage CARPET permet de décrire des problèmes qui peuvent être représentés par des automates cellulaires à voisinage carré ou hexagonal en deux ou trois dimensions. Un automate cellulaire est implanté au moyen d'un programme SPMD, c'est-à-dire que chaque cellule est un processus qui exécute le même code que toutes les autres, mais sur des données différentes. Un automate cellulaire est décrit au moyen de sa dimension, 2 ou 3, du rayon de voisinage des cellules et de l'état des cellules, qui est un constitué par un ensemble de sous-états. Chaque sous-état est un nombre, entier ou réel. Le voisinage d'une cellule est limité par le rayon choisi, il ne peut pas être quelconque. Enfin, il n'existe pas de canaux de communications différenciés entre les cellules. La seule information qu'une cellule peut obtenir d'une autre cellule est son état.

4.6 Cellang

Le langage Cellang [27] fait partie de l'environnement de simulation d'automates cellulaires Cellular [28]. Ce langage permet de définir des automates cellulaires à n dimensions, pas seulement à deux ou trois dimensions comme CARPET. Cependant, le seul type de données disponible est le type entier, ce qui limite l'intérêt de Cellang dans les domaines qui font appel au calcul en nombres réels. L'état d'une cellule est ainsi représenté par en ensemble de nombres entiers. Le voisinage d'une cellule peut être de taille quelconque, il n'est pas limité à

un rayon particulier, cependant, seuls les voisinages rectangulaires sont autorisés, les voisinages hexagonaux doivent être simulés par le programmeur.

Cellang offre un concept intéressant d'agent. Les agents permettent de représenter un ensemble de valeurs qui se déplace de cellule en cellule, par exemple une boule de billard qui roule sur une table de billard, chaque cellule représentant une portion de la surface de la table. Cela permet d'exprimer de manière claire le déplacement de particules, plutôt que de le cacher dans un algorithme plus compliqué décrivant l'évolution des cellules selon qu'une particule entre ou sort. Comme une cellule ne peut que consulter l'état de ses voisines, et non pas le modifier directement, les agents offrent un moyen de communiquer des données aux cellules voisines à l'initiative de la cellule d'où proviennent ces données. Chaque agent est associé à une seule cellule, et plusieurs agents peuvent être associés à la même cellule. Les agents cessent d'exister à la fin de chaque cycle de l'automate cellulaire. Pour avoir des agents qui réapparaissent lors du prochain cycle, ils doivent être propagés explicitement par les cellules. Comme dans CARPET, il n'existe pas de moyen de communication évolué entre les cellules, les seules informations transmissibles sont l'état d'une cellule voisine ou un agent provenant d'une cellule voisine.

4.7 CuPit-2

Contrairement aux deux langages précédents, qui se consacrent à la programmation d'automates cellulaires, CuPit-2 [29, 30] est destiné à la programmation de réseaux de neurones artificiels dynamiques. Les éléments de calcul, ou processus, sont des nœuds. Ils disposent de données locales et d'instructions agissant sur ces données, ainsi que des interfaces vers l'extérieur afin de pouvoir communiquer avec d'autres nœuds. Les nœuds peuvent être rassemblés en groupes, ou en couches pour reprendre la terminologie des réseaux de neurones artificiels. La structure de plus haut niveau dans un programme est le réseau, constitué d'un ensemble de groupes et de nœuds qui peuvent communiquer. Les communications se font au moyen de connexions entre des nœuds. Ces connexions sont analogues aux canaux de communication que l'on retrouve dans ParCeL-1 entre autres en ce sens qu'elles relient deux nœuds entre eux et véhiculent des données d'un type choisi par le programmeur mais constant au cours de l'exécution d'un programme. Cependant, les connexions ont une nature duale par rapport aux nœuds. En effet, une connexion n'est pas un canal « transparent », elle peut modifier les données qu'elle transmet. Les connexions disposent, comme les nœuds, de données locales et d'interfaces vers les nœuds qu'elles relient. Elles disposent en outre d'instruction locales permettant de modifier les données qui transitent par elles. Il est possible que plusieurs canaux débouchent au même point d'un nœud. Le problème d'écriture concurrente qui apparaît est résolu au moyen de l'exécution, par le nœud destinataire, d'une fonction de réduction qui retourne une seule valeur à partir de toutes celles qui ont été transmises. Dans le but de pouvoir programmer des réseaux de neurones reconfigurables, il est possible de créer et de supprimer des nœuds en cours d'exécution, de même qu'il est possible de créer, de supprimer, et de modifier les connexions entre les différents nœuds.

