Hervé Miguel Cordeiro Paulino

Desenho e Implementação do pSystem para Arquitecturas de Memória Distribuída

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
1998

Hervé Miguel Cordeiro Paulino

Desenho e Implementação do pSystem para Arquitecturas de Memória Distribuída

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Mestre em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
1998

Conteúdo

Gostaria de dedicar este trabalho aos meus pais, que ao longo dos anos me apoiaram incessantemente nos meus estudos e cujo sacrifício pretendo homenagear.

Agradecimentos

Antes de mais gostaria de agradecer todo o apoio e orientação científica e pedagógica que me foi facultada pelo meu orientador Dr. Fernando Silva. Gostaria de agradecer a sua constante disponibilidade, preocupação e interesse, assim como a significante contribuição dos seus conhecimentos no desenvolvimento deste trabalho. Gostaria também de agradecer a Luís Lopes pelas suas relevantes sugestões, que me ajudaram bastante, tanto no desenvolvimento do projecto, como na escrita deste texto. A Eduardo Correia pela manuntenção e optimização possível das máquinas utilizadas como plataforma de teste. A José Carlos Miranda pela facultação do programa que permite estudar o custo de transmissões de mensagens entre dois agentes. A Ricardo Correia e a José Maria Fernandes pela amizade, troca de ideias, de conhecimentos, de experiências e de frustrações e conquistas. E por fim gostaria de agradecer o incessante apoio da minha família e da minha namorada Cláudia Moura, que também contribuiu com a sua imensa paciência.

Resumo

Recentemente, têm sido propostas e construídas máquinas paralelas baseadas em componentes standard. A arquitectura destas máquinas é predominantemente de memória distribuída, composta por *clusters* de multiprocessadores interligados por redes muito rápidas, como a Myrinet ou a ATM. Todavia, o sucesso deste tipo de arquitecturas está condicionado aos ambientes de programação que proporcionam. Ambientes de programação paralela, como o PVM ou o MPI, são de grande utilidade para a programação de arquitecturas de memória distribuída, no entanto, requerem do programador um conhecimento detalhado da arquitectura alvo e o controlo explícito de toda a comunicação. Uma alternativa consiste na utilização de uma camada de *software*, por exemplo o TreadMarks, que forneça memória virtualmente partilhada, permitindo um modelo de programação mais próximo do de memória partilhada.

Esta tese apresenta o desenho e a implementação do **di_pSystem**, um ambiente de programação para arquitecturas de memória distribuída. Este consiste numa extensão ao pSystem, um ambiente de programação para arquitecturas de memória partilhada que permite a anotação de programas em C para execução paralela. O di_pSystem deverá preservar, ao máximo, a facilidade de programação do pSystem, assim como a sua modularidade em termos de arquitectura. Deste modo, distribuirá e balanceará dinamicamente todo o trabalho paralelo gerado durante a execução paralela. A comunicação está escondida do programador e é gerida automaticamente, quando necessário, pelo ambiente. Os programadores têm apenas de identificar as funções da linguagem C que são candidatas a execução paralela; a comunicação para a distribuição de trabalho é gerida implicitamente pelo sistema.

O módulo de comunicação do di_pSystem utiliza o interface paralelo MPI para suportar a computação distribuída. Este módulo é o único que depende do sistema utilizado. No entanto, a sua adaptação a outros sistemas é simples. O di_pSystem também pode ser utilizado para estudar a performance de algoritmos de distribuição de trabalho. No âmbito deste trabalho implementaram-se quatro heurísticas assim como um pequeno número de aplicações que serviram para uma avaliação inicial à performance do sistema.

Abstract

Recently, new parallel machines based on commodity components have been proposed and built. These are usually distributed memory architectures composed of clusters of multiprocessors interconnected by very fast networks, such as Myrinet or ATM. The success of these architectures is however conditioned by the programming environments they offer to users. Parallel interfaces such as MPI or PVM are a great help for programming distributed architectures, but they still require from users a detailed knowledge of the underlying architecture as well as explicit control of communication. An alternative approach is to use a software layer, such as Treadmarks, to provide virtual shared memory hence allowing a programming model closer to that of shared memory.

This thesis presents the design and implementation of the di_pSystem, a programming environment for distributed memory architectures. It is an extension of the pSystem, a programming environment for shared memory that allows the annotation of C programs for parallel execution. Our view is that the new system should retain, as much as possible, the ease of programming of the pSystem as well as its modularity in terms of architecture. Similarly, the di_pSystem will dynamically schedule and load balance all parallel work that is made available during parallel execution. Communication is hidden from the programmer and is automatically handled by the environment whenever necessary. Programmers are only required to identify the C functions that are candidate for parallel execution; communication for work distribution is managed implicitly by the system.

The communication module within di_pSystem uses the MPI interface to support distributed computations. This is the only system dependent module, but it can easily be adjusted to support other systems. The system can also serve as test bed for studying the performance of scheduling algorithms and it already implementations for four dynamic algorithms. We have also implemented a small number of applications that have served as benchmarks for an initial study of the performance of the system.

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação e Objectivos

Na última década assistiu-se a um aumento muito significativo da velocidade dos processadores. Contudo, essa velocidade é ainda insuficiente para permitir, em tempo adequado, obter respostas para problemas de elevada complexidade, como sejam a previsão meteorológica global, a extracção de conhecimento de grandes conjuntos de dados (*data mining*) e a simulação da dinâmica de fluídos. Uma das soluções adoptadas no sentido de aumentar substancialmente a capacidade computacional, consiste na utilização de vários processadores cooperando na resolução desses problemas. No entanto, estas máquinas, denominadas máquinas de arquitectura paralela, são frequentemente muito dispendiosas, devido aos custos associados ao desenvolvimento e construção de raiz de processadores, redes de interligação, memórias e outros componentes. A estes custos podemos ainda acrescentar o tempo despendido na elaboração de *software*, e.g., sistemas operativos próprios, protocolos de comunicação entre os processadores e protocolos de coerência de informação.

O aumento gradual do fosso entre a velocidade dos processadores e das memórias tem contribuído significativamente para limitar a performance dos computadores com arquitecturas de von Neumann. Pelo que a abundância de processadores rápidos e baratos no mercado e o constante aumento da velocidade das redes de interligação tem estimulado a construção de máquinas paralelas de alto desempenho com um custo relativamente reduzido. Esta tendência, tem vindo a aumentar o interesse em ambientes de programação paralela que possibilitem, de uma forma simples mas eficaz, a programação neste tipo de arquitectura.

As arquitecturas paralelas de baixo custo baseadas em componentes *off-the-shelf* só se tornam atraentes se existirem ambientes de programação que possibilitem a sua programação de uma forma simples e eficiente. Para fazer face a esta questão, vários grupos de investigação têm reunido esforços para o desenvolvimento de ambientes que satisfaçam estes requisitos.

O pSystem [LS97] é um ambiente de programação para arquitecturas de memória partilhada inovador na sua concepção, permitindo a escolha e implementação de novas heurísticas de distribuição de trabalho. Não existindo nenhum ambiente para arquitecturas de memória distribuída, que tenha esta característica, a adaptação do pSystem para este tipo de arquitectura revela-se de grande interesse. O objectivo deste trabalho consiste nisto mesmo: desenhar e implementar uma adaptação do pSystem para arquitecturas de memória distribuída que tem como objectivos:

- apresentar um ambiente que permita a programação, em linguagem C, de arquitecturas de memória distribuída;
- preservar o mais possível o modelo de programação de memória partilhada, utilizado no pSystem, escondendo a camada de comunicação do programador;
- não implementar a camada de comunicação de raiz; recorrer a um outro sistema que proporcione essa funcionalidade como o MPI [MPI95] ou o PVM [GB94a];
- incorporar a distribuição do trabalho e o balanceamento da carga no sistema, para possibilitar ao programador concentrar-se apenas na construção da aplicação;
- desenhar o ambiente de uma forma modular isolando a distribuição de trabalho e a camada de comunicação, para permitir a experimentação de vários algoritmos de distribuição de trabalho e a adaptação da camada de comunicação a vários sistemas;
- realizar um estudo de performance do ambiente utilizando vários programas de teste e várias heurísticas de distribuição de trabalho.

1.2 Arquitecturas Paralelas de Baixo Custo

Existem programas que contêm partes no seu código que não requerem uma ordem específica de execução, podendo ser executados em paralelo. No modelo de von Newmann tradicional explorar esta característica de certos programas é difícil. Os processadores superescalares melhoram esta falha do modelo, mas, apesar de poderem executar várias instruções concorrentemente, estão limitados pelo número de instruções que podem lançar de cada vez e pela funcionalidade das unidades de processamento disponíveis. Acresce-se ainda o facto de um bloco de código só poderem executar concorrentemente quando é obvio que o resultado da computação a obter é idêntico ao que seria obtido através da computação sequencial do mesmo. A detecção automática de concorrência de tão baixa granulosidade é, contudo, bastante difícil, sendo o motivo de uma vasta área de investigação em compiladores.

A abundância de processadores relativamente baratos e rápidos é propícia ao desenvolvimento de máquinas de elevada performance agrupando pequenos multiprocessadores ou uniprocessadores ligados através de uma rede de alta velocidade, como são os casos das redes ATM [BC93], Myrinet [BCF+95], Gigabit [Par94] ou mesmo Fast Ethernet. Assim, com um custo relativamente baixo, pode-se obter uma arquitectura paralela, susceptível de permitir a exploração de paralelismo em aplicações [S95].

1.3 Ambientes de Programação Paralela

A utilização eficiente de arquitecturas paralelas requer a elaboração de ambientes de programação que permitam a construção, simples e eficaz, de aplicações. Este objectivo depende essencialmente da forma como as tarefas a executar em paralelo são detectadas e dos recursos gastos na sua execução. Como a construção de arquitecturas de baixo custo incide essencialmente em arquitecturas cuja memória está distribuída e o trabalho a realizar está especialmente vocacionado para este tipo de arquitecturas, esta secção visa focar as características gerais dos ambientes que utilizam o modelo de programação distribuído. Este modelo requer a troca de mensagens para toda a comunicação entre as entidades envolvidas na computação.

A natureza do modelo de programação de um ambiente de programação paralela para arquitecturas de memória distribuída depende de vários factores. Entre estes salientam-se o nível de intervenção do programador na anotação do paralelismo e da comunicação existente numa aplicação. Em ambos os casos a anotação pode ser implícita ou explícita, i.e., realizada, respectivamente, com ou sem a intervenção do programador.

Os ambientes que utilizam a detecção implícita das tarefas são bastante mais fáceis de programar, facto que se deve ao paradigma de programação utilizado, idêntico ao sequencial. No entanto, estes ambientes debatem-se com o problema da granulosidade das tarefas, que terá de ser suficientemente grande para compensar o custo da execução paralela da tarefa. Esta abordagem é essencialmente utilizada em linguagens declarativas, como por exemplo em implementações paralelas de sistemas de Prolog, YapOr [RSC97]. Em linguagens imperativas esta abordagem é complexa e limitada. Para tornear este problema outros sistemas recorrem ao programador, que indica explicitamente o que deve ser executado em paralelo, portanto a eficiência da execução da aplicação é transferida parcialmente para o programador. Para que este obtenha a maior quantidade de paralelismo possível e tarefas com uma granulosidade correcta, necessita de conhecer profundamente a aplicação que está a desenvolver e a arquitectura do sistema. Ambientes que utilizam a anotação explícita de paralelismo são, por exemplo, o CC++ [CCK93], o Jade [LR91] e o Charm [KRS].

