Une Interface C++ pour programmer sur la connection machine

Philippe R.B. Devloo, Loula Fezoui, Stéphane Lacire

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Rapports de Recherche

N°1698

Programme 6
Calcul scientifique, Modélisation et Logiciel numérique

UNE INTERFACE C++ POUR PROGRAMMER SUR LA CONNECTION MACHINE

Philippe R.B. DEVLOO
Loula FEZOUI
Stéphane LACIRE

Mai 1992

Institut National
de Recherche
en Informatique
et en Automatique

Sophia Antipolis
B.P. 109
06561 Valbonne Cedex

UNE INTERFACE C++ POUR PROGRAMMER SUR LA CONNECTION MACHINE

Philippe R.B. DEVLOO* Loula FEZOUI†
Stéphane LACIRE†

*Departemento de Mecânica Espaciale Controle Instituto de Pesquisas Espaciais C.P. 515 CEP 12201 São José dos Campos -SP SP - BRASIL.

†INRIA Sophia-Antipolis, 2004, Route des Lucioles, 06560 VALBONNE, FRANCE.

UNE INTERFACE C++ POUR PROGRAMMER SUR LA CONNECTION MACHINE

Philippe R.B. Devloo, Loula Fezoui, Stéphane Lacire

RESUME: On décrit ici un ensemble de classes C++, noyau d'une interface à la librairie PARIS de la Connection Machine. Des exemples numériques simples: opérations arithmétiques (produit scalaire, SAXPY, ..) et la résolution d'un système linéaire par Jacobi sont utilisées pour valider l'interface. On compare les performances de l'interface C++ avec celles de la librairie C/PARIS ainsi qu'avec celles obtenues en utilisant deux langages évolués de la Connection Machine: C* et CM-FORTRAN.
Un cas test classique en écologie est présenté pour illustrer les performances du calcul parallèle avec des entiers.

A C++ INTERFACE FOR PROGRAMMING ON THE CONNECTION MACHINE

ABSTRACT: In this work, a set of C++ classes are described which provide the Connection Machine programmer with a simplified interface to the PARIS library. Numerical test cases such as parallel arithmetic operations (scalar product, SAXPY,..) and a Jacobi iterative resolution, are used to perform a validation of the C++ interface. Performance results of the C++ interface are presented and compared to those obtained using the PARIS library or using high-level langages of the CM-2 system such as C* or CM-FORTRAN.
A classical example in numerical ecology is used for illustrating the performance in parallel computation with integers.

Introduction

Ces dernières années ont vu apparaître de nombreuses machines parallèles très diverses dans leur architecture et leur réseau de communication. De plus, les langages de programmation de haut niveau disponibles sur ces machines sont des extensions de langages classiques adaptées aux particularités de l'architecture considérée. Cela rend délicat et parfois coûteux l'adaptation de codes existants et leur transfert d'une architecture vers une autre. Malgré la convivialité (relative) de la Connection Machine, la conception et l'implantation d'algorithmes massivement parallèles nécessitent parfois beaucoup d'efforts notamment pour définir une répartition adéquate des données de calcul sur les milliers de processeurs locaux.
D'autre part, l'engouement actuel pour la programmation orientée-objet et le **C++** en particulier, montre l'intérêt des informaticiens et des numériciens pour une convivialité accrue et une plus grande facilité de programmation sans sacrifier la portabilité des codes obtenus. Nous présentons ici une tentative de construction, à l'aide de **C++**, d'une interface de programmation entre d'une part un langage classique de haut niveau, le **C**, et d'autre part **PARIS** (PARallel Instruction Set) sur la CM2. La bibliothèque que nous proposons ici est liée d'une part à certaines particularités de l'actuelle CM2 (parallélisme massif des données, l'existence de PARIS,...) et sert d'autre part d'interface avec le langage **C**. Nous pensons cependant que des extensions vers d'autres langages de haut niveau et d'autres machines sont possibles et réalisables.
Le présent rapport est constitué de trois parties:
Dans la première partie, nous rappelons brièvement les caractéristiques de la Connection Machine **CM-2**. La deuxième partie est consacrée à la description de la bibliothèque C++ à partir de notions liées au parallélisme massif d'une part et aux idées de base de la programmation orientée–objet (classes, surcharge, polymorphisme ...) d'autre part. Dans la troisième partie, nous présentons des tests de validation de la bibliothèque et de comparaison avec d'autres langages. Des applications numériques simples ont été choisies pour mieux rendre compte des performances comparées entre les langages de haut niveau, PARIS et la bibliothèque C++. Ces applications sont les suivantes:

- opérations arithmétiques de base.
- résolution d'un Laplacien.
- un exemple simple de simulation en écologie.

1 La Connection Machine.

1.1 Présentation de la CM-2.

L'unité de calcul parallèle. L'unité de calcul parallèle est le coeur de la CM. Il est constitué de milliers de processeurs (ici 16384) qui peuvent être interconnectés de façon logicielle permettant ainsi les échanges d'information entre les processeurs. Chaque processeur possède sa propre mémoire (256 Kbits, temps d'accès de 5Mb/s) et une unité arithmétique et logique (ALU) d'architecture RISC. Chaque élément d'une variable parallèle est stockée dans la mémoire d'un processeur, au même endroit sur tous les processeurs. Ainsi, les opérations sur une variable sont effectuées en parallèle pour tous les éléments la composant. L'organisation interne de la machine peut se résumer brièvement comme suit:

- 16 processeurs $\Leftrightarrow$ 1 chip.
- 2 chips $\Leftrightarrow$ 1 Weitek.
- 2 Weiteks $\Leftrightarrow$ 1 board.
- 16 boards $\Leftrightarrow$ 1K processeurs.

Les chips sont disposés en hypercube. Le Weitek est un accélérateur de calcul flottant disposant de sa propre mémoire.

Le Front End. L'autre élément fondamental de la CM est constitué par l'ordinateur frontal (ou plusieurs selon la configuration). Certaines variables sont calculées de façon séquentielle; de telles variables résident dans la mémoire du frontal. L'unité de calcul parallèle peut alors être vue comme une extension intelligente de la mémoire du frontal.

C'est le frontal qui exécute le programme, qui dialogue avec le programmeur, et c'est lui qui envoie les ordres à l'unité de calcul parallèle par l'intermédiaire d'un séquenceur : le séquenceur décode les commandes venant du frontal et les diffuse sur tous les processeurs. On peut cependant rendre un processeur inactif, c'est à dire, pour certaines instructions, ne les faire exécuter que par certains processeurs (dits actifs). On définit alors un contexte (ensemble des processeurs actifs).

Les autres périphériques

CM I/O System. Le système d'entrée/sorties de la CM est organisé comme un ensemble de processus client-serveur tournant sur un ou plusieurs ordinateurs reliés par une liaison Ethernet.

Le Data Vault. Le Data Vault est le système de stockage de masse de la CM. Il peut contenir 10Gb de données, extensibles jusqu'à 20 Gb. La vitesse de transfert se fait à 25 Mb/s. Des Data Vaults travaillant en parallèle peuvent atteindre 100 Mb/s.