4.8 Sather

Sather [31, 32] est un langage qui se veut universel. Selon ses auteurs, il peut être décrit comme une tentative d'être aussi efficace que C, C++ ou FORTRAN, aussi élégant mais plus sûr qu'Eiffel ou CLU, et essayant de fournir des fonctions d'ordre supérieur aussi bien

que Common Lisp, Scheme ou Smalltalk. Il vise, en un certain sens comme le modèle BSP, le calcul à haute performance universel. C'est un langage très abouti qui offre des concepts évolués puisés dans le domaine du parallélisme, des langages à objets et des langages fonctionnels. Il offre notamment un ramasseur de miettes, une vérification des accès aux données lors de l'exécution, une gestion des exceptions, un concept d'itérateurs pour parcourir des collections d'objets et des fonctions d'ordre supérieur au moyen de clôtures.

Le parallélisme est introduit dans l'extension de Sather dénommée pSather. Ce dernier appartient à la catégorie des langages qui unifient les notions de processus et d'objets. L'orientation objet et le parallélisme sont exprimés de manière orthogonale dans pSather. La synchronisation des processus est toujours explicite, ce qui implique une exécution asynchrone des processus et ce qui est en opposition complète avec le modèle BSP. Les données peuvent être réparties sur les processeurs par le compilateur ou alors par le programmeur. L'environnement d'exécution fournit une mémoire partagée entre tous les processus.

4.9 SR

SR [33] est présenté comme un langage destiné à la programmation parallèle et distribuée. Un programme SR en exécution est composé de plusieurs machines virtuelles, réparties au besoin sur plusieurs processeurs, qui constituent des espaces d'adressage indépendants, et dans lesquelles plusieurs processus peuvent être exécutés. Les processus sont appelés des *globales* ou des *ressources* et sont similaires aux modules de Modula-2; ils sont donc composés de deux parties : une spécification et un corps. Les communications entre processus peuvent être effectuées au moyen de variables partagées, d'appels de procédures distantes, de rendez-vous ou de passage de message asynchrone. Ces divers modes de communication ont été introduits dans le but d'avoir un langage aussi général que possible. Cependant, le langage n'offre qu'un moyen d'expression bien adapté aux applications matricielles et à la répartition de celles-ci sur les différentes machines virtuelles, ce qui rapproche fortement ce langage de FORTRAN.

4.10 FORTRAN

FORTRAN est le langage traditionnel utilisé pour le calcul scientifique. Diverses extensions parallèles ont été proposeées, et un standard a vu le jour, High Performance FORTRAN ou HPF. Ce dernier est un langage ayant une orientation toute particulière pour les applications classiques de calcul numérique nécessitant de hautes performances, c'est-à-dire l'algèbre linéaire ou toutes le calcul matriciel. Le parallélisme est exprimé par la répartition des données des programmes, regroupées en tableaux, sur les différents processeurs. Le compilateur s'occupe alors d'essayer de regrouper au mieux les données et les instructions qui y accèdent sur les processeurs à disposition. FORTRAN est cependant moins bien adapté pour l'expression de calculs selon des modèles réguliers comme les modèles cellulaires ou les modèles d'éléments finis. Il s'agit en définitive d'un langage assez primitif qui ne survit que parce que de nombreux anciens programmes ont été écrits en FORTRAN.

4.11 Split-C

Il s'agit du langage développé pour le modèle de calcul LogP. Les programmes sont donc écrits pour un modèle déxécution SPMD avec un espace d'adressage global. On peut trouver une description de Split-C dans [34].