Nos ambientes de programação paralela, a comunicação pode estar implícita no programa, ou seja, o utilizador não tem de saber o que está a ser enviado e para quem, como por exemplo no CC++ e no Charm, ou pode ter de ser referida explicitamente pelo utilizador, por exemplo PVM e MPI. Na programação com controle explícito da comunicação é necessário saber de antemão toda a comunicação existente. O envio de uma mensagem requer que se saiba do lado do emissor, quais os dados a enviar e quem é o receptor e do lado do receptor, quem enviou a mensagem e qual a variável onde esta deve ser colocada.

No modelo de programação distribuído toda a comunicação é realizada através de troca de mensagens, pelo que o volume de mensagens é um factor importante na eficiência de uma aplicação. Por este motivo torna-se necessário minimizar o tráfego de mensagens agrupando-as e fazendo *overlaps* com computação para maximizar a utilização dos recursos disponíveis. Outro problema importante diz respeito à granulosidade das tarefas, uma vez que a execução remota de uma tarefa envolve custos que têm de ser compensados pela sua granulosidade. Sendo assim, é necessário avaliar a granulosidade mínima de uma tarefa para que a sua execução em paralelo seja rentável.

Além dos problemas do custo do envio de uma mensagem e da granulosidade das tarefas, o modelo de programação distribuído debate-se com os problemas usuais dos modelos de programação paralela. Segue-se uma pequena descrição destes problemas:

Código sequencial excessivo. Por vezes as aplicações contêm zonas do código onde não é possível obter paralelismo. Pelo que o trabalho não pode ser distribuído pelos processadores, logo os recursos da arquitectura não estão a ser aproveitados eficientemente;

Carga não balanceada. Para que a execução seja o mais eficiente possível é necessário evitar ter-se processadores parados, sendo, portanto, imprescindível que o trabalho esteja bem distribuído entre eles, de preferência dinamicamente;

Sincronização. Os pontos de sincronização são necessários na programação paralela mas podem provocar atrasos na execução, pois alguns processadores terão de esperar pelo outros, desperdiçando tempo de computação.

1.4 O pSystem

O pSystem é um ambiente de programação paralela para arquitecturas de memória partilhada. Consiste numa extensão à linguagem de programação C que introduz um pequeno conjunto de anotações que permitem ao programador explicitar o paralelismo numa aplicação e gerir os acessos à memória partilhada. Com este conjunto de anotações o pSystem pretende proporcionar ao utilizador comum da linguagem C um ambiente simples onde poderá implementar, com um custo pequeno de aprendizagem, aplicações para memória partilhada.

A programação numa linguagem conhecida, evita ao programador o estudo de especifidades inerentes à sintaxe, semântica e compilação de uma nova linguagem. Além disso, a adaptação de programas já desenvolvidos em C para o pSystem é bastante fácil e intuitiva, o que se revela como outra qualidade deste ambiente. Para ilustrar esta característica segue-se o código de dois programas que computam a função de Fibonacci. Sendo o primeiro relativo à implementação sequencial e o segundo à implementação efectuada com o pSystem.

```
/******************************************************/
/* Fibonacci: implementação sequencial               */
/******************************************************/
#include <stdio.h>
long fib(int);

main(int argc, char **argv) {
```

```
  int n;
  long fibn;

  n = atoi(argv[1]);
  fibn = fib(n);
  printf ("Fibonacci(%ld) = %ld\n", n, fibn);
}

long fib(int n) {

  if (n < 2)
    return (1);
  return (fib(n-1)+fib(n-2));
}

/******************************************************/
/* Fibonacci: implementação paralela -- pSystem         */
/******************************************************/
#include  "pSystem.h"
#include  <stdio.h>
pSystem task <fib>( long, long? );

void  main( int argc, char *argv[] ){
  long  n, fibn;

  n = atol( argv[1] );
  start_pSystem( --argc, ++argv );
  new task  <fib>( n, ?fibn );
  end_pSystem();
  printf("Fibonacci(%ld) = %ld\n", n, fibn);
}

task  <fib>( long n, long ?fibn ){
  long pr[2];

  if( n < 2 )
    fibn = 1;
  else{
    --n; new task  <fib>( n, ?pr[0] );
    --n; new task  <fib>( n, ?pr[1] );
```

```
    Wait();
    fibn = pr[0] + pr[1];
  }
}
```

Como se pode verificar as alterações são pouco significativas e simples de introduzir.

No pSystem, a extracção do paralelismo é feita de modo explícito. O utilizador anota funções como tarefas a executar em paralelo, o que lhe permite ter perfeito controle sobre as componentes sequenciais e paralelas do programa. A granulosidade de cada tarefa depende da computação a realizar por cada função.

A distribuição de trabalho foi alvo de estudo no pSystem. O sistema foi desenvolvido de forma a permitir que a escolha de uma nova heurística seja simples e transparente. Para realizar esta operação basta substituir o ficheiro que inclui a heurística actual pelo que contém a desejada, compilá-lo e ligá-lo dinamicamente ao resto do sistema. Esta característica permite ainda o desenvolvimento de novas heurísticas, o que torna o pSystem ideal para a realização de estudos sobre distribuição dinâmica de trabalho.

1.5 O pSystem em Arquitecturas de Memória Distribuída

O trabalho a realizar, tem como objectivo adaptar o pSystem para arquitecturas de memória distribuída, preservando, no entanto, as suas características fundamentais. Sendo assim, a implementação a realizar será uma extensão à linguagem de programação C onde o paralelismo é anotado explicitamente. A implementação manterá ao máximo a interface paralela utilizada para memória partilhada, facilitando assim a adaptação, para execução em arquitecturas de memória distribuída, de programas já desenvolvidos no pSystem. O código seguinte ilustra as alterações necessárias a realizar na implementação paralela em pSystem da recorrência de Fibonacci, para que esta possa ser executada em arquitecturas de memória distribuída.

```
/*****************************************************/
/* Fibonacci: implementação paralela - di_pSystem      */
/*****************************************************/
#include  "di_pSystem.h"
#include  <stdio.h>
```

```
di_pSystem task <fib>( long, long? );

void  main( int argc, char *argv[] ){
  long  n, fibn;

  n = atol( argv[1] );
  Start_di_pSystem( --argc, ++argv );
  new task  <fib>( n, ?fibn );
  End_di_pSystem();
  printf("Fibonacci(%ld) = %ld\n", n, fibn);
}

task  <fib>( long n, long ?fibn ){
  long pr[2];

  if( n < 2 )
    fibn = 1;
  else{
    --n; new task  <fib>( n, ?pr[0] );
    --n; new task  <fib>( n, ?pr[1] );
    Wait();
    fibn = pr[0] + pr[1];
  }
}
```

Como se pode verificar as alterações incidem apenas no nome de algumas funções. De resto o algortimo utilizado é idêntico.

Esta implementação segue um modelo de programação distribuído simétrico, **SPMD** **S***ingle* **P***rogram* **M***ultiple* **D***ata*), i.e., o mesmo programa é executado em todos os processadores mas sobre um conjunto de dados diferente. Apesar de esta implementação estar essencialmente vocacionada para arquitecturas de memória distribuída, pode ser utilizada em arquitecturas, tanto de memória partilhada, como de memória virtualmente partilhada. O estudo realizado por LeBlanc e por Markatos [LM92], revela que o modelo de programação distribuído, com base na troca de mensagens, comporta-se melhor que o modelo de memória partilhada em arquitecturas de memória partilhada recentes. Sendo assim, esta implementação está especialmente vocacionada para as máquinas recentes de memória partilhada e de memória distribuída virtualmente partilhada.

Para que o programador se possa concentrar exclusivamente na elaboração da aplicação, sem ter de se preocupar com detalhes de comunicação e de distribuição de trabalho, a comunicação entre as entidades de processamento estará escondida do programador e a distribuição de trabalho estará incorporada no sistema. As funções que definirão a interface de comunicação terão como suporte para a troca de mensagens o protocolo standard de comunicação em multicomputadores, MPI. O módulo de comunicação surgirá como um módulo independente do resto do sistema, o que possibilitará a sua implementação noutros sistemas de comunicação, ou mesmo o seu desenvolvimento de raiz, sem que tal afecte o resto do sistema.

O balanceamento da carga de trabalho é muito importante na programação paralela, portanto será alvo de estudo na elaboração deste sistema. A abordagem a seguir visa retirar toda a responsabilidade directa ao programador, pois o sistema fará dinamicamente a gestão do trabalho que terá para executar. O utilizador deverá apenas escolher, de um leque de heurísticas de distribuição de trabalho disponibilizadas pelo sistema ou elaboradas pelo próprio, a que mais lhe convém. Posteriormente, na fase de lançamento da aplicação, a heurística poderá ser refinada com a introdução de parâmetros de trabalho mínimo e máximo que as entidades processadoras deverão respeitar.

Em suma, esta implementação do pSystem é um extensão à linguagem de programação C, onde o paralelismo é anotado explicitamente pelo programador. Pode ser utilizado em máquinas com arquitecturas de memória partilhada ou distribuída, assim como em redes de computadores. O modelo de programação que implementa é o SPMD, onde a camada de comunicação está escondida do programador. Sendo assim, programar neste sistema é semelhante a programar em C, bastando anotar as tarefas a executar em paralelo, pois o sistema encarrega-se de lançar a aplicação, assegurar a comunicação e balancear a carga de trabalho.

1.6 Estrutura da Tese

Este texto está estruturado da seguinte forma:

O **Capítulo 2** faz um resumo das arquitecturas paralelas utilizadas actualmente, incidindo mais nas que poderão ser utilizadas como plataformas alvo do trabalho a realizar. Descreve ainda alguns ambientes de programação paralela, para este tipo de arquitecturas, que têm objectivos semelhantes aos propostos neste trabalho.

No **Capítulo 3** descreve-se o ambiente de programação paralela para arquitecturas de memória partilhada, pSystem. Familiarizando o leitor com a sua interface paralela e o seu motor de execução. Depois descreve-se a potencialidade do pSystem como ferramenta para avaliar heurística de distribuição de trabalho.

O **Capítulo 4** introduz o trabalho realizado, nomeadamente o ambiente de programação para arquitecturas de memória distribuída, di_pSystem. São descritas as suas características gerais, como a interface paralela, o módulo de comunicação, o novo conceito de variável de leitura global e os vários módulos que compõem um agente. Cada módulo e interacção entre módulos é descrita detalhadamente.

No **Capítulo 5** descrevem-se as soluções apresentadas para implementar o ambiente de programação di_pSystem. Primeiramente incide-se sobre o modelo geral de execução, partindo depois para a descrição dos algoritmos desenvolvidos para implementar mecanismos pontuais. Por fim descrevem-se as funções que compõem o módulo de comunicação e como o MPI foi utilizado para as implementar.

No **Capítulo 6** apresenta-se um estudo de performance do di_pSystem. São descritas as características das aplicações implementadas para o efeito, assim como algumas heurísticas de distribuição de trabalho. São apresentados estudos de ganhos de velocidade, de comparação com o MPI, e do impacto da granulosidade e da velocidade das redes na performance.