Les outils de visualisation graphique. Pouvoir visualiser les données pendant le calcul, peut aider à une meilleure compréhension du problème. Les outils graphiques de la CM permettent des animations en temps réel ainsi que des affichages rapides et de grande qualité. Un "Framebuffer" directement connecté à l'unité de calcul parallèle peut transférer des informations graphiques des processeurs vers un moniteur couleur haute résolution à une vitesse de 40 Mb/s. De plus, un protocole permet à un programme s'éxecutant sur la CM2 d'envoyer ses sorties graphiques sur n'importe quelle station de travail supportant le système X Window.

Les processeurs virtuels. Dans la configuration maximale, la machine dispose de 65536 processeurs physiques; si le problème en demande plus, le système travaille alors en mode **processeurs virtuels**, fournissant ainsi un plus grand nombre de processeurs, mais chacun disposant d'une quantité moindre de mémoire. Ce processus est géré automatiquement par le système et il est donc transparent pour le programmeur. Sa mise en oeuvre repose sur le partage de la mémoire du processeur physique en autant de zones que de processeurs virtuels désirés. L'exécution est alors séquentielle sur l'ensemble de processeurs virtuels attachés à un processeur physique. Un ensemble de variables parallèles est appelé **VP**set. Le rapport du nombre de processeurs virtuels sur le nombre de processeurs physiques est appelé le **VPR**atio.

Les communications. Le sytème fournit 3 modes de communications entre les différents processeurs: le **routeur**, les **NEWS** et le **scanning**.

Les routeurs. Ces communications se font par l'intermédiaire d'éléments de commutation appelés routeurs; c'est un mode de communication générale,

tous les processeurs peuvent communiquer entre eux. Il permet à chaque processeur d'envoyer un message à plusieurs autres processeurs en parallèle. Il y a un routeur par chip (16 processeurs). Une machine de 64K a donc 4096 routeurs disposés sur un N-cube (N = 12). Il existe N liaisons entre deux routeurs quelconques, et $N * 2^{N-1}$ liaisons au total.

Les NEWS. La grille NEWS (North, East, West, South) est un mode de communication très rapide mais plus structuré : c'est un réseau de voisinage immédiat où un proceseur peut communiquer avec ses quatres voisins cardinaux. Ce type de communication est très performant sur des applications adaptées où l'utilisateur a défini auparavant une grille structurée de processeurs pour ses données. La CM-2 reconnait des grilles jusqu'à 31 dimensions.

L'opération de scanning. Le "scanning" est un mode de communication très puissant qui fonctionne sur une grille NEWS; il combine à la fois les communications et les calculs. Ainsi, les données se déplacent de proche en proche tout en étant des opérandes dans chaque processeur. Les communications par NEWS et "scanning" sont très rapides mais subordonnées à des grilles de calcul structurées.

1.2 Les langages de programmation

Les langages parallèles permettent au programmeur d'organiser ses variables de façon à ce que les opérations puissent s'exécuter sur plusieurs éléments de la variable en même temps. De ce fait, il y a certaines différences entre la programmation parallèle et la programmation séquentielle. Le but d'un bon langage de programmation parallèle est de fournir au programmeur des fonctionnalités qui lui permettent de ne pas avoir à spécifier lui-même les opérations parallèles, la synchronisation ... Par exemple, l'instruction A=B+C qui en séquentiel ajoute le nombre B au nombre C et stocke le résultat dans A doit pouvoir signifier aussi des milliers d'additions simultanées si A, B et C ont été déclarées comme variables parallèles. Le choix des structures parallèles est un aspect très important de la programmation parallèle. Si elles ont été correctement choisies, la programmation vient ensuite naturellement.

Les opérations sur les variables parallèles sont locales à chaque processeur si tous les éléments nécéssaires sont dans la mémoire de chaque processeur, ou peuvent nécessiter des communications entre les processeurs. Les opérations concernant à la fois des variables parallèles et des variables scalaires nécessitent des opérations de diffusion (diffusion de la variable scalaire sur les processeurs)

ou de réduction (résultat dans la variable scalaire). La CM dispose actuellement de trois langages de programmation de haut niveau : CM Fortran, C* et *Lisp.

Le CM Fortran. Le CM Fortran a été construit à partir du Fortran 77 ANSI et du Fortran 8x ANSI (appelé 90 depuis). Il utilise directement les caractéristiques du Fortran 8x pour les manipulations de tableaux, à la différence que chaque élément du tableau est stocké dans un processeur virtuel différent. Les tableaux correspondent directement aux grilles de communications multidimensionnelles de la CM. Des adaptations au mecanisme particulier de la machine en font cependant un langage qui ne s'identifie pas à ce qu'est ou sera le Fortran 90 une fois adopté.

Le C*. C'est une extension du langage C auquel on a ajouté les notions de variables parallèles et de programmation parallèle. C* utilise la plupart des opérateurs C en les adaptant aux variables parallèles. L'ancien C* utilisait de nouveaux types de variables (*mono* pour une variable scalaire et *poly* pour une variable parallèle); la nouvelle version (6.0) utilise la notion de grille sur laquelle sont définies les variables.

Le *Lisp. Extension du Common Lisp, il fournit des primitives qui correspondent directement aux opérations codées de la machine; il permet au programmeur de construire ses propres abstractions à partir des primitives. *Lisp restitue des codes objets très performants mais il reste en général largement plus utilisé en calcul symbolique que numérique.

PARIS. PARIS est un ensemble de macro-instructions parallèles: PARallel Instruction Set. On le considère comme un langage de bas niveau qu'il faut néanmoins distinguer des micro-instructions exécutées par le séquenceur et des nano-instructions exécutées directement par les processeurs. Il joue le rôle d'une bibliothèque vis à vis des langages de haut niveau et permet des opérations sur les entiers (signés et non signés), les réels et les complexes ainsi que des fonctions de communication et de transfert des données entre les processeurs et le frontal.

2 Construction de la bibliothèque C++.

La construction d'une biblitothèque en langage C++ avait plusieurs motivations principales. La première était de faciliter à l'utilisateur débutant l'accès à la Connection Machine, en lui permettant de programmer dans un langage classique, le langage C; la bibliothèque jouant le rôle d'interface entre C et l'ensemble de macro-instructions PARIS.
On peut penser qu'on définit ainsi un C* bis. Ceci n'est pas le cas, car C* est une extension de C, spécifique à la CM, et évoluant avec elle. Dans le cas d'une extension de type appel à une bibliothèque, seule celle-ci est susceptible d'évoluer avec la machine. De plus, on peut construire sur le même modèle d'autres bibliothèques ou des sous-ensembles de cette bibliothèque, permettant d'adapter et de traduire le code initial en vue d'une exécution sur plusieurs machines.
La deuxième motivation était liée à la performance de C*, qui était, jusqu'à la nouvelle version livrée en 1991, la plus faible comparée à *Lisp, CM-Fortran et PARIS, alors que les premiers tests réalisés avec l'utilisation de la bibliothèque C++ donnait des performances de même niveau que les autres langages et ,dans des cas particuliers, on retrouvait même la performance de PARIS.
Nous devons remarquer cependant que ces langages ainsi que l'ensemble du système évolue très vite et ainsi le nouveau C* est bien différent de l'ancienne version qui a justifié en partie ce travail. Les comparaisons numériques que nous présenterons dans la deuxième partie de ce rapport illustreront ce propos.