L'accès à la mémoire locale d'un processeur est exprimé différemment de l'accès à la mémoire globale, ou distante. Le langage fournit deux sortes de pointeurs : des pointeurs sur des objets locaux, qui ne peuvent repérer que des objets appartenant à l'espace d'adressage alloué à un processeur donné, ainsi que des pointeurs sur des objets de la mémoire globale, qui peuvent repérer des objets appartenant à l'espace d'adressage complet. Cela permet d'exprimer la différence de coût entre un accès à de la mémoire locale et un accès à de la mémoire distante. La syntaxe de Split-C est très proche de celle de C. Quelques mots-clefs et constructions ont été ajoutés pour faciliter l'expression des fonctions ci-dessous.

Un opérateur d'affectation spécial est fourni, qui permet de demander une valeur à la mémoire sans devoir attendre que cette valeur soit reçue pour continuer l'exécution du programme. Un autre opérateur est disponible pour avertir un processeur distant qu'une partie de sa mémoire a été modifiée par un autre processeur, celui sur lequel l'opérateur est appliqué. Les données transférées entre processeurs ne sont pas limitées aux types de base, elles peuvent être composées de types structurés. Split-C permet de définir des tableaux de valeurs et divers modes de répartition sur les processeurs qui sont assez proche de ceux des versions récentes de FORTRAN. Enfin, différents types de barrières de synchronisation existent, permettant de s'assurer que toutes les communications ont été effectuées, qu'une certaine quantité de données déterminée a été reçue par chaque processeur, ou, de manière plus classique, que tous les processus sont arrivés à un même point dans l'exécution de leur programme.

Split-C apparaît ainsi comme un langage principalement destiné aux applications matricielles, et dans ce cas il ressemble beaucoup à FORTRAN et ne présente que peu d'intérêt par rapport à ce dernier.

4.12 Jade

Il s'agit d'une extension du langage C extrêmement simple [19]. À l'aide de trois mots-clefs le programmeur ajoute des commandes permettant à son code d'être exécuté en parallèle. Il part du principe que le programme séquentiel annoté peut être divisé en un certain nombre de tâches à grain grossier. C'est une conception intermédiaire entre la parallélisation automatique laissée entièrement à la charge du compilateur, et la parallélisation explicite dans laquelle c'est au programmeur d'écrire un programme parallèle. Les indications que donne le programmeur sont simplement des informations au sujet de l'accès exclusif aux variables. L'accès en exclusion mutuelle peut être demandé en lecture seule, en écriture ou en lecture et écriture. Le compilateur peut alors déduire certaines dépendances de données et ainsi exécuter en parallèle certaines parties du programme.

4.13 ABCL/f

Pour pouvoir décrire ABCL/f [35], il est important de commencer par présenter ABCL/1. Ce dernier reprend le concept d'acteur mais le modèle de calcul retenu est moins uniforme que celui d'Agha. L'acteur demeure l'entité de base d'un programme, et les différents acteurs évoluent en parallèle. Cependant, ABCL/1 est une extension de Common-Lisp et les nombres ainsi que les structures de données élémentaires sont directement empruntés au LISP plutôt que d'être implantés au moyen d'acteurs. Cela dans un but d'efficacité et de facilité d'expression en utilisant un langage hôte connu.

Un acteur en ABCL/1 dispose d'une mémoire locale décrivant son état et d'un script décrivant ses différents comportements en fonctions des divers messages reçus. La présence de

mémoire locale est une nouveauté par rapport au modèle d'Agha. Un acteur peut modifier le contenu de sa mémoire locale sans pour autant entraîner de remplacement de comportement. Cela contribue à fournir un style de programmation plus procédural.

Il est possible de créer des acteurs nouveaux lors de l'exécution d'un programme. On peut créer un acteur en définissant son script au moment de sa création, ou alors en recopiant un acteur existant. L'envoi de messages est le seul moyen de communication entre acteurs et les acteurs ne sont activés que lors de la réception d'un message. En ABCL/1, les acteurs sont tous séquentialisés et ne traitent qu'un seul message à la fois. Cependant, contrairement au modèle d'Agha, il existe deux modes d'envoi : normal ou express, et trois types de messages : asynchrones, synchrones ou anticipés. Les différents types d'envois de messages permettent de contrôler plus finement le déroulement d'un programme parallèle. L'envoi d'un message express peut interrompre le calcul d'un acteur dans le cas où un autre acteur a déjà trouvé la solution par exemple. Enfin, la réception des messages est liée à des opérations de filtrage qui permettent de ne traiter que certains message sélectionnés.