Por fim no **Capítulo 7** apresentam-se as conclusões que se podem retirar deste trabalho, assim como o trabalho que se poderá realizar brevemente.

Capítulo 2

Arquitecturas Paralelas e Ambientes de Programação

As arquitecturas paralelas e os ambientes de programação paralela, estão, obviamente associados. Neste capítulo pretende-se, de forma resumida, descrever quais são as arquitecturas paralelas actualmente disponíveis no mercado e nos centros de investigação, com especial ênfase nas arquitecturas suportadas pela implementação do pSystem para memórias distribuídas, isto é, máquinas com arquitectura **MIMD** (**M**ultiple **I**nstruction **M**ultiple **D**ata) e redes heterogéneas de computadores. Pretende-se ainda descrever alguns ambientes de programação paralela, incidindo especialmente nas diferentes soluções utilizadas para os vários problemas que a programação paralela apresenta.

2.1 Arquitecturas Paralelas

Desde os anos sessenta, têm vindo a desenvolver-se computadores com arquitecturas denominadas paralelas (mais do que uma unidade de processamento). Em 1972, Flynn [Fly72] propôs uma classificação para os vários tipos de arquitecturas, baseada no fluxo de instruções e de dados, conhecida como a taxonomia de Flynn. Esta classificação continua a ser a mais seguida, apesar de terem sido apresentadas outras classificações, por exemplo por Feng e Händler [HB85]. A taxonomia de Flynn classifica as arquitecturas de computadores em quatro grandes categorias:

SISD (**S**ingle **I**nstruction **S**ingle **D**ata)

Inclui os computadores usuais com um processador (máquinas de von Neumann) onde apenas uma instrução é executada de cada vez [1] e sobre um único conjunto de dados.

SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata)

Esta categoria está intimamente ligada com o paralelismo de dados. Estas máquinas executam a mesma instrução em vários processadores sobre diferentes conjuntos de dados. Os processadores vectoriais são um bom exemplo deste tipo de arquitectura, pois contêm uma unidade de controle central que controla todos os processadores, que por sua vez contêm uma unidade de execução. Como apenas uma instrução é executada de cada vez, existe apenas um fluxo da memória para a unidade de controle. Esta, difunde a instrução para as unidades de execução, que com os dados respectivos, a executam. Uma instrução só termina quando todas as unidades de execução tiverem terminado a sua execução.

MISD (**M**ultiple **I**nstruction **S**ingle **D**ata)

Esta categoria é puramente teórica, pois ainda não foi construído nenhum computador que execute várias instruções sobre um único conjunto de dados. Além disso, não parece existir interesse prático, tanto na comunidade cientifica como na comercial, para que tal venha a acontecer num futuro próximo.

MIMD (**M**ultiple **I**nstruction **M**ultiple **D**ata)

Os computadores incluídos nesta categoria são constituídos por vários processadores que operam independentemente em conjuntos de dados diversos.

2.1.1 Arquitecturas MIMD

Com o constante desenvolvimento das arquitecturas paralelas, a taxonomia de Flynn deixa de ser suficiente para caracterizar as arquitecturas que incluem a categoria MIMD. Isto deve-se ao facto de estas arquitecturas não se distinguirem pelo seu fluxo de controle e de dados, mas sim pela estrutura da sua memória (*hardware*). Sendo assim, as arquitecturas MIMD dividem-se em três categorias:

[1] aqui abstraímo-nos do facto de arquitecturas superescalares/*pipelined* poderem executar várias instruções simultaneamente em vários itens de informação

- memória partilhada
- memória distribuída
- memória distribuída virtualmente partilhada.

Nas arquitecturas de memória partilhada, cada processador pode ler e escrever qualquer elemento no espaço de memória global. Os processadores estão pendurados num ou em vários *bus*, tendo cada um deles, normalmente, uma memória *cache* suficientemente grande para reduzir o trafego no *bus* e o tempo de acesso aos dados. Para garantir a coerência da *cache* e melhorar o seu desempenho usam-se os algoritmos de *snopping*: **write invalidate** e **write update**. Com *write invalidate*, sempre que um elemento é alterado numa das *caches*, as cópias existentes nas outras são invalidadas, sendo o valor correcto enviado posteriormente quando for requisitado. Com *write update* (ou *write through*), todas as cópias são actualizadas mal o valor seja alterado. Esta segunda leva a que haja mais tráfego no *bus*, mas reduz as faltas na *cache*, ao contrário, o *write invalidate* reduz o transito no *bus* e os acertos na *cache*. Exemplos deste tipo de máquinas são a IBM 9000, a Cray C94 e a SPARCCenter.

As arquitecturas de memória partilhada oferecerem um modelo de programação relativamente simples, pois está mais próximo do modelo sequencial. No entanto, não são arquitecturas escalonáveis (no máximo 100 processadores, normalmente cerca de 20) devido à contenção no *bus* de comunicação com a memória. Este factor é limitativo quando se pretende uma máquina com um grande número de processadores. Acresce ainda que, devido ao facto da tecnologia de desenvolvimento de memórias não acompanhar, nem de perto, os progressos atingidos nos últimos anos no desenvolvimento de processadores, o custo de comunicação (acesso à memória) neste tipo de arquitectura tem vindo a ser cada vez mais um factor limitativo de performance [ML92].

Numa arquitectura de memória distribuída cada um dos processadores tem uma memória local e encontram-se ligados através de uma rede interna muito rápida. Os dados de cada processador são privados, ou seja, um processador não pode aceder directamente a dados que se encontrem na memória de um outro. Quando um processador pretende aceder a dados remotos, tem de enviar explicitamente uma mensagem e esperar pela resposta que conterá a informação requisitada. Exemplos de máquinas com esta arquitectura são a Intel XP/S-MP, a Cray T3E-1200 e a IBM SP2.

As arquitecturas de memória distribuída oferecem um modelo de programação mais complexo do que as de memória partilhada, visto que não existindo memória partilhada toda a comunicação entre processadores é explícita e tem de ser efectuada através

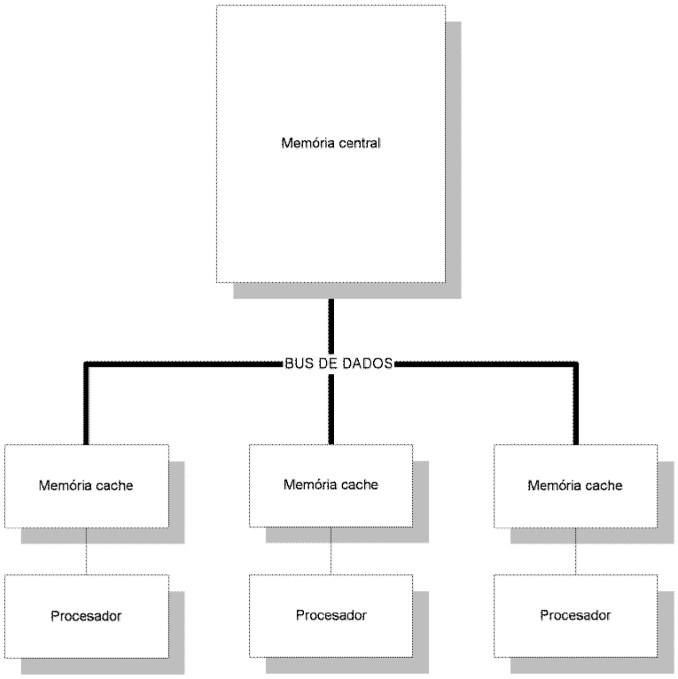


Figura 2.1: Arquitectura de uma máquina de memória partilhada.

de mensagens. No entanto, este tipo de arquitecturas é muito escalonável, a Intel vendeu ao Sandia National Laboratories em Junho de 1997 um computador com 9200 processadores Pentium Pro. Usando o *benchmark* Linpack de medida de capacidade de processamento, esta máquina obteve o resultado recorde de 1,34 teraflops, excedendo em mais 25% a barreira dos teraflops quebrada em Dezembro de 1996. Estes resultados, indicam a rápida evolução que este tipo de arquitectura poderá ter nos próximos anos.

Um novo tipo de arquitectura que tenta conciliar o melhor das anteriores, é a arquitectura de memória distribuída mas virtualmente partilhada, também denominadas NUMA (**N**on **U**niform **M**emory **A**ccess). Estas, juntam a facilidade de programação das arquitecturas de memória partilhada com a capacidade de escalonamento das arquitecturas de memória distribuída. Exemplos de máquinas com este tipo de arquitectura são a SGI Power Chalange, a Cray Origin 2000, a FLASH e a Alewife. Na figura 2.2 temos o exemplo da arquitectura na FLASH que suporta os modelos de memória distribuída (troca de mensagens) e de memória partilhada. Este tipo de arquitectura também pode ser obtido, através de um suporte de memória virtual partilhada por software sobre uma máquina de memória distribuída, como é feito no TreadMarks [T94-6].

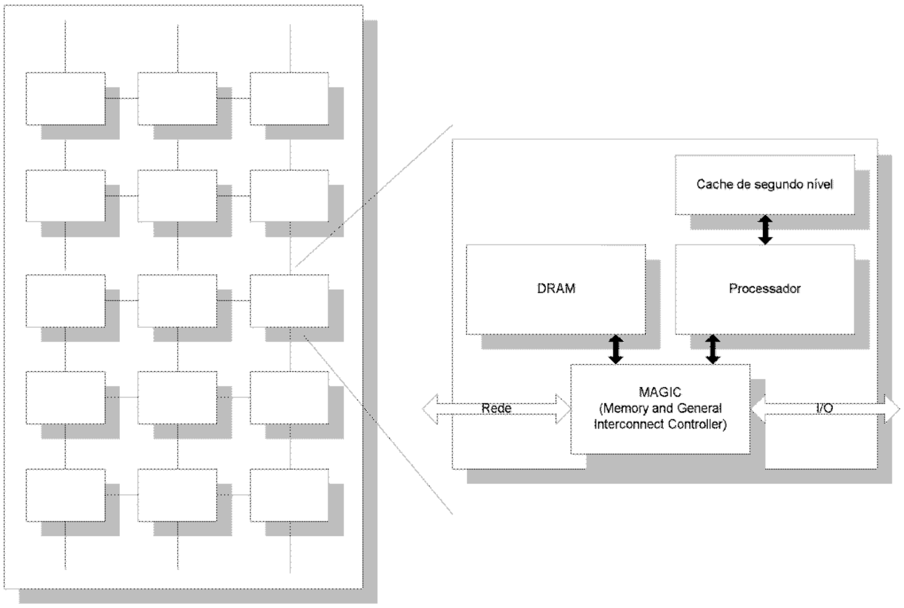


Figura 2.2: Arquitectura da FLASH (Flexible Arquitecture for SHared memory).

2.1.2 Arquitecturas Dataflow

Devido aos problemas inerentes ao transporte do modelo de von Neumann para as arquitecturas paralelas, como sejam a contenção no *bus* de acesso à memória em arquitecturas de memória partilhada e o custo adicional do envio de mensagens em arquitecturas de memória distribuída, têm-se desenvolvido esforços no sentido de encontrar alternativas a este modelo. Um exemplo são os modelos de execução **dataflow**.