2.1 La programmation orientée objet.

La programmation orientée objet offre de nouvelles possibilités au programmeur, matérialisées par les notions de classe, de constructeurs/destructeurs, de surcharge d'opérateurs, de polymorphisme, d'héritage etc ...

2.1.1 Les classes.

La notion de classe constitue la base de la programmation orientée objet. En fait, c'est une notion semblable à celle de *struct* en C, mais, contrairement à celle-ci, une classe peut avoir des fonctions comme membre (appelées alors méthodes). Ces fonctions utilisent généralement les variables définies dans la classe. Les différentes instances d'une classe sont appelées **objets**.
Les classes ont comme particularité majeure le fait que leurs membres (variables ou fonctions) ont trois niveaux de protection. Les données publiques sont accessibles partout dans le programme. Les données privées ne sont accessibles qu'à partir des méthodes associées à la classe. Enfin, les données protégées ne

peuvent être utilisées que par les fonctions de la classe ou des classes dérivées.

2.1.2 Constructeurs/Destructeurs.

Quand un objet est déclaré dans un programme, une méthode particulière est appelée au niveau de la déclaration. Cette méthode se nomme le constructeur de la classe, elle se déclare comme toute autre fonction et consiste à initialiser les variables associées à la classe (réservation mémoire par exemple). Le constructeur peut être déclaré avec des paramètres, qui servent généralement de valeurs initiales aux variables.

De la même façon, quand un objet sort d'une portée, un destructeur est appelé. Il effectue toutes les opérations nécessaires avant que la mémoire associée à l'objet soit désallouée.

exemple de la classe **floatpar** (flottant parallèle) :

```
class floatpar{

private:

CM_field_id_t     id;          pointeur mamoire
unsigned int      sign, exp;   nb de bits pour la mantisse et l'exposant

public:

floatpar(float b);             constructeur de la classe
 floatpar();                   destructeur de la classe
```

2.1.3 La surcharge d'opérateurs.

On peut déclarer des opérateurs entre deux objets de type quelconque. Le résultat est le même que pour une fonction, mais l'utilisation d'opérateurs est plus naturelle. Exemple de déclarations:

```
floatpar operator+(floatpar &b);
floatpar operator*(floatpar &b);
floatpar operator=(floatpar &b);
```

Selon le type des objets concernés, l'opération sera scalaire ou parallèle.

```
floatpar a(), b(0.), c(1.), d(3.);

a = b * (c*d + a);
```

Ceci permet de laisser le programmeur manier aisément et de manière naturelle des variables parallèles.

2.1.4 Le polymorphisme.

Souvent on voudrait donner le même nom à des fonctions qui font les mêmes opérations, mais qui diffèrent dans le type des variables utilisées ou dans le nombre de paramètres transmis. Dans la plupart des langages, ceci n'est pas permis; le polymorphisme permet à des fonctions de même nom d'être déclarées avec des paramètres différents. Selon les arguments transmis, le compilateur décide quelle fonction sera appelée à l'exécution. Exemple :

```
int abs(int a);
float abs(float b);
floatpar abs(floatpar c);
```

Le programmeur n'a pas a retenir de nouveaux noms de fonctions pour utiliser des variables parallèles. La surcharge d'opérateurs est un cas de polymorphisme.

2.1.5 L'héritage.

L'héritage est une caractéristique des langages orientés objets qui permet à une nouvelle classe de bénéficier de toute la structure d'une autre classe dont elle hérite. Ceci permet de réutiliser des classes déjà définies pour d'autres problèmes et de les modifier pour les adapter en créant une classe dérivée (une classe dérivée peut redéfinir des fonctions de la classe mère). Ainsi, on peut spécialiser de plus en plus les classes tout en gardant les généralités de la classe de base.

2.2 Description des classes implémentées

Les classes implémentées actuellement sont les classes de base indispensables à l'écriture d'un programme parallèle sur la Connection Machine. L'ensemble des classes défini ici n'est pas exhaustif et l'on pourra facilement rajouter de nouvelles classes et méthodes...

Pour écrire un programme parallèle sur la CM, certaines notions sont indispensables :

- définir une géométrie (disposition des processeurs)
- pouvoir définir des processeurs virtuels.
- définir des variables parallèles (réel, entier signé ou non signé)
- pouvoir faire des opérations sur ces variables
- définir un contexte (ensemble des processeurs actifs).

D'où les classes de base suivantes :

2.2.1 La classe geometry.

Une des caractéristiques de la Connection Machine est de pouvoir organiser les processeurs selon une géométrie donnée, permettant ainsi de faire "coller" l'organisation des processeurs avec le problème posé (par exemple, une image 2D sera représentée sur une grille 2D avec un processeur par pixel). Les seules restrictions sont les suivantes :

- la longueur d'un axe doit être paire.
- Le produit de toutes les longueurs doit être un multiple du nombre total de processeurs physiques disponibles.

Si la géométrie ainsi définie est de structure régulière on peut bénéficier des avantages de la communication par les NEWS.

La classe *geometry* permet les opérations suivantes :

- créer une géométrie en vérifiant les contraintes énumérées ci-dessus.
- disposer de renseignements divers tels que le nombre de processeurs, le vp-ratio, les coordonnées, etc ...
- utiliser les communications NEWS.
- effectuer des opérations de scanning.

2.2.2 La classe vpset.

Pour qu'une géométrie soit valide, on a vu que le nombre de processeurs qu'elle représente (produit des longueurs) doit être un multiple du nombre de processeurs physiques disponibles. En effet, pour disposer de plus de processeurs que la limite physique, le système va découper un processeur physique en plusieurs processeurs virtuels. Cette nouvelle configuration s'appelle VPset (Virtual Processor set). Cette facilité permet par exemple à un programme écrit pour une CM 16K de fonctionner sur une CM 8K.

Description de la classe :

- VPset(geometry &geo) : créateur de la classe, définit un VPset.
- VPset() : destructeur de la classe.
- set() : ce VPset est le VPset actif (on peut changer de VPset durant une exécution).
- set_geom(geometry &geo) : affecte la géométrie geo au VPset.