ABCL/f sacrifie une partie du niveau d'abstraction d'ABCL/1 au profit d'une efficacité accrue lors de l'exécution. ABCL/f introduit ainsi le typage des données et la synchronisation explicite des accès aux variables *futures*. Le typage des données permet des optimisations lors de la compilation et le synchronisation explicite des accès aux variables *futures* permet d'éviter d'écrire un code dans lequel les acteurs sont bloqués en attente par nécessité de variables *futures*. Enfin, ABCL/f permet le placement des données par le programmeur, afin d'en optimiser le placement.

4.14 Langages fonctionnels, logiques

On ne parlera pas en détail des versions parallèles de langages fonctionnels ou logiques, on se contentera de donner quelques exemples.

Dans le cas des langages logiques, Strand est un dérivé de PROLOG dans lequel les sous-clauses sont exécutées en parallèle. L'arbre de recherche est alors parcouru en parallèle et la liaison de variables n'est autorisée qu'à partir de ce moment-là. Cela peut conduire à une explosion combinatoire que l'on peut contrôler à l'aide de sélections non-déterministes.

Dans le cas des langages fonctionnels, on a en général affaire à des fonctions sans effet de bord, ou à des lambda expressions. L'ordre d'évaluation, ou l'ordre de réduction des lambda expressions en forme normale, n'ayant alors plus d'importance, on peut envisager d'exécuter en parallèle les différents appels de fonction ou les différentes réductions. Les fonctions évaluées en parallèle sont en général celles correspondant aux arguments d'une autre fonction. Cela limite le nombre de processus qui peuvent être exécutés en parallèle car les fonctions que l'on trouve dans les programmes ont en général très peu d'arguments, environ deux ou trois seulement. Cependant, ces langages sont bien adaptés aux problèmes fortement dirigés par les données.

4.15 ParCeL-2

Les programmes visés par ParCeL-2 sont des programmes massivement parallèles à grain fin. C'est-à-dire que l'on aura affaire à un très grand nombre de processus qui exécutent des calculs relativement simples. En outre, puisque son modèle de calcul de est cyclique et synchrone, ParCeL-2 s'intéresse tout particulièrement aux domaines d'application des automates cellulaires, des différences finies et des éléments finis.

Un programme ParCeL-2 décrit les différents processus qui seront exécutés en parallèle, ainsi que les interconnexions entre les divers processus. Le nombre de processus ainsi que la topologie de communication entre ceux-ci doit être connue à la compilation. Il n'est pas possible de créer de nouveaux processus en cours d'exécution, ni de modifier les connexions entre processus. ParCeL-2 est donc un langage à parallélisme explicite et dans lequel les communications entre processus sont également explicites. Dans un souci de fournir un langage permettant de décrire des structures complexes de processus, on peut regrouper plusieurs processus en un groupe. Ce groupe sera vu par les processus externes comme un seul processus ayant un comportement plus complexe qu'un processus élémentaire. Les processus présents dans un groupe peuvent communiquer entre eux, mais pas directement avec les processus extérieurs au groupe, ils doivent utiliser le groupe comme interface de communication.

Étant donné que, comme on l'a vu plus haut, le modèle de calcul de ParCeL-2 est statique, et que par conséquent le nombre de processus est constant et la topologie d'interconnexion des processus est aussi constante, on peut représenter un programme ParCeL-2 comme un graphe dirigé dont les sommets sont les processus et les arcs sont les canaux de communication.

Une caractéristique importante de ParCeL-2 est que la description des calculs est séparée de celle des communications. On veut dire par là qu'un programme ParCeL-2 est composé de deux parties. Dans la première on trouve la déclaration des cellules avec le code à exécuter ainsi que le nombre et le type des canaux de communication. La seconde partie est en fait la description d'un graphe dirigé. Le format de description adopté est celui de l'environnement de gestion de graphes Graphlet [36, 37].