No modelo *dataflow* a progressão da execução é determinada pelas dependências de dados entre os vários blocos do programa. Um bloco de um programa só é executado quando é necessário para o prosseguimento da execução e não pela ordem definida pelo utilizador. Sendo assim, um programa pode ser representado por um grafo acíclico cujos nós são os blocos de instruções e os arcos as dependências existentes entre estes. O bloco de instruções sediado num nó só pode iniciar a sua execução quando todos os dados de que necessita estão disponíveis. As máquinas *dataflow* existentes são, essencialmente, protótipos e como exemplo temos a máquina ADAM e a Monsoon. O modelo de programação agregado a esta arquitectura tem a vantagem de proporcionar a exploração fácil de paralelismo implícito.

2.1.3 Redes Heterogéneas

Mais recentemente, as redes de computadores ou máquinas construídas a partir de processadores e placas disponíveis no mercado, têm sido utilizadas como uma máquina paralela virtual de comportamento idêntico ao multicomputadores de arquitectura de memória distribuída. A execução de uma aplicação nesta máquina virtual depende apenas de um ambiente de programação que sustente este tipo de arquitectura, ou, mais simples ainda, um sistema que se encarregue da comunicação entre a máquinas, por exemplo o PVM ou o MPI. Este tipo de arquitectura tem como grande vantagem o baixo custo associado à sua aquisição.

Como podemos verificar na figura 2.3, as componentes desta máquina virtual podem ser de arquitecturas diferentes. O único defeito das redes de computadores usuais, será o facto do uso da máquina paralela virtual para correr aplicações poder ter um impacto negativo na performance de outras aplicações a correr nos nós de computação. Todavia, isto depende do tipo de trabalho a ser realizado pelos utilizadores das máquinas e, além disso, as aplicações paralelas poderão ser executadas quando as máquinas não estiverem a ser utilizadas.

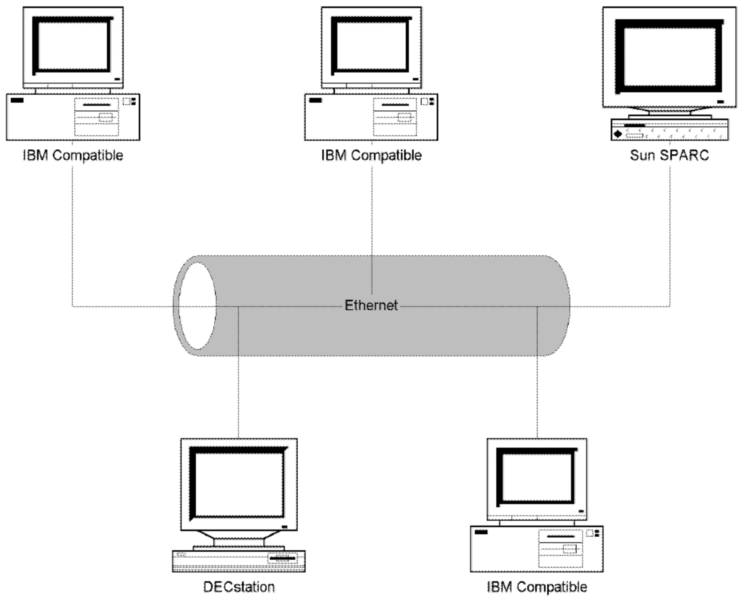


Figura 2.3: Rede heterogénea de computadores.

2.2 Ambientes de Programação Paralela

Existe actualmente um grande numero de ambientes de programação paralela [Che93], entre os quais extensões a linguagens já conhecidas como o C ou o Fortran, bibliotecas de funções e linguagens construídas de raiz. Alguns destes ambientes serão focados, de forma sumária, em especial aqueles que têm objectivos mais próximos com a implementação do pSystem distribuído.

2.2.1 Extensões a Linguagens de Programação

O **Concurrent C** [GR86, Gr90] é uma extensão à linguagem de programação C que permite o uso desta linguagem para desenvolver aplicações paralelas. Neste ambiente, os processos interagem através da troca síncrona de mensagens (*synchronous message passing*). Estas primitivas combinam a sincronização dos processos com a transferência de informação, ou seja, dois processos interagem sincronizando e depois transferindo informação. Este encontro denomina-se *rendez-vous* e pode ser simples se a troca de informação for unidireccional, ou extenso se for bidireccional. Num encontro bidireccional a informação é transferida depois do encontro se efectuar, tendo o emissor de esperar pela conclusão da transferência por parte do receptor para poder prosseguir a computação.

A escolha de um modelo síncrono relativamente a um assíncrono teve como razões o facto de grande parte das interacções entre processos ser síncrona e esta poder ser implementada de forma mais eficiente, pois não necessita de buffers de mensagens e de um controlador do tamanho do *buffer*, podendo ser a mensagem transferida directamente de um processo para o outro sem necessitar de buffers intermédios.

Uma definição de processo tem duas partes: um *tipo* e um *corpo*. Um processo é uma instanciação de uma definição de processo, tem o seu próprio fluxo de controle e executa em paralelo com os outros. Pode-se pensar em cada processo como tendo a sua própria pilha, registos, *program counter*, etc. O tipo do processo é a única parte pública de uma definição de processo. O seu corpo contém o código a executar, análogo ao corpo de uma função. As implementações deste ambiente devem ter um *scheduler* que distribui os processos pelos processadores, a sua heurística não é definida pelo *Concurrent C*, deixada como detalhe de implementação.

O **Concurrent C++** [GR88, Gr90] estende o ambiente anterior para permitir o uso da linguagem C++ no desenvolvimento de aplicações paralelas, mantendo, no entanto, o mesmo modelo de execução. O objectivo deste ambiente é combinar as potencialidades do paradigma de programação por objectos (abstracção de dados) do C++ com as possibilidade de programação paralela do Concurrent C. A noção de classe encaixa perfeitamente com a noção de definição de processo, sendo o tipo e o corpo correspondidos às partes públicas e privadas da classe. As noções de classe e herança permitem uma definição mais estruturada dos objectos concorrentes.

O **Split-C** [CDG]é uma extensão paralela da linguagem de programação C que permite o acesso a variáveis globais em multiprocessadores de memória distribuída. Existe uma implementação para a CM-5 a partir do `gcc` e de mensagens activas [ECG92], estando prevista a sua implementação para outras arquitecturas. O Split-C fornece um pequeno conjunto de primitivas para acesso global e declarações para paralelismo. Estas englobam as características mais salientes dos modelos de memória partilhada, troca de mensagens e paralelismo a nível de dados.

O seu modelo de controle é do tipo SPMD, onde todos os processadores iniciam a computação em simultâneo. Estes podem seguir vários rumos de fluxo de controle e juntarem-se em pontos de encontro como barreiras. Cada processador tem um identificador constante denominado *MYPROC*.

Qualquer processador pode aceder a qualquer endereço no espaço de endereçamento global, que está dividido fisicamente pelos vários processadores. A região local a cada um destes contém a pilha do processador para variáveis automáticas, variáveis estáticas e externas, e uma parte da *heap*. Existem dois tipos de apontadores globais, cuja diferença se reflecte no custo do acesso local ou global. Os apontadores globais referenciam todo o espaço de endereçamento global, os apontadores normais referenciam apenas a porção deste espaço sob a alçada do processador que faz o acesso.

O operador *split-phase* (:=) permite que a computação e a comunicação sejam sobrepostas. O pedido ou o envio de um valor é independente da conclusão da operação. O Split-C introduz ainda outro operador (:-) que tem como função sinalizar ao processador a modificação do valor de uma variável que ele possui. Qualquer dos operadores de atribuição pode ser utilizado para atribuir estruturas ou vectores. O Split-C fornece esta facilidade pois a sobreposição de comunicação com computação é mais vantajosa quando se transfere um grande volume de dados.

O **Charm** [KRS, RSS] foi desenvolvido na Universidade de Illinois e tem como objectivo ser um suporte para a programação de máquinas MIMD. Na óptica dos criadores do Charm não é necessário criar uma nova linguagem para expressar as subcomputações sequenciais de um programa paralelo. O ambiente utiliza uma linguagem já existente:

C.

A unidade de execução neste sistema é o **chare** que especifica os dados e a computação a serem mapeadas como uma unidade num único processador. No entanto, um processador pode ter zero, um ou mais *chares*. Cada *chare* pode receber várias mensagens provenientes de outros *chares*, tendo uma secção (função) de código C para lidar com cada mensagem. A função a ser invocada aquando da recepção de uma mensagem é definida pelo emissor que ao enviar a mensagem indica qual é o *entry-point* que esta deve activar. O *entry-point* do *chare* receptor encaminhará a mensagem para a sua função (*entry-point function*). Desenvolver um programa em Charm é definir um conjunto de *chares*, sendo o *main* um *chare* especial que inicia a computação.

O modelo de programação adoptado pelo Charm é a execução guiada por mensagens (*message driven execution*). Um *chare* só é chamado para executar quando existe uma mensagem para ele. Ao contrário dos processos, os chares não estão continuamente a executar, nem tem uma execução tipo *time-sliced*, estão, no entanto, permanentemente prontos para executar qualquer mensagem que lhes seja enviada. Este modelo minimiza o tempo de processamento morto dos processadores, pois desde que haja uma mensagem para um *chare* o processador encarrega-se da sua execução. A execução de um *entry-point* é atómica, ou seja, executa sem interrupções, e dois *entry-points* do mesmo *chare* não podem executar concorrentemente. Este modelo possibilita o *scheduling* adaptativo das computações entre e dentro dos módulos.

Existe também uma versão do Charm baseada no C++ intitulada **Charm++** [KK93].

O **Composicional C++** (CC++) [CCK93, CK] é uma extensão ao C++ que pretende ser uma linguagem de programação paralela de uso geral. Permite paralelismo ao nível de dados, de tarefa e de objectos e implementa os modelos de memória partilhada e de troca de mensagens.

O paralelismo é indicado explicitamente pelo utilizador através de três construtores para a composição paralela: o bloco `par`, a instrução `parfor` e a instrução `spawn`. O bloco `par` é o meio mais básico para especificar composição paralela. Estes blocos permitem o uso de qualquer instrução do CC++, com excepções na transferência de controle de fluxo. Não pode haver transferência de controle de fluxo:

- de uma instrução dentro bloco para fora do bloco,
- de fora do bloco para uma instrução dentro do bloco,
- entre instruções dentro do bloco.

As instruções no bloco podem ser sequenciais e os blocos `par` podem ser encapsulados. Um novo *thread* de controle é criado por cada instrução de primeiro nível no bloco `par`. Um bloco termina quando todas as instruções terminam. No entanto, não se requer que um bloco `par` termine. O bloco `par` é útil quando se quer criar um número fixo, conhecido durante a compilação, de *threads*. Apesar de se poder usar a recursão num bloco par para se criar um número arbitrário de threads, a iteração é normalmente um método mais natural e conveniente. Por esta razão o CC++ introduziu o construtor de ciclo paralelo `parfor` cuja sintaxe é idêntica à do `for` em C. O corpo da instrução pode ser uma instrução simples, um bloco sequencial ou um bloco paralelo. Novamente não pode haver transferências de controle de fora para dentro e de dentro para fora do ciclo.