2.2.3 La classe floatpar.

La classe *floatpar* définit des variables réelles parallèles ainsi que les opérations (principalement arithmétiques) s'y rapportant. Un floatpar est défini par trois variables :

- son emplacement mémoire sur le processeur (*field-id*).
- le nombre de bits pour l'exposant (généralement 8).
- le nombre de bits pour la mantisse (généralement 23)

Bien sûr, l'ensemble des méthodes définies dans cette classe n'est pas complet, on peut définir toutes sortes d'opérations entre les variables. Cependant, les principales opérations sont définies. La classe floatpar comporte actuellement environ 60 méthodes, parmi lesquelles :

- les opérations arithmétiques de base : +, -, *, /, =.
- des opérations plus complexes comme mult_const_add (f1 = f2*a +f3).
- des opérations de réduction : +=
- des comparaisons : ==, <, <= ...

- des communications : get_news, send_news, scan ...
- min, max, random, read_proc (lecture d'un processeur particulier) ...

2.2.4 Les classes intpar et unspar.

La classe intpar définit des entiers signés parallèles et la classe unspar des entiers non signés parallèles. La plupart des méthodes définies dans la classe *floatpar* le sont aussi pour les classes *intpar* et *unspar*. Cependant, les suites d'opérations prédéfinies comme mult_const_add, mult_add ... ne le sont pas pour les entiers. En effet, si elles ont été implémentées pour les réels c'est que leur équivalent existait en C/Paris, permettant de tirer profit au maximum de l'accélérateur virgule flottante de la CM-2.

2.2.5 La classe cond.

La classe cond permet de définir des contextes, c'est à dire rendre certains processeurs inactifs afin qu'ils n'exécutent pas certaines opérations. Sur chaque processeur existe un bit appelé **Context Flag**. Si ce bit est à 1, le processeur est actif.
Un objet de la classe cond est une variable parallèle définie par deux bits :

- un bit *my-bit* représentant le context flag.
- un bit *saved-context* pour sauvegarder le contexte précédent.

Pour définir un contexte, il faut imposer une condition et ne prendre que les processeurs vérifiant cette condition. Toutes les opérations de comparaison définies pour les classes floatpar, intpar et unspar génèrent des objets de type cond.
Exemple :

```
cond ex;
floatpar a, b, c, d;

ex = (a > b)||(c < d);
```

Seuls les processeurs vérifiant $a > b$ ou $c < d$ auront la valeur 1 dans le bit nommé ici *ex*. Ainsi, il ne reste plus qu'à charger le Context Flag avec ce bit pour rendre effectivement ces processeurs actifs. Le cond fonctionne comme le *if* du C à la différence qu'il s'applique ici à des variables parallèles.
Les méthodes définies :

- void set() : charge le context flag avec le bit *my-bit* et sauvegarde l'ancien contexte dans *saved-context* (instruction if pour une variable scalaire).
- void invert() : inverse le bit *my-bit* et donne donc le contexte opposé du précédent (instruction else pour une variable scalaire).
- void keep() : *saved-context* est initialisé au contexte actuel généré par les set(). Il n'y aura donc pas de retour au contexte précédent possible.
- int num_proc() : retourne le nombre de processeurs actuellement actifs.

On retouve l'ancien contexte lorsque le destructeur de l'objet cond est appelé, et donc lorsqu'on sort de la portée de l'objet (caractérisée par }). Trois opérateurs liant des objets de type cond ont aussi été définis : && (et logique), || (ou logique) et = (affectation).

La gestion du contexte est très délicate et un mauvais contexte peut entrainer des résultats imprévisibles, c'est pourquoi certaines règles sont à suivre :

- Set() ne doit être appelé qu'une seule fois par objet.
- Après le set(), seul un invert() peut être appelé. On peut cependant faire des affectations après le set(), mais on ne retrouvera pas le contexte de départ à la sortie de la portée.
- On peut imbriquer plusieurs "cond", mais il ne faut pas oublier les indicateurs de portée ({ et }).

On a essayé de faire ressembler l'utilisation du cond à celle du if en C. Voici la syntaxe du cond :

```
{  cond cd ((a > 0) && (a < b));
cd.set();
{
   ...
}
cd.invert();
{
   cond cd ((a > b) && (a < c));
   cd.set();
   {
      ...
   }
```

```
    cd.invert();
    {
      ...
    }
  }
  }
```

Utilisation du cond dans une boucle à contexte décroissant (exemple du crible d'Eratosthène):

```
cond cd(u >= 2);
cd.set();

while (nonfini)
{
  n = min(u);
  v = u mod n;
  cond cd(v > 0);
  cd.set();
  cd.keep();
}
```

2.2.6 Autres classes.

D'autres fichiers sont présents dans la bibliothèque afin d'aider le progammeur : la classe *cmtimer* pour les mesures de temps d'exécution (temps CM et temps frontal) avec start, stop, reset etc... et un fichier conversion.hpp, qui ne définit pas de classe mais des méthodes permettant de faire des conversions entre les différentes variables parallèles.

3 Applications numériques.

3.1 Opérations arithmétiques élémentaires

3.1.1 Conditions expérimentales.

Les expériences numériques présentées ici ont été effectuées sur une Connection Machine modèle CM-2 de 16K processeurs, système version 6.0. Le frontal est un Sun 4/390 donnant accès à 8K processeurs. Les tests ont donc étés effectués sur 8K avec des VPRatio de 1 (8K processeurs) et de 16 (128K processeurs), ceci pour quatre langages disponibles sur la CM-2 pour les calculs flottants : PARIS, CM-Fortran, C* et C++ (nous n'avons pas utilisé le *Lisp). Les chiffres donnés sont des performances calculées à partir du temps d'exécution CM (nombre d'opérations divisé par le temps), elles sont données en Mégaflops pour les opérations portant sur des flottants. Le temps CM est le temps d'exécution sur l'unité de calcul parallèle, et non pas le temps réel, temps d'exécution total sur le Front End, qui dépend de la charge du frontal.
Pour obtenir des temps d'exécution significatifs pour une évaluation correcte des performances, il a fallu exécuter plusieurs fois les instructions du tableau (ici 5000 fois).

Les résultats sont en Mflops pour les opérations sur les réels. On rappelle que le Mflop est le nombre de millions d'opérations en virgule flottante effectué en une seconde de temps CPU.

Langages et compilateurs:

- C/PARIS version 6.0
- g++ version 1.39.1 (basée sur GCC 1.39) avec optimisations.
- C* version 6.0.2 pour sun4 avec option d'optimisation -O2.
- CM Fortran version 1.1 avec option d'optimisation -O.