La séparation de la description du code des cellules de la description de la topologie permet d'éviter les traditionnels codes d'initialisation répétitifs qui ne font que créer des cellules et connecter les extrémités des canaux de communications entre elles au début d'un programme.

Finalement, ParCeL-2 ne cherche pas à définir un langage entièrement nouveau, il se présente plutôt comme un ensemble d'extensions à apporter à un langage pour permettre d'exprimer les concepts parallèles de ParCeL-2. Cela permet de bénéficier de l'effort de conception d'un langage de programmation qui a déjà été fait. Il convient bien évidemment d'imposer des restrictions à l'usage du langage de programmation de base pour le code des processus. Par exemple, transmettre un pointeur dans un canal de communication n'a pas de sens puisque chaque processus ne peut accéder qu'à sa mémoire locale. Pour l'instant on a défini une extension du langage C, mais on pourrait très bien envisager des extensions d'autres langages séquentiels comme Scheme ou Ada, et même des extensions de langages parallèles comme pSather, par exemple.

5 Conclusion

Dans ce document on a présenté divers modèles de calcul parallèles et divers langages de programmation utilisant ces modèles de calcul. On retiendra tout particulièrement le modèle BSP et les langages cellulaires comme source d'inspiration pour le développement du langage ParCeL-2 et du modèle de calcul qui lui est associé. On notera que le but principal de ParCeL-2 est de fournir un langage offrant une mise en œuvre efficace de programmes massivement parallèles à grain fin, ainsi que leur programmation de manière élégante. Dans cette optique on s'inspirera plus volontiers des travaux réalisés dans le domaine des langages les pour automates cellulaires et les réseaux de neurones que de ceux réalisés dans le domaine des langages fonctionnels ou de la parallélisation automatique par exemple.

Références

[1] Gary D. Doolen, editor. *Lattice Gas Methods for PDE's, Theory, Applications and Hardware. Proceedings of the NATO Advanced Research Workshop*. North-Holland, September 1989.

[2] Bastien Chopard and Michel Droz. Cellular automata modeling of physical systems. Cours COSMASE 1996, University of Geneva, February 1996.

[3] Stéphane Vialle. *ParCeL-1 : un langage parallèle d'acteurs autonomes synchrones*. PhD thesis, Université de Paris-Sud, Supélec, U.F.R. scientifique d'Orsay, PRiSM, June 1996.

[4] Susanne E. Hambrusch. Models for parallel computation. In *Proceedings of Workshop on Challenges for Parallel Processing, 1996 Internationl Conference on Parallel Processing*, pages 92–95, August 1996.

[5] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant. Bulk synchronous parallel computing — a paradigm for transportable software. In *Proceedings of the 28th Annual Hawaii Conference on System Sciences*, volume II. IEEE Computer Society Press, January 1995.

[6] Jonathan M. D. Hill. Bsp cost parameters, sorted by megaflop/s rate, for the oxford bsp toolset, September 1998.

[7] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in pram computation. In ACM, editor, *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, June 1989.

[8] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of prams. *Theoretical Computer Science*, pages 3–28, March 1990.

[9] Jonathan M. D Hill and David B. Skillicorn. Lessons learned from implementing BSP. In *High Perfomance Computing and Networking (HPCN'97)*. Springer-Verlag, April 1997.

[10] Ian Foster. *Designing and Building Parallel Programs, Concepts and tools for Parallel Software Engineering*. Addison Wesley, 1994.

[11] Pierre Kuonen. *Le Parallélisme*. EPFL, 1996.

[12] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Eicken von. Logp: Towards a realistic model of parallel computation. Technical Report UCB//CSD-92-713, University of California at Berkeley, Department of Computer Science, June 93.

[13] Gianfranco Bilardi, Kieran Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. Bsp vs logp. In *Proc. of the 8th ACM Symposium on Parallel Algorithms and Architecture (SPAA 96)*, pages 25–32, 1996.