O CC++ introduz a instrução `spawn` que executa uma expressão arbitrária num novo *thread* de controle. A sua execução é intercalada justamente com o resto do programa e, contrariamente ao `par` e ao `parfor`, o `spawn` termina imediatamente independentemente do *status* do processo lançado.

A sincronização entre os *threads* no CC++ é feita à custa de um novo modificador de tipos, o `sync`. Este indica que a variável é usada para sincronização e é tratada do mesmo modo que é tratada uma variável `const` no C++ havendo duas excepções:

- O valor inicial de um objecto `sync` não precisa de ser declarado aquando da sua criação. A inicialização pode ser atrasada, sendo realizada quando é atribuído um valor ao objecto. Depois de inicializado, o valor do objecto não pode ser alterado.
- Qualquer tentativa de leitura a um objecto `sync` não inicializado é atrasada até algum tempo finito depois da sua inicialização.

Para facilitar a construção de algoritmos que possam explorar localidade no CC++, é fornecido um mecanismo que agrupa objectos. Este construtor denomina-se **objecto processador lógico** (*logical processor object*). Além de fornecerem um meio para explorar localidade, os objectos processadores também formam o mecanismo básico para construir sistemas de *software* heterogéneos. O CC++ desvia-se um pouco do C++ no facto que todas as declarações globais estão contidas numa classe especial, **classe processadora lógica** (*logical processor class*). Funções que seriam globais no C++ são agora membros públicos da classe processadora lógica. Do mesmo modo declarações de classes e de tipos passam a ser declarações da mesma.

O objectivo do CC++ é ser um superconjunto completo do C++. Para tal suporta variáveis globais, variáveis de acesso a ficheiros e variáveis estáticas através dos objectos processadores. No CC++ também existem apontadores e referências para objectos. No entanto, é necessário distinguir as situações em que um apontador ou referência reside no mesmo processador lógico que o objecto a ser referido, ou num processador lógico diferente do objecto referido. No primeiro caso será um apontador ou referência local, o segundo caso um apontador ou referência global.

2.2.2 Bibliotecas de Funções

O **PVM** [GB94a, GB94b] foi desenvolvido na Universidade de Tennessee e no Oak Ridge National Laboratory e tem como objectivo permitir que um conjunto de computadores possa cooperar na execução de aplicações paralelas. Este conjunto, denominado **máquina paralela virtual**, pode incluir máquinas com arquitecturas uniprocessador, arquitecturas de memória partilhada ou de memória distribuída, que podem ser incluídas ou excluídas do conjunto durante a execução. O utilizador pode, ou não, explorar as características de cada máquina, direccionando tarefas especificas para as máquinas mais apropriadas. A unidade de execução do PVM é a **tarefa** (*task*), *thread* de controle sequencial que alterna comunicação com computação. No PVM várias tarefas podem executar num só processador.

O PVM usa o modelo de troca explícita de mensagens. Cada tarefa realiza parte da computação decomposta, comunicando através do envio ou recepção explícita de mensagens. O tamanho de uma mensagem é apenas limitado pela quantidade de memória disponível. Uma das grandes virtudes do PVM é suportar heterogeneidade ao nível das arquitecturas dos computadores e das redes de interligação. As mensagens podem conter mais do que um tipo de dados, mesmo quando são trocadas entre arquitecturas com diferentes representações de dados. O PVM tira ainda partido do mecanismo de troca de mensagens nativo dos multicomputadores, havendo empresas que implementem versões optimizadas do PVM para os seus computadores, podendo estas, no entanto, comunicar com a versão original.

O PVM fornece um conjunto de funções que permite agrupar tarefas. Uma tarefa pode entrar ou sair de um grupo sem ter de informar as outras ou fazer um *broadcast* (mensagem dirigida a todos os elementos de um grupo) para um grupo que não o seu. Os grupos não são necessariamente disjuntos, ou seja, uma tarefa pode pertencer a mais do que um grupo.

O PVM é composto por duas componentes, o *demon* `pvmd` que tem de ser lançado em todos os computadores que compõem a máquina virtual e a biblioteca de funções `libpvm` que inclui as funções acessíveis ao utilizador para que possa gerir a execução e máquina virtual. O PVM suporta actualmente as linguagens de programação C, C++ e Fortran. Existem várias variantes do PVM como o TPVM (*Threads-oriented* PVM) [FS94] e o LPVM (*Lightweight-process* PVM) [ZG95].

O **MPI** [MPI95] é um conjunto de primitivas de comunicação standard para sistemas que usem o modelo de troca de mensagens, tirando proveito das possibilidades mais atractivas desses sistemas. Tornar o MPI um standard foi o trabalho de cerca de sessenta pessoas e quarenta organizações, especialmente dos Estados Unidos e da Europa. Contribuíram para o seu desenvolvimento a maior parte dos construtores de máquinas paralelas e muitos investigadores de universidades, laboratórios públicos e industria. A grande vantagem do MPI é a sua portabilidade e a facilidade de utilização. A sua definição como standard permite aos construtores de máquinas paralelas optimizar as suas máquinas para executarem eficientemente as várias funções que fornece.

O MPI evita cópia de memória para memória, sobrepõe comunicação com computação e usa, quando disponíveis, co-processadores para comunicação. O utilizador não tem de ser preocupar com erros de comunicação, pois estes são tratados a um nível inferior da camada de comunicação. A sua plataforma de execução pode ser um multicomputador de memória distribuída, uma *workstation* ou uma rede heterogénea que combine estes dois tipos de máquinas. O MPI está vocacionado para programas MPMD ou SPMD, não tendo, à partida, nenhum suporte explicito para *threads*.

O MPI inclui funções que permitem:

- operações básicas de comunicação como o *send* e o *receive* e funções para melhorar a sua performance;
- operações sobre um grupo de processos como, por exemplo, o *broadcast* mencionado na descrição do PVM nesta secção e as *barreiras*;
- funções para criar e manipular *grupos*;
- objectos opacos com atributos que definem um contexto de comunicação para comunicação ponto a ponto, denominados *comunicators*.
- topologia de processos, que facilitam o mapeamento de grupos de processos para melhorar as estruturas topológicas, como as grelhas multidimensionais;

- suporte para Fortran77 e C;
- gestão do ambiente de execução, permitindo ao utilizador fazer perguntas sobre o ambiente de execução.

Existem actualmente várias implementações do MPI entre as quais se destacam o Lam [BDV], o Chimp [KDCZ94] e o MPICH [GL96]. Foram já realizados alguns estudos comparativos entre as algumas destas implementações [Nev96], assim como entre o MPI e outros sistemas semelhantes como o PVM [Sap].

O **TreadMarks** [T94-6, KDCZ94] permite a programação através do modelo de memória partilhada num computador multiprocessador de memória distribuída ou numa rede de computadores.

No TreadMarks, quando se efectua um acesso à memória partilhada, primeiro determina-se se os blocos de memória pedidos se encontram na memória local ou remota. No último caso, os dados são enviados para o processador que efectuou o pedido sem que para isso seja necessária a intervenção do utilizador. Quando um bloco de memória é actualizado o TreadMarks assegura que os outros processadores são notificados da alteração. Deste modo evita que estes utilizem valores errados, mantendo assim a consistência da memória. A notificação não é global, pois o custo desta operação é grande e pode não compensar, visto que a zona de memória modificada pode não ser de interesse para todos os processadores. A notificação também não é imediata, pois as atribuições são constantes num programa, o que geraria muito tráfego na rede. A abordagem escolhida passa por notificar os processadores nos pontos de sincronização, portanto, a notificação entre dois processadores é efectuada quando estes se encontram. Os programas que não contêm acessos a dados não determinísticos (*data races*) produzem o mesmo resultado independentemente da notificação ser imediata ou atrasada até à sincronização. Aglomerar várias notificações numa só mensagem a enviar no ponto de sincronização reduz consideravelmente o custo das comunicações entre processadores.

As *data races* levam a resultados não determinísticos, sendo portanto necessário sincronizar os processos no acesso à memória partilhada. A sincronização impõe um acesso exclusivo às zonas críticas do programa. Para obter a sincronização dos acessos à memória partilhada, o TreadMarks utiliza o algoritmo **Lazy Release Consistency** (LRC) que consiste em efectuar as operações de consistência quando se adquire o *lock* sobre uma variável e não quando se liberta o *lock*, como acontece no algoritmo usual de *Release Consistency* (ou *Eager Release Consistency*). Em vez de enviar uma mensagem para todos os processadores informado-os da modificação na variável, o LRC só actualiza o próximo processador que adquire o *lock*, encapsulando o valor da variável na mensagem de aquisição, o que reduz em muito o tráfego na rede. De facto apenas uma mensagem é enviada, em contraste com as mensagens para cada processador mais outra para a aquisição do *lock*.

O TreadMarks possibilita ainda vários acessos simultâneos de escrita a páginas (*multiple-writer protocols*) eliminando assim o problema de *false-sharing*. Para tal, quando uma cópia local de uma página é alterada o TreadMarks cria uma página gémea inalterada. Num ponto de sincronização é criado um *diff* que contém as diferenças entre as páginas, sendo então descartada a página gémea. Se o processador que se encontra no mesmo ponto de sincronização tiver uma cópia da mesma página, é-lhe enviado uma mensagem que indica que a página foi alterada. Ao receber a mensagem o processador invalida a página. Quando voltar a acede-la gera uma falta de página e pede ao processador original o *diff* correspondente à página para que a possa actualizar. Sendo assim, à excepção do primeiro acesso, nunca é necessário pedir uma nova cópia da página, pois esta é constantemente actualizada pelos *diffs*.

O **Dome** [ABL95] é uma biblioteca de classes do C++ que utiliza o PVM como suporte para a comunicação e controle entre os processos. Quando uma das suas classes é instanciada o objecto é automaticamente dividido e distribuído pelo ambiente. Assim computações que utilizem este objecto são realizadas em paralelo nos nós que compõem a máquina virtual paralela. O Dome utiliza o *overloading* de operadores para permitir o manuseamento simples dos seus objectos e para esconder pormenores de paralelismo.

Quando da criação de um objecto Dome, este é divido e distribuído pelos processos do programa distribuído. O Dome permite várias possibilidades para dividir os dados. A primitiva `whole` indica que todos os dados de um dado objecto são replicados em todos os processos, a distribuição `block` indica que os dados devem ser igualmente distribuídos e `dynamic` indica que a distribuição inicial é idêntica à do `block` mas pode ser redistribuída consoante o balanceamento da carga.

O modelo de execução utilizado pelo Dome é o SPMD. Cada nó executa em paralelo uma, ou mais, cópias do programa sobre um subconjunto dos dados de cada objecto Dome. A sua arquitectura orientada por objectos permite-lhe esconder a colocação dos dados e a comunicação do utilizador, possibilitando a sua alteração consoante as alterações no ambiente de execução. O Dome balança dinamicamente a carga do trabalho tendo em consideração as velocidade dos processadores e das redes de interconexão.

2.2.3 Linguagens de Programação

O **Mentat** [Gri93] é uma linguagem de programação por objectos baseada no C++ que explora a programação por objectos para esconder os aspectos relativos à programação paralela, incluindo comunicação, sincronização e distribuição de trabalho. O Mentat usa um modelo de computação guiado pelos dados (*data driven*) de granulosidade média e detecta e gere automaticamente as dependências de dados. Devido ao seu modelo de computação o Mentat é especialmente vocacionado para arquitecturas de memória distribuída (sem partilha virtual).