3.1.2 Résultats.

Performances pour des opérations sur des réels (en Mflops):

Opération	VPR	C/PARIS	C++	C*	CMF
$zi = xi + yi$	1	153	158	155	146
$zi = xi + yi$	16	237	238	239	231
$zi = xi * yi$	1	168	164	168	158
$zi = xi * yi$	16	273	273	273	214
$zi = xi * a + yi$	1	273	277	276	267
$zi = xi * a + yi$	16	432	436	434	422

Performances pour des opérations sur des entiers (même calcul que pour les Mflops):

Opération	VPR	C/PARIS	C++	C*	CMF
$zi = xi + yi$	1	212	154	209	189
$zi = xi + yi$	16	195	192	280	190
$zi = xi * yi$	1	7	7	7	7
$zi = xi * yi$	16	7	7	7	7
$zi = xi * a + yi$	1	128	58	113	18
$zi = xi * a + yi$	16	149	81	156	18

Au vu de ces résultats on peut faire les remarques suivantes:

- Les temps d'exécutions peuvent être très importants lorsque les opérations portent sur des entiers, pour les multiplications en particulier. Ceci est dû au fait que les opérations sur les entiers s' effectuent sur les processeurs "bit à bit", alors que les réels utilisent l'accélérateur de virgule flottante. Pour nos tests sur la bibliothèque C++, nous avons essayé la suite d'opérations suivante :conversion entier-réel, calcul en virgule flottante, conversion réel-entier; mais les conversions prennent beaucoup de temps, et le gain reste minime. On peut remarquer aussi une mauvaise performance de C/PARIS pour l'addition d'entiers à VP = 16 par rapport à C*. Le code C/PARIS produit par le compilateur C* semble identique au code écrit pour le test de C/PARIS. Ce résultat "curieux" reste inexpliqué.

- Les performances obtenues avec des codes écrits dans un langage évolué augmentent si on fait appel aux fonctions spécialisées de PARIS.

Pour réaliser par exemple l'opération $zi = xi * a + yi$ il y a l'ordre PARIS: *CM_f_mult_const-_add_1L* qui utilise toutes les possibilités "hardware" de la machine pour effectuer le calcul, sans passer par les phases d'addition, de multiplication puis d'affectation. Le nouveau C* est en ce sens très puissant car il trouve de lui même les optimisations nécessaires (voir les résultats ci-dessus). Dans la première version de la bibliothèque C++, cette optimisation se fait manuellement (par l'utilisateur) en appelant des fonctions particulières comme *z.mult(x,y)* par exemple. CM-Fortran génère un code un peu plus lent malgré l'optimisation. Celle-ci ne semble pas opérer sur les entiers; ainsi pour l'opération "x*a+b" l'exécution s'est faite sur le frontal.

3.2 Résolution d'un Laplacien

On utilise l'équation de Laplace en deux dimensions:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \ , \tag{1}$$

comme modèle simple de calcul numérique.

3.2.1 Approximation numérique.

On veut résoudre l'équation sur une grille rectangulaire.
On pose:

$$u_{i,j} = u(i\Delta x, j\Delta y)$$

On utilise un schéma aux différences finies à 5 points pour discrétiser l'équation:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{(\Delta x)^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{(\Delta y)^2} = 0 \tag{2}$$

Le but de l'expérience étant de comparer les performances obtenues avec différents langages, on considère un maillage en carrés: $(\Delta x = \Delta y)$

Le schéma s'écrit alors:

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = 0 \tag{3}$$

On peut remarquer que les points qui interviennent dans ce schéma sont les voisins Nord, Est, Sud et Ouest du point courant $\{i, j\}$.
L'implémentation de l'algorithme sur la CM-2 est alors très simple. On affecte à chaque processeur la valeur de u au point $\{i, j\}$, les valeurs aux points voisins sont obtenues par communication à l'aide de NEWS.

3.2.2 Résolution par Jacobi.

Le problème étant posé dans un domaine borné (ici le carré [0,1]*[0,1]), on ajoute une condition aux limites (la donnée sur la frontière du domaine) pour obtenir l'unique solution du problème:

$$\begin{cases} \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2} = 0 \quad x, y \in]0,1[\times]0,1[= D \\ u(x,y) = g \qquad\qquad\qquad\qquad x, y \in \partial D \end{cases} \tag{4}$$

où ∂D désigne la frontière du carré D.
En écrivant l'équation (4) en chaque point de la grille, on aboutit à un système linéaire:

$$\begin{cases} a_{11}u_1 + a_{12}u_2 + ... + a_{1n}u_n = c_1 \\ a_{21}u_1 + a_{22}u_2 + ... + a_{2n}u_n = c_2 \\ \qquad\qquad ... \\ \qquad\qquad ... \\ a_{n1}u_1 + a_{n2}u_2 + ... + a_{nn}u_n = c_n \end{cases} \tag{5}$$

où les a_{ij} et les c_i sont des coefficients connus et les $\{U_i\}_{i=1,n}$ désignent les inconnues du problème (n est le nombre total de points de la grille). La condition aux limites est prise en compte dans les coefficients C_i. On peut réécrire le système précédent sous forme matricielle:

$$A.U = C \tag{6}$$

avec $A(i,j) = a_{ij}$; $C(i) = c_i$; $U(i) = u_i$.

Pour résoudre ce système, nous utilisons un algorithme itératif, la méthode de résolution de Jacobi, car celle-ci aboutit à un algorithme parallélisable. Il s'écrit:

$$\begin{array}{l} k = 0 \\ U^0 = 0 \\ Tantque(Residu > Epsilon) \\ \quad k = k + 1 \\ \quad U_i^k = \dfrac{1}{a_{ii}}[-\sum_{i \neq j} a_{ij}u_j^k + C_i] \\ \quad Residu = \|U^k - U^{k-1}\| \\ Fin \end{array} \tag{7}$$

Où "Epsilon" est un seuil de tolérance fixé.

Les algorithmes écrits dans les différents langages testés ici sont donnés en annexe.

3.2.3 Comparaison des différents langages.

Les conditions expérimentales sont les mêmes qu'au chapitre précedent. Les calculs ont été effectués avec 8K ou 16K processeurs.
Les exécutions ont été réalisées sur des grille 128*128 et 256*256 points

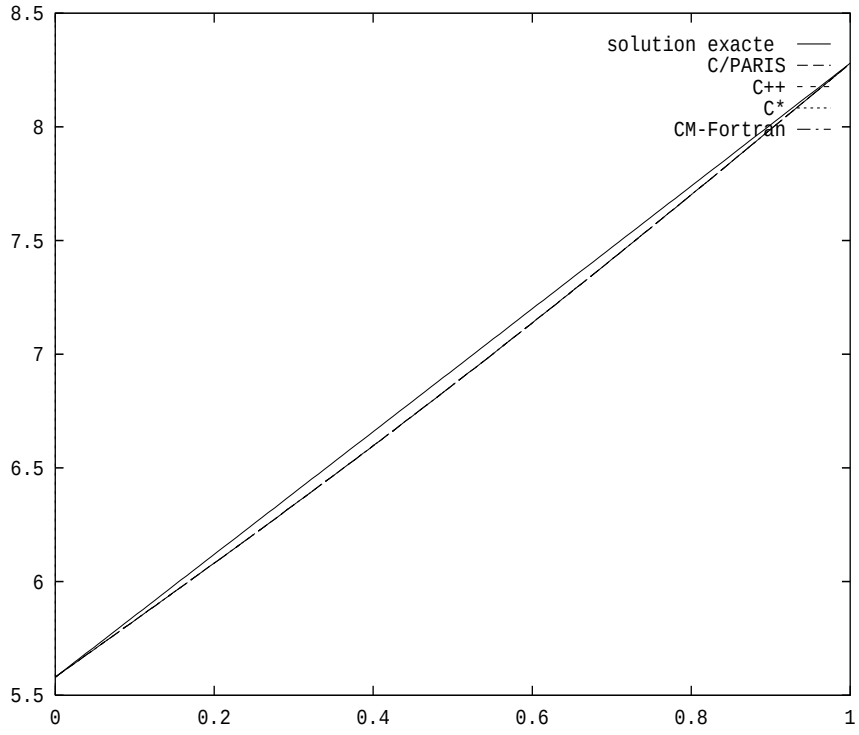

Figure 1: comparaison des différents langages - grille $128 * 128$ points

La figure 4 représente la solution exacte et les solutions approchées en $y = 0.5$. Les solutions approchées correspondant à l'écriture de l'algorithme dans les différents langages sont (heureusement!) identiques.