[14] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. Loggp: Incorporating long messages into the logp model - one step closer towards a realistic model for parallel computation. In *7th Annual Symposium on Parallel Algorithms and Architectures*, Santa Barbara, CA, 1995.

[15] Joseph F. Jájá and Kwan Woo Ryu. The block distributed memory model. Technical Report CS-TR-3207, Computer Science Department, University of Maryland, January 1994.

[16] Gul Agha. *ACTORS, a model of concurrent computation in distributed systems.* MIT Press, 1986.

[17] Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott. Towards a theory of actor computation. In R. Cleaveland, editor, *Third International Conference on Concurrency Theory (CONCUR), Lecture Notes in Computer Science*, volume 630, pages 565–579. Springer-Verlag, 1992.

[18] Gul A.Agha, Ian A.Mason, Scott F.Smith, and Carolyn L.Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–69, January 1997.

[19] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in jade. In *Proc. Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, 1991.

[20] Jean-Marc Jézéquel. Parallélisme massif et langage à objets : une approche spmd. Technical Report 1607, INRIA-Rennes, Domaine de Voluceau Rocquencourt, B.P. 105, 78153 Le Chesnay CEDEX, February 1992.

[21] R. H. Perrott. Parallel language developments in Europe. *Concurrency Practice and Experience*, 4(8):589–617, 1992. Department of Computer Science, The Queen's University, Belfast.

[22] W. F. McColl. GL: An architecture independent programming language for scalable parallel computing. Technical Report 93-072-3-9025-1, NECI, June 1993.

[23] Simon Knee. Program development and performance prediction on bsp machines using opal. Technical Report TR-18-94, Programming Research Group, Oxford University, August 1994.

[24] Geoff Barrett. *Occam 3 Reference Manual.* INMOS Limited, 1992.

[25] G. Spezzano and D. Talia. Carpet: A programming language for parallel cellular processing. In *Proceedings 2nd Europ. School on PPE for HPC*, pages 71–74, April 1996.

[26] M. Cannataro, S. Di Gregorio, R. Rongo, W. Spataro, G. Spezzano, and D. Talia. A parallel cellular automata environment on multicomputers for computational science. *Parallel Computing*, 21:803–823, 1995.

[27] J. Dana Eckart. Cellang 2.0: Language reference manual. *ACM SIGPLAN Notices*, 27(8), August 1992.

[28] J. Dana Eckart. A cellular automata simulation system: Version 2.0. *ACM SIGPLAN Notices*, 27(8), August 1992.

[29] Holger Hopp and Lutz Prechelt. Cupit-2: A portable parallel programming language for artificial neural networks. In *1997 IMACS Conference on Applications of Computer Algebra*, 1997.

[30] Holger Hopp and Lutz Prechelt. Cupit-2 – a parallel language for neural algorithms: Language reference and tutorial. Technical Report 4/97, Institut für Programmstrukturen und Datenorganisation, March 1997.

[31] David Stoutamire and Stephen Omohundro. *Sather 1.1.* ICSI Berkeley; NEC Research Institute, August 1998.

[32] Benedict Gomes, Holger Klawitter, David Stoutamire, and Boris Vaysman. *Sather 1.1: A Language Manual.* ICSI Berkeley, June 1996.

[33] Ronald A. Olsson, Gregory R. Andrews, Michael H. Coffin, and Gregg M. Townsend. Sr a language for parallel and distributed programming. Technical Report 92-09, Department of Computer Science, University of Arizona, Tucson, March 1992.

[34] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in split-c. In *Proceedings of Supercomputing '93*, pages 262–273, Portland, November 1993.

[35] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/ f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation. In G. Blelloch, M. Chandy, and S. Jagannathan, editors, *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, number 18 in Dimacs Series in Discrete Mathematics and Theoretical Computer Science, pages 275–292. American Mathematical Society, American Mathematical Society, 1994.

[36] Michael Himsolt. *GML: Graph Modelling Language.* Universität Passau, 94030 Passau, Germany, December 1996. Draft Version.

[37] Michael Himsolt. Gml: A portable graph file format. Technical report, Universität Passau, 94030 Passau, Germany, 1996.