A unidade básica de execução no Mentat é uma função correspondente a um membro de uma classe Mentat. O programador é responsável pela identificação das classes cujos métodos são suficientemente complexos para que a sua execução em paralelo compense. As instâncias das classes Mentat são utilizadas como as classes do C++. As dependências de dados e de controle entre estas classes são detectadas e geridas automaticamente pelo compilador, através do tratamento de uma linguagem intermédia, e pelo sistema de execução, sem que seja necessária a intervenção do programador. O sistema de execução suporta mais do que a invocação de um método através de uma chamada de procedimento remota (*RPC*), suporta um modelo guiado pelos dados baseado num grafo, onde, quem invoca o método de um objecto não tem de esperar pelo seu resultado. O sistema de execução cria grafos de execução e gere a comunicação e sincronização existente. Uma grande vantagem deste ambiente de programação é a sua portabilidade, sendo executável num leque considerável de arquitecturas MIMD.

O **Code 2.0** [NB92] é um modelo de programação gráfica independente da arquitecturas onde possa ser implementado e que permite construir aplicações paralelas como grafos de evolução dinâmicos onde os nós são computações sequenciais e os arcos definem as dependências entre os nós.

Para criar um programa em Code é necessário seguir vários passos: desenhar a estrutura da comunicação do programa, anotar o grafo preenchendo formulários que introduzem uma série de propriedades aos nós e arcos, por exemplo que código sequencial deve ser executado por cada nó, e, por fim, escolher, de um leque de arquitecturas para as quais existe uma implementação do modelo, a arquitectura alvo da execução. Depois de ultrapassados estes passos um tradutor transforma o grafo anotado numa linguagem que possa ser compilada e executada na arquitectura escolhida.

Existe uma implementação que utiliza o PVM como linguagem alvo [Vok95], onde os elementos computacionais do Code denominados **UC** são distribuídos ao longo dos nós definidos como parte da máquina virtual paralela. Como a granulosidade de uma UC não garante o custo da sua execução paralela, mais do que uma UC é executada por um só processo. No entanto, nesta implementação só corre um processo (tarefa do PVM) em cada processador da máquina virtual. Cada tarefa do PVM corre o seu próprio *scheduler* que mantém uma fila de UCs a executar. Este corre assincronamente em relação aos outros, retirando UCs da sua fila até que o programa termine. As interacções (*data flow*) no ambiente distribuído podem ser locais ou não locais, aproveitando o conhecimento dos casos onde a interacção é local minimizando o número de mensagens enviadas.

Capítulo 3

O pSystem

O pSystem [Lop95] é um sistema que permite a implementação e execução de aplicações em arquitecturas paralelas de memória partilhada. Trata-se de uma extensão da linguagem de programação C com anotações que permitem ao programador explicitar o paralelismo. O sistema é composto por um pré-processador e um motor de execução.

O pré-processador transforma os programas anotados em código C com chamadas ao pSystem. O código resultante, depois de compilado e ligado à biblioteca `libpSystem.a` (que inclui as funções que compõem o motor de execução), está pronto a ser executado. O número de agentes que executarão o programa é definido pelo utilizador através de uma opção introduzida na linha de comandos. Todo este processo está ilustrado na figura 3.1.

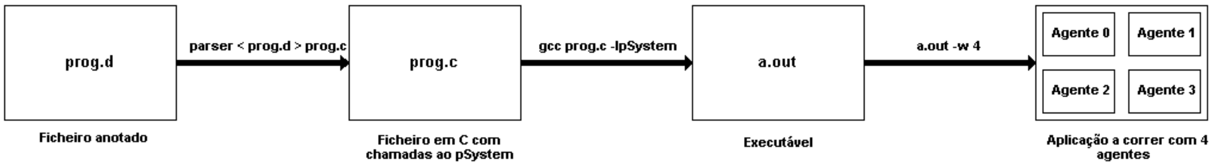


Figura 3.1: Compilação e execução de uma aplicação no pSystem.

O desenvolvimento do pSystem teve como motivação principal a definição de uma plataforma de teste flexível, sobre a qual fosse possível estudar diferentes algoritmos de distribuição dinâmica de trabalho. Este é um aspecto ausente na maioria dos sistemas actuais, que incorporam a distribuição de trabalho no motor de execução e dificultam o estudo comparativo de estratégias de distribuição. O pSystem fornece um suporte modular de alto nível para a implementação de heurísticas de distribuição de trabalho, revelando-se assim ideal para o estudo do impacto da estratégia de distribuição de trabalho na performance de aplicações paralelas

Existem actualmente duas implementações do pSystem para arquitecturas de memória partilhada, uma baseada em processos UNIX e outra baseada em *threads*. Lopes e Silva fizeram um estudo que compara estas duas implementações em termos de performance e de recursos gastos [LS97].

3.1 A Interface Paralela

A interface paralela consiste num conjunto de anotações que estende a linguagem de programação C e permite ao programador a anotação explícita de paralelismo nas suas aplicações. Estas anotações, são posteriormente transformadas, durante a compilação, em funções C que incluem chamadas ao motor de execução do pSystem (ver secção 3.2). A sintaxe e semântica de cada função é descrita de seguida:

- `pSystem task <função> ([[Tipo_entrada][,Tipo_saída?]])` indica ao sistema que `função` pode ser executada em paralelo e qual o seu tipo. Esta declaração deve ser colocada no principio do programa ou num dos seus ficheiros `.h`.
- `Start_pSystem(int, char **)` inicia as estruturas de dados usadas pelo pSystem, nomeadamente, inicializa os agentes e a memória partilhada. As variáveis `argv` e `argc` podem ser passadas como argumentos para controlar parâmetros de sistema (por exemplo, número de agentes, parâmetros de distribuição de trabalho).
- `End_pSystem()` termina o pSystem esperando que todos os agentes terminem e libertando a memória partilhada utilizada pelo sistema e pela aplicação.
- `new task <função> ([arg1][,?arg2])` constrói uma tarefa correspondente à função `função` e atribui-a para execução num agente. A mesma função pode ser executada sequencialmente, no agente local, se for invocada da seguinte forma: `função ([arg1][,?arg2])`.
- `Wait()` permite ao programador introduzir pontos de sincronização. Suspende a tarefa a ser executada actualmente até que a árvore de computação abaixo desta tarefa termine. Durante a suspensão são escolhidas outras tarefas para executar.

- `Shmalloc(int n)` e `Shfree(int n)` reservam e libertam blocos de memória partilhada respectivamente. `Shmalloc(int)` reserva `n` bytes.

- `Initlock(lock ptr)`, `Lock(lock ptr)` e `Unlock(lock ptr)` Implementam operações sobre locks em memória partilhada, respectivamente, inicialização, aquisição e libertação. A variável `ptr` é do tipo *built-in* `lock`.

A única restrição imposta pela interface paralela ao utilizador é a utilização de um só argumento de entrada nas tarefas. No entanto, esta pode ser facilmente resolvida pela encapsulação dos vários argumentos numa estrutura de dados em memória partilhada, bastando depois passar o seu endereço como argumento.

3.2 O Motor de Execução

O motor de execução do pSystem, responsável pela execução das aplicações, está divido em três componentes: o **gestor de memória partilhada**, a **máquina de execução** e o **scheduler**.

O gestor de memória partilhada é o responsável pelas operações efectuadas sobre a memória partilhada e pela consistência da informação que nela se encontra. A consistência é garantida através de *locks* que garantem o acesso mutuamente exclusivo a estruturas de dados criticas. A correcta utilização de *locks* é da responsabilidade do utilizador que deve identificar as zonas criticas do seu programa.

O gestor de memória partilhada é o único componente do pSystem que potencialmente depende da arquitectura da máquina e/ou do seu sistema operativo. Este facto deve-se tipicamente à utilização, por motivos de eficiência, de instruções *assembly* na implementação das funções relativas aos *locks*. Para que o sistema seja implementado noutras plataformas basta adaptar este componente, mais especificamente estas funções, para a plataforma desejada.

A máquina de execução implementa o modelo de execução do pSystem. Sendo assim, é responsável pela criação das tarefas, pela sua execução em paralelo, pela sua suspensão, pela sua sincronização e retorno dos resultados. É igualmente responsável pela gestão das filas de trabalho e resultados do agente. Cada agente executa a máquina de execução sem que seja controlado por nenhum sistema central de execução. A execução progride ao sabor das dependências de dados entre as tarefas, existindo pontos de sincronização para que estas sejam respeitadas.

O *scheduler* implementa as funções responsáveis pela heurística de distribuição de trabalho. Devido à modularidade do pSystem, a introdução de uma nova heurística depende apenas da substituição do ficheiro que contém estas funções e da recompilação do modulo.

Figura 3.2: O motor de execução do pSystem.

Como se ilustra na figura 3.2, a máquina de execução interage com o gestor de memória partilhada. A interacção consiste no envio de pedidos de aquisição e de libertação de espaço de memória para tarefas e para os resultados provenientes da sua execução. A máquina de execução envia todas as tarefas recentemente criadas para o *scheduler* que, utilizando a heurística de distribuição de trabalho, as entrega a agentes para posterior execução. O *scheduler* por sua vez pode pedir trabalho à máquina de execução sempre que existam agentes no sistema com débito de trabalho.

3.3 Heurísticas de Distribuição de Trabalho

Em programação paralela a estratégia de distribuição de trabalho (dinâmica) utilizada na execução de aplicações influencia fortemente a performance observada no sistema. No entanto, o problema do mapeamento do trabalho gerado nas entidades processadoras é muito complexo e não tem uma solução que possa ser aplicada eficazmente no caso geral. Consequentemente, sempre que se desenvolve uma aplicação é importante realizar um estudo prévio. O pSystem, devido à sua modularidade, além das heurísticas que fornece, possibilita a implementação de novos algoritmos que depois de integrados poderão ser estudados. Esta característica destaca-o dos restantes sistemas e torna-o um excelente utilitário para o teste e estudo de várias heurísticas de distribuição de trabalho.

As heurísticas que o pSystem actualmente fornece são [SL94]:

Sender initiated Uma tarefa criada por um agente é atribuída ao *scheduler* para que este a entregue a um outro agente com menos trabalho. A tarefa é entregue ao primeiro agente que obedeça à condição, caso não exista, a tarefa é colocada na fila de trabalho do próprio agente. Quando um agente necessita de trabalho retira uma tarefa da sua fila de trabalho, bloqueando caso esta esteja vazia.

Receiver initiated As tarefas criadas são colocadas na fila de trabalho local e vão sendo retiradas pelo agente ao longo da execução. Quando a fila está vazia é enviado um pedido de trabalho ao *scheduler*. Este pesquisa os agentes e retira uma tarefa quando encontra a primeira fila não vazia

Adaptive I Esta heurística combina as duas anteriores. O agente está sujeito a dois estados dependendo do volume de trabalho que dispõe no momento. Quando o número de tarefas é inferior a um parâmetro predefinido o agente comporta-se como na heurística *receiver initiated*, caso contrário comporta-se como na heurística *sender initiated*.