les temps d'exécution montrent cependant des différences de performance entre les langages:

Machine/grille	VPR	C/PARIS	C++	C*	CMF
8K - 128*128	2	117	81	109	112
8K - 256*256	8	170	106	163	156
16K - 128*128	1	180	130	169	144
16K - 256*256	4	287	190	262	280

Tableau des performances (Mflops)

Machine/grille	VPR	C/PARIS	C++	C*	CMF
8K - 128*128	2	33	47	35	34
8K - 256*256	8	268	428	279	299
16K - 128*128	1	21	29	23	27
16K - 256*256	4	158	240	174	162

Temps d'exécution (secondes)

Machine/grille	VPR	C/PARIS	C++	C*	CMF
8K - 128*128	2	16865	16865	16867	16865
8K - 256*256	8	49572	49569	49580	49574
16K - 128*128	1	16865	16685	16867	16865
16K - 256*256	4	49572	49569	49580	49574

Nombre d'itérations nécessaire pour un résidu de 10^{-6}

On peut vérifier que le passage de 8K à 16K (pour une même grille) multiplie les performances par un facteur qui tend vers 2 avec le VPR. On remarquera que les temps CPU correspondants sont eux divisés par ce facteur. Ce bon comportement est dû pour une grande part au caractère structuré des communications (usage de NEWS).
Outre les remarques concernant les performances obtenues avec les différents langages, cette application simple permet des comparaisons sur la programmation même (écriture, facilité d'utilisation, etc..):
Les codes des programmes PARIS, C++, C* et CM-Fortran sont donnés en annexe.

- C-PARIS est le langage de programmation de base de la CM, les fonctions sont très nombreuses mais aussi très précises, il s'apparente ainsi à un langage de type assembleur et les codes ne sont donc pas très lisibles, exemple d'instruction:
  CM_f_mult_add_const_3_1L(x, y, a, mantisse, exposant).

Cependant, on peut appeler ce type d'instructions à partir des langages de haut niveau (C, FORTRAN par exemple) pour accélérer des parties de calcul intensif.

- Le CM-Fortran basé sur le Fortran 77 est plus évolué mais peut paraître moins rigoureux et surtout moins convivial que le C par exemple: numérotation des colonnes, formats d'affichage ... L'avantage du CM-Fortran est qu'il est basé sur la notion de tableau; on peut donc manipuler un processeur comme élément d'un tableau, ce qui est très utile lorsqu'on utilise des grilles. Cependant, les dimensions des grilles doivent être explicitement déclarées dans le programme (comme en FORTRAN 77), mais ceci est plus contraignant en mode parallèle qu'en mode séquentiel.

- On retrouve cette restriction sur les déclarations dans le langage C* qui semble cependant donner les meilleures performances (après PARIS).

- La bibliothèque C++ permet de ne définir la grille que lors de l'exécution du programme et propose d'autres avantages comme le contrôle des paramètres, les facilités d'affichage du C++ ainsi que les avantages liés à la programmation objet surtout du point de vue conception de bibliothèque (on peut définir ses propres classes, améliorer celles qui existent ...). Cependant, des améliorations sont à faire en particulier pour les performances de calcul. Par exemple, dans la résolution du Laplacien, les communications entre processeurs sont nombreuses et on effectue des opérations sur les valeurs récupérées; C/PARIS permet en une seule instruction d'aller chercher la valeur sur le processeur voisin et d'effectuer le calcul, alors que C++ fait l'opération en plusieurs étapes. On peut aussi envisager, pour une optimisation plus poussée la construction d'un pré-compilateur et d'un analyseur syntaxique qui recherche, en fonction des opérateurs et des variables utilisés, la "meilleure" instruction PARIS à appeler.

3.3 Wator II - simulation écologique.

3.3.1 Objectifs.

WatorII est une simulation écologique adaptée d'un programme appelé Wator ([8]).
Il s'agit d'une simulation de type "proie-prédateur"; ici entre des requins et des épinoches (petits poissons). Une telle simulation nécessite le parallélisme pour suivre le comportement simultané de plusieurs milliers (voire plus) d'individus, ce qui est difficilement envisageable en mode séquentiel ou alors très coûteux en temps et en mémoire bien que les opérations par individu soient en général peu

nombreuses. D'autre part, l'exécution sur la CM permet de visualiser en "temps réel" l'évolution de la population et ce grâce aux outils graphiques signalés plus haut.
L'océan est représenté par une grille 2D où chaque point de la grille figure une parcelle de l'océan qui sera vide ou occupée par un seul poisson.
La principale difficulté dans ce type de simulation est de définir des règles de déplacement des populations.
Les règles ont été ici volontairement simplifiées. Ainsi les poissons se déplaceront suivant les quatre directions cardinales et sur une distance d'un processeur à la fois. Cette règle de déplacement va permettre d'utiliser les communications de type NEWS.
Chaque point de la grille est affecté à un processeur qui peut avoir ainsi trois états possibles:

- Vide (eau)
- Occupé par une épinoche
- Occupé par un requin

A chaque étape, tous les poissons sont mis à jour. Pour chaque type de poisson, il y a un nombre de paramètres qui déterminent le comportement:

- Un poisson donne naissance à un autre en fonction de son âge et de la dernière naissance.
- Les épinoches et les requins ont des périodes de reproduction différentes.
- Les requins meurent de faim s'ils n'ont pas mangé d'épinoches depuis un certain temps (temps de survie).
- Les épinoches se nourrissant de plancton ("non pollué") et ne meurent donc pas de faim.

On définit les variables parallèles suivantes:

- unspar état { 0 (Eau), 1 (Epinoche), 2 (Requin)}
- unspar breed_time (période de reproduction Epinoche ou Requin)
- const starvation_time (temps de survie)
- unspar age

- unspar last_ate (dernier repas)
- unspar last_breed (dernière naissance)

La vie et la mort ont lieu en fonction de quelques règles simples:
Règles de mise à jour des épinoches:

age = age + 1;
Si l'épinoche peut bouger
    Si age >= age_to_breed
        une épinoche d'age 0 est laissée à la place
Sinon, elle ne peut pas donner naissance

Règles de mise à jour des requins:

age = age + 1;
last_ate = last_ate + 1;
Si last_ate > starvation_time
    le requin meurt
Sinon, il cherche une épinoche
    s'il y a une épinoche dans le voisinage
        il mange l'épinoche
        il prend sa place
    sinon il bouge vers une place vide
    si age >= age_to_bread
        un requin d'age 0 est laissé à la place

Implémentation:

Afin de suivre le comportement des populations de requins et d'épinoches suivant les conditions imposés, de nombreux paramètres sont modifiables à chaque exécution (pourcentages d'occupation dans l'océan des différentes espèces, périodes de reproduction et résistance à l'inanition pour les requins).