Adaptive II Esta heurística generaliza a anterior introduzindo um novo estado (estado neutro) e utilizando dois parâmetros, em vez de um, denominados **high** e **low**. Abaixo do parâmetro *low* o agente comporta-se como *receiver initiated* e acima do parâmetro *high* comporta-se como *sender initiated*. O objectivo do algoritmo é tentar manter o agente no estado neutro durante a maior parte da execução, ou seja, entre os dois parâmetros. Neste estado as tarefas são colocadas na fila local desde que isso não implique uma mudança de estado e entregues ao *scheduler* caso contrário.

A criação de uma nova heurística depende apenas da implementação de duas funções, uma que indica onde o trabalho criado por um agente deve ser colocado e outra que define a estratégia de procura de trabalho do agente.

Capítulo 4

di_pSystem: O pSystem em Arquitecturas de Memória Distribuída

Este capítulo descreve o **di_pSystem** (pSystem em arquitecturas de memória distribuída). Actualmente, com a crescente velocidade das redes de interligação e com um mercado pródigo em processadores rápidos e baratos, é cada vez mais fácil adquirir ou construir uma máquina paralela de memória distribuída. Este cenário, acrescido da utilização de redes de computadores como máquinas paralelas, tem contribuído para a crescente popularidade, e consequente evolução, do modelo de programação distribuído. O desenvolvimento do standard MPI contribuiu significativamente para a proliferação de ambientes de programação que oferecem ao utilizador uma máquina paralela virtual. Estudos como os realizados por LeBlanc e Markatos [ML92, LM92] também têm contribuído para este crescente interesse no modelo de programação distribuído. Estes concluem que, devido ao aumento gradual do fosso entre a velocidade dos processadores e das memórias e ao constante aumento de velocidade e largura de banda das redes de alta performance, os ganhos obtidos através da programação via troca de mensagens tendem a ser melhores do que os obtidos através da programação via memória partilhada, em arquitecturas de memória partilhada recentes.

É neste cenário que surge a principal motivação no desenvolvimento de uma implementação do di_pSystem. Pretende-se que a implementação preserve ao máximo a sintaxe utilizada nas implementações para memória partilhada. Deste modo, os programas escritos para memória partilhada podem ser executados, quase sem alterações, numa série de novas arquitecturas; memória distribuída, memória distribuída virtualmente partilhada e redes de computadores. Além disso, consegue-se esconder toda a parte de controlo de troca de mensagens, libertando o programador para o desenvolvimento do programa.

O aumento da potencialidade do pSystem, como suporte de estudos sobre heurísticas de distribuição de trabalho para arquitecturas distribuídas é outra das motivações deste trabalho.

4.1 A Interface Paralela

O di_pSystem é uma extensão à linguagem de programação C com anotações para paralelismo explícito semelhantes às utilizadas pelo pSystem. Uma alteração introduzida consiste na eliminação das funções que não têm utilidade nesta implementação, ou seja, as funções `Shmalloc()`, `Shfree()`, `Initlock()`, `Lock()` e `Unlock()` utilizadas para realizar acções sobre a memória partilhada. As restantes alterações consistem na introdução de uma nova declaração e de uma função para lidar com variáveis de leitura global (conceito a introduzir na secção 4.3) e de variantes ao construtor `new task`. Em resumo, a interface paralela do di_pSystem é idêntica à utilizada para memória partilhada, sem as funções relativas a acções sobre memória partilhada e com as seguintes alterações:

- `di_pSystem_var tipo_da_variável nome_da_variável` declara uma variável como sendo de leitura global.
- `Read(&variável)` garante que o valor da variável de leitura global `variável` está correcto.
- `new task <função> ([arg1][,?arg2])` análoga à anotação correspondente do pSystem para memória partilhada. No entanto, a passagem de um vector de elementos, que era feita através de um apontador para memória partilhada, tem de ser explicita. Sendo assim, o programador tem de indicar o apontador para o vector e o número de elementos a serem enviados. Por exemplo,

  ```
  new task <função>(*arg1, 10, ?*arg2, 20)
  ```

  indica que o argumento de entrada é um vector de 10 elementos do tipo de `arg1` e o argumento de saída é um vector de 20 elementos do tipo de `arg2`.

O processo de compilação do ficheiro anotado é idêntico ao descrito na figura 3.1

4.2 O Módulo de Comunicação

Nesta implementação, o módulo de comunicação não foi concebido de raiz, tendo-se recorrido ao MPI como suporte de comunicação para a implementação das funções de comunicação entre os agentes do di_pSystem. A concepção do di_pSystem desta forma, permite beneficiar de um conjunto de primitivas de comunicação standard. Por outro lado, o módulo de comunicação é modular e pode ser adaptado a outros sistemas ou concebido de raiz, sem que para isso seja necessário reprogramar qualquer outra parte do sistema.

O di_pSystem difere do pSystem no que se refere ao lançamento de uma aplicação. Ao contrario do pSystem em que o mapeamento dos agentes é feito pelo sistema operativo, no di_pSystem será necessário indicar quais os nós (máquina pertencente a uma rede de computadores ou processador de uma máquina multiprocessador) que farão parte do ambiente de execução e quantos agentes executarão em cada um dos nós. A definição da topologia da execução deve ser colocada num ficheiro que será referenciado na linha de comandos aquando do lançamento da aplicação.

O modelo de execução utilizado é o SPMD e portanto em cada nó definido corre uma (ideal) ou mais cópias do programa. Cada cópia do programa executa um agente. O arranque da aplicação depende, essencialmente, da forma como o sistema utilizado no módulo de comunicação inicializa uma computação na topologia definida. Neste caso particular, o MPI lança automaticamente o número de cópias do programa definidas em cada nó.

Cada agente tem um identificador único que o identifica perante os outros. O comportamento do agente com identificador zero, denominado principal, difere dos restantes. Neste agente a função `Start_di_pSystem()` retorna, permitindo ao agente a execução do resto da aplicação. Todos os outros agentes, denominados secundários, bloqueiam nesta função à espera de trabalho. Sendo, momentaneamente, o único em modo de execução, o agente principal é o primeiro a gerar trabalho. Este será distribuído pelos outros agentes que o recebem e executam, gerando, eventualmente, mais trabalho. Num estado mais avançado da computação todos os agentes farão parte integrante da execução da aplicação lançada. Quando todo trabalho paralelo estiver executado e o di_pSystem terminar, a cópia do programa relativa ao agente principal manter-se-á em activa para executar o resto da aplicação.

4.3 O Conceito de Variável Global

Apesar de a implementação actual não oferecer um suporte completo, via software, para memória distribuída virtualmente partilhada, o di_pSystem permite a leitura de variáveis globais nas suas tarefas. Uma variável de leitura global é uma variável do espaço de endereçamento global, declarada como `di_pSystem_var` que pode ser acedida, para leitura, por qualquer agente durante a execução de qualquer tarefa.

Antes do di_pSystem ser iniciado o conteúdo de cada variável é igual em todos os agentes. Uma variável global só pode alterada na componente sequencial, executada pelo agente principal. No entanto, se esta for declarada como sendo uma variável de leitura global, a sua alteração é propagada a todos os agentes. Este processo levanta o problema da coerência das cópias das variáveis. Para o resolver, o di_pSystem implementa dois modelos: a *invalidação* e a *actualização* do conteúdo das variáveis [LP89].

A **invalidação** do conteúdo de uma variável no di_pSystem, consiste na invalidação das cópias existentes nos agente secundários, sempre que a variável seja alterada no agente principal. Quando um destes agentes necessitar do conteúdo de uma variável cuja cópia local foi invalidada, faz um pedido, do valor actual, ao agente principal.

Para que cada agente possa determinar se o valor de uma variável está correcto, ou não, dispõe de uma tabela denominada *cache*. O valor da variável está correcto se e só se o seu endereço se encontrar na *cache*. Quando um agente (secundário) acede a uma variável de leitura global, tem de verificar se o seu endereço se encontra na *cache*. Se estiver, pode utilizar o seu valor sem restrições, senão tem de enviar um pedido do valor ao agente principal e bloquear à espera da resposta. Quando o pedido é satisfeito, o valor é enviado ao agente e atribuído à variável. O endereço da mesma variável é adicionado à *cache* (figura 4.1).

A **actualização** do conteúdo de uma variável de leitura global (figura 4.2), consiste na propagação, a todos os agentes, de qualquer alteração verificada em qualquer variável deste tipo. Desta forma, o conteúdo de uma variável de leitura global está sempre actualizado em qualquer agente, pelo que é dispensável o uso da *cache*.

Sempre que um agente recebe uma mensagem relativa à actualização de uma variável de leitura global, invoca um *lock* sobre a mesma. O *lock* garante que o agente não acede à variável antes que a actualização tenha sido completada. Depois da operação estar concluída, o *lock* é libertado para que o acesso à variável seja livre.

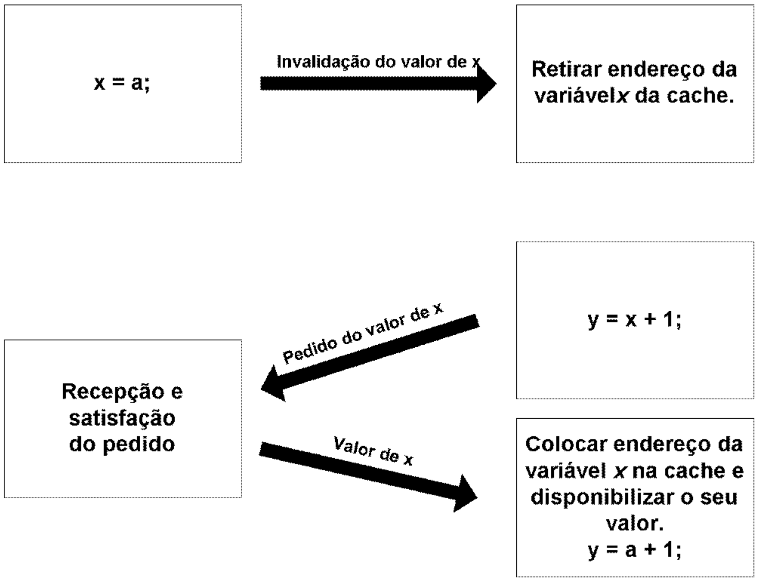


Figura 4.1: Modelo de invalidação de uma variável.

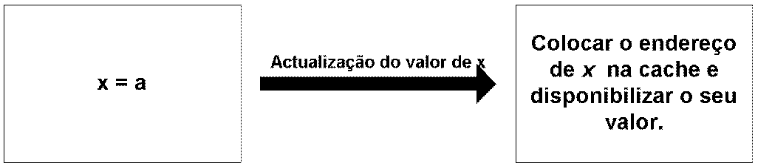


Figura 4.2: Modelo de actualização de uma variável.

4.4 A Estrutura de um Agente

Para que a sua concepção fosse mais intuitiva e estruturada, cada agente compõem-se por um **thread de execução** e um **thread de comunicação**. Cada um destes *threads* tem acesso a dois módulos: o **scheduler** e o **gestor de funções e variáveis**. O *thread* de comunicação tem ainda acesso ao **módulo de comunicação**. A progressão da computação em cada agente depende da interacção e cooperação destes cinco módulos que o compõem. A figura 4.3 ilustra estes módulos e quais os fluxos de interacção entre eles.