Du fait du déplacement en parallèle des individus, de nombreux conflits peuvent apparaître, par exemple deux individus essaient d'aller au même endroit au même moment ou encore un requin a vu une épinoche au même moment où celle-ci a vu un emplacement vide: qui va bouger en premier ? Pour résoudre ou seulement éviter ces conflits, nous ajoutons les règles suivantes:

- Les requins et épinoches ne sont pas mis à jour en même temps, les deux espèces évoluent donc en séquence.
- Les épinoches ne bougent pas de manière aléatoire, il y a un ordre de recherche. Par exemple, toutes vers l'est, puis celles qui n'ont pas bougé tentent de se déplacer vers l'ouest, puis le sud et enfin le nord. En procédant ainsi on évite le premier conflit.
- Les requins étant supposés moins nombreux, un algorithme de mouvement aléatoire a été implémenté.

L'ensemble des règles choisi enlève toute prétention à un caractère réaliste de l'évolution. Cependant et en dépit de toutes ces simplifications (qui pourront être aisément supprimées pour certaines d'entre elles), l'évolution globale des populations n'est pas "farfelue" et reste en conformité avec les conventions choisies; ce que montre les résultats suivants.

3.3.2 Exploitation des résultats.

On peut exploiter les résultats obtenus de diverses façons, nous avons choisi de suivre les dépendances entre la quantité de requins et celle d'épinoches ainsi que la façon dont le système s'équilibre.
Les courbes suivantes montrent l'évolution des espèces à la recherche d'un équilibre:

L'axe des x représente le nombre d'itération (une itération correspondant à une période de deux jours environ). En ordonnée on trouve les niveaux de population ramenées à une même échelle à cause de la disparité de niveau entre les deux espèces. Les pics sur la courbe des requins correspondent aux périodes de reproduction.

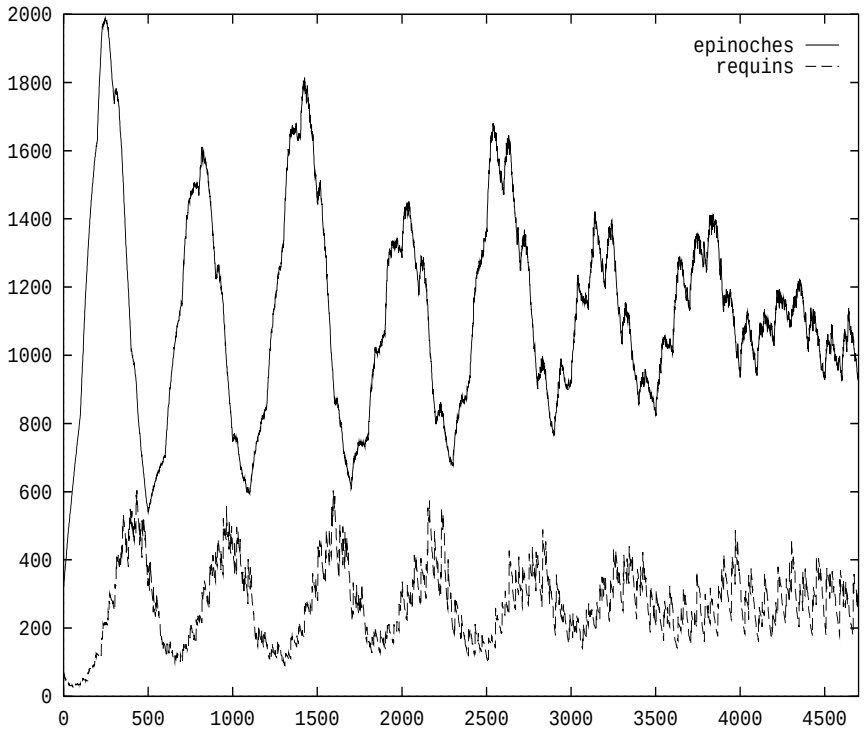

Figure 2: Evolution de la population

Voici les paramètres de la première courbe:

- population initiale épinoches : 50/1000
- population initiale requins : 1/1000
- espace vide : 949/1000
- âge de reproduction des épinoches : 100
- âge de reproduction des requins : 80
- résistance à l'inanition des requins : 20

Les résultat de la figure ci-dessus sont bien conformes à une évolution de population de type proie - prédateur. Comme on peut le constater, les pics des prédateurs suivent de très près les minima des proies, et celà malgré les simplifications du problème.

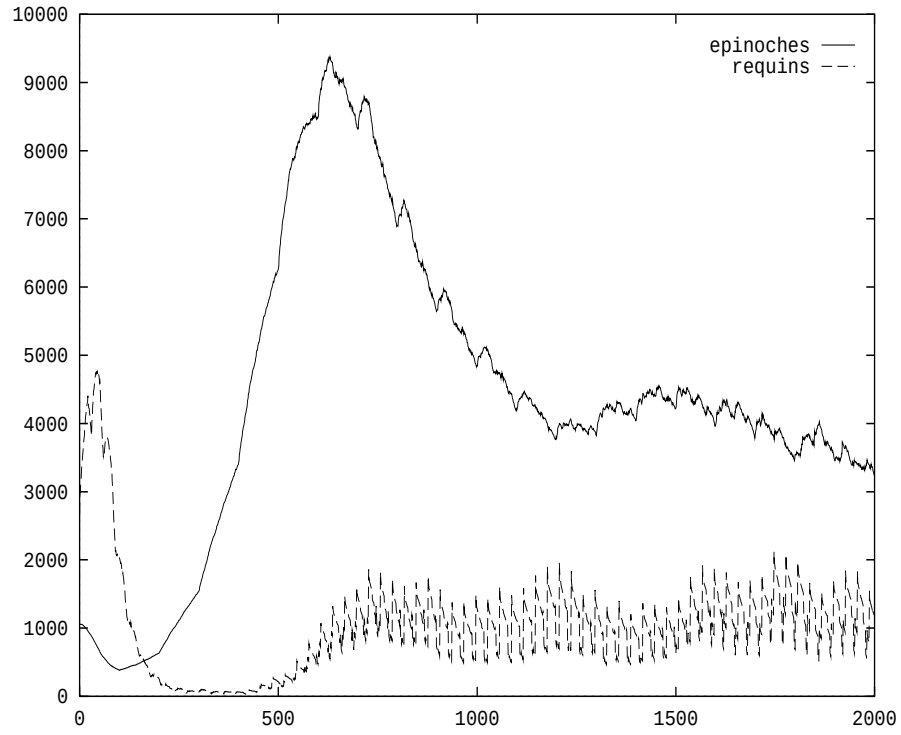

Figure 3: Evolution de la population

Paramètres de la seconde courbe:

- population initiale épinoches : 50/1000
- population initiale requins : 11/1000
- espace vide : 939/1000
- âge de reproduction des épinoches : 100
- âge de reproduction des requins : 30
- résistance à l'inanition des requins : 20

Ces deux courbes nous montrent la façon dont le système tente de s'équilibrer. Pour la première courbe, le système met plus longtemps à s'équilibrer, chaque espèce étant assez forte pour tenter de l'emporter. La seconde courbe montre que malgré de meilleures conditions au départ (plus nombreux et se reproduisant plus vite que pour la première courbe), les requins ont failli être exterminés,

ceci parce qu'ils étaient loin de la position d'équilibre. Tout se passe de la façon suivante: plus les requins sont nombreux et plus ils mangent et donc moins il y a d'épinoches. Quand il n'y a plus assez d'épinoches les requins commencent à mourir et les épinoches peuvent se multiplier de nouveau permettant ainsi aux requins de manger , etc...
Outre l'observation d'un système proie-prédateur, l'écriture de ce programme a permis d'utiliser les possibilités graphiques de la CM-2 et d'améliorer la bibliothèque, en particulier pour la manipulation des contextes.
Ci dessous, un exemple de visualisation de WatorII, normalement en couleur sur écran vidéo ou sun couleur et ramené ici en niveaux de gris, (d'où la mauvaise qualité de l'image reproduite). Sur la figure, les requins apparaissent en noir (couleur réelle rouge) et les épinoches en blanc (couleur réelle verte).

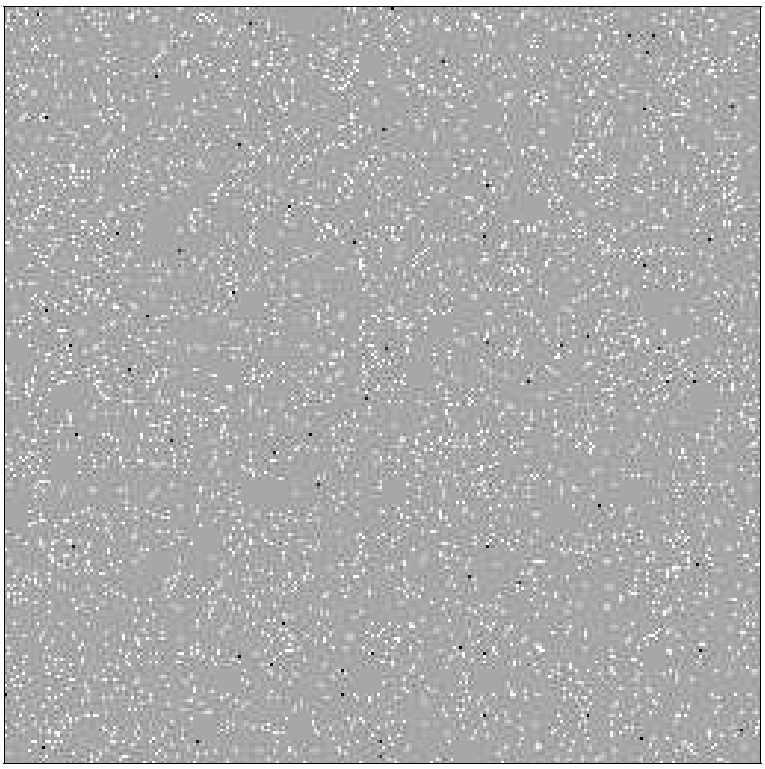

Figure 4: WatorII - exécution sur 256*256 processeurs

Conclusion

L'objectif visé dans ce travail était double: comparer les langages de haut niveau disponibles sur la CM et ce tant au point de vue programmation que performance des codes résultants d'une part, mettre au point une bibliothèque utilisant les potentialités d'un langage orienté objet (C++ ici) tout en respectant les particularités de la machine utilisé d'autre part.
Nous n'avons pas cherché ici à répondre à la question que l'on se pose (ou que l'on peut se poser):
" Faut-il programmer encore en FORTRAN en calcul scientifique ?"
Ou encore:
"L'avènement du parallélisme ne va-t-il pas définitivement discréditer le FORTRAN au profit de langages plus *ouverts* et plus *conviviaux*? " Signalons qu'au moment où nous écrivons, le FORTRAN 90 (plus orienté vers le calcul vectoriel et parallèle tout en restant compatible avec le FORTRAN 77) n'est pas encore sur le marché.
Nous pensons cependant que le calcul parallèle ne se banalisera dans les milieux de recherche et de l'industrie que lorsque les outils logiciels pour l'environnement et la programmation seront maitrîsés voire lorsque des standards s'imposeront au-delà des différences dans les réseaux de communication existants ou à venir. Ainsi et pour ce qui concerne les langages (C* et CM-FORTRAN) et la machine (CM2), testés ici, les résultats de performance obtenus sur des opérations élémentaires sont comparables. Pour des codes plus complexes, d'autres facteurs (écriture, niveaux d'optimisation des compilateurs,...) interviennent dans les résultats de performance, parfois sensiblement différents d'un langage à un autre. Si l'on ajoute aux résultats de performance la facilité de programmation, le C* semble l'emporter quelque peu sur le CM-FORTRAN, avec cette réserve que le C* parait d'autant plus "facile" qu'on est familier avec le C classique.
Cela étant, tous les langages disponibles sur la CM2 sont adaptés à cette machine et donnent donc lieu à des codes non portables, ce qui n'est pas particulier à cette machine.
D'où l'idée d'un environnement de type bibliothèque qui contiendrait les adaptations nécessaires et suivrait l'évolution des machines considérées tout en modifiant au minimum les programmes écrits dans un langage plus "classique". Les premiers résultats obtenus avec la bibilothèque C++ peuvent ne pas paraître très probants du point de vue performance (par comparaison avec le nouveau C* par exemple) mais des améliorations et des extensions sont possibles pour viser un plus grand niveau d'optimisation tout en modifiant le moins possible les programmes de l'utilisateur (portabilité).

Bibliographie

- [1] Parallel Instruction Set - Thinking Machine Corporation.
- [2] Connection Machine Technical Summary - TMC.
- [3] Connection Machine Software Summary V6.0- TMC.
- [4] Programming in C/PARIS - TMC.
- [5] Programming in Fortran - TMC.
- [6] Programming in C* - TMC.
- [7] The C++ Programming Language - Bjarne STROUSTRUP.
- [8] Solving Problems on concurrent processors - Prentice Hall.

Annexes

- Programme source de Jacobi en :
  - CM-Fortran
  - C*
  - C++
  - C/PARIS
- Programme source en C++ de WatorII
  - Main.cc
  - WatorDomain.cc
  - WatorDef.h
  - WatorDomain.h

Table des matières