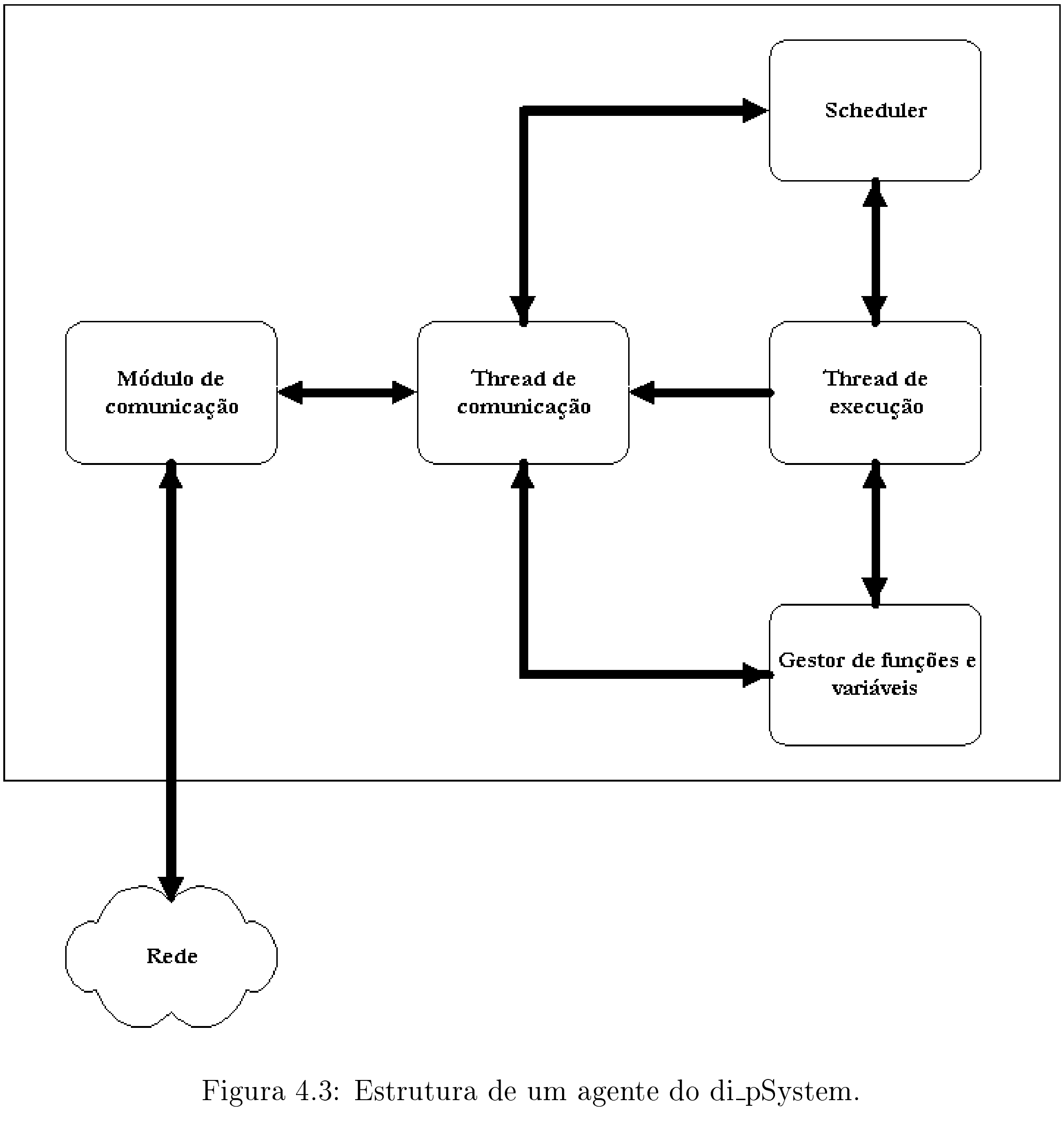


Figura 4.3: Estrutura de um agente do di_pSystem.

4.4.1 O Gestor de Funções e Variáveis

O di_pSystem destina-se a ser utilizado em redes heterogéneas de computadores. Uma vez que a identificação de funções e variáveis de leitura global de âmbito distribuído é feita por nome, é necessário implementar um mecanismo de resolução de nomes. Este mecanismo converte nomes de funções e/ou variáveis de leitura global em endereços locais num agente. A responsabilidade da implementação deste mecanismo pertence ao gestor de funções e variáveis que mantém, para o efeito, duas tabelas de endereços, uma para as funções e outra para as variáveis. O gestor de funções e variáveis é ainda responsável pela gestão da *cache* sediada em cada agente secundário. A figura 4.4 ilustra os componentes deste módulo.

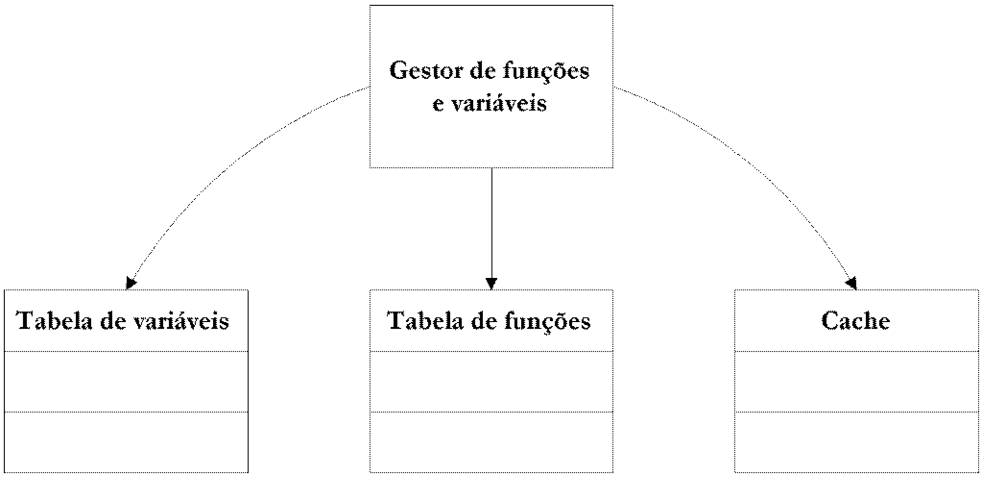


Figura 4.4: Componentes do gestor do funções e variáveis.

As funções que este módulo implementa são:

```
void Variable_Table_Init(int )
void Varible_Table_Insert (void *, int)
void Funtion_Table_Init(int)
void Funtion_Table_Insert (void *)
void Read_Variable(void *)
void Propagate_Value(void *)
int  Get_Variable_Index(void *)
int  Get_Function_Index(void *)
void Cache_Init(int)
void Cache_Insert_Element(void *)
void Cache_Delete_Element(void *)
int  Cache_Find_Element(void *)
```

O gestor de funções e variáveis interage (figura 4.3) com o *thread* de execução quando este lhe pede o índice de um endereço que se encontre nas suas tabelas, ou que verifique se uma certa variável se encontra na *cache*. Interage também com o *thread* de comunicação quando este lhe indica que o valor correcto de uma variável foi recebido, pelo que o seu endereço deve ser colocado na *cache*.

4.4.2 O *Thread* de Execução

A execução de uma aplicação baseia-se na criação, execução e suspensão de tarefas. Esta é a função do *thread* principal do sistema denominado *thread* de execução. Uma tarefa pode ser executada localmente, pelo agente que a criou, ou remotamente, por um outro agente. Quando a execução é local, o *thread* de execução é responsável pela colocação do resultado no endereço destino. As funções que implementa são:

```
void Start_di_pSystem(int, char **)
void End_di_pSystem()
void Make_Task(void (*)(), void *, long, long )
void Run_Task(Task *)
void Handle_Task()
void Put_Result(Task *)
void Make_Result(Task *)
void Wait()
```

A criação e suspensão de uma tarefa são explicitamente indicadas pelo utilizador. No entanto, sempre que é criada uma tarefa existe uma interacção (figura 4.3) entre o *thread* de execução e o *scheduler*, para que este último decida se a execução da tarefa deve ser local ou remota. Além de interagir com o *scheduler* e o gestor de funções e variáveis, o *thread* de execução também interage com o *thread* de comunicação, ao entregar-lhe os resultados de tarefas que executou e cujo resultado deve ser enviado para agentes remotos.

4.4.3 O *Thread* de Comunicação

O *thread* de comunicação é a entidade responsável por toda a comunicação entre um agente e os demais. Consequentemente, é o único componente a ter acesso à módulo de comunicação. Entre as suas competências contam-se:

- a colocação nos endereços respectivos, dos resultados das tarefas que foram executadas remotamente;
- envio dos resultados das tarefas cuja colocação é da responsabilidade de outro agente;
- envio e recepção de informação relativa à evolução da execução no sistema (tarefas, informação sobre balanceamento de carga, pedidos e coerência de valores de variáveis globais).

As funções que compõem este módulo são:

```
void Communication(Args *)
void Probe_Message()
void Send_Task(int, Task *)
void Receive_Task(int)
void Send_Work_Request()
void Send_Result()
void Receive_Result(int)
void Send_Variable(int)
void Receive_Variable()
void Inform_Copies(int)
void Send_Info_Request()
void Send_Work_Info()
void Finalize()
```

Destas, apenas a primeira não se inclui no módulo de comunicação, que consiste num submódulo do *thread* de comunicação que contém as funções necessárias para a comunicação.

O *thread* de comunicação interage (figura 4.3) com o *scheduler* para a distribuição de tarefas geradas dinamicamente.

4.4.4 O *Scheduler*

O *scheduler* é o responsável pela implementação da heurística de distribuição de trabalho utilizada por um agente. A distribuição de trabalho é descentralizada, tendo, cada agente, um *scheduler* que, através de informação relativa ao trabalho local e global, decide para onde deve enviar ou pedir trabalho. As funções que implementa são:

```
void Sched_Publishing_Tasks()
int  Sched_Management()
```

O utilizador pode escolher uma heurística de distribuição de trabalho de um leque de heurísticas disponibilizadas pelo di_pSystem, ou, se preferir, implementar a sua própria heurística. Uma heurística de distribuição de trabalho envolve duas funções. `Sched_Publishing_Task()` invocada a partir do *thread* de execução que, mediante o volume de trabalho local, faz uma primeira decisão sobre a execução local ou remota de cada tarefa recém criada. A decisão final está a cargo da função `Sched_Management()`, invocada a partir do *thread* de comunicação. Esta decide, em função da carga geral de trabalho, quais são os agentes receptores das tarefas a enviar. Se todos os agentes tiverem trabalho suficiente, estas são entregues, para balancear o sistema, ao que tiver menos. Se for o próprio agente é colocada para execução local. Esta função pode ainda actualizar o estado do agente, por exemplo, *receptor* ou *emissor* na heurística *adaptive I*. A implementação de uma nova heurística depende apenas da programação destas duas funções, e é simplificada pela disponibilização de um conjunto de estruturas de dados e de variáveis que contêm informação relevante para a distribuição de trabalho.

O *scheduler* interage (figura 4.3) com o *thread* de execução e com o *thread* de comunicação, da forma mencionada na descrição destes módulos.

Capítulo 5

A Implementação do di_pSystem

Este capítulo descreve a implementação do di_pSystem. De inicio descreve as estruturas de dados que servem de suporte à sua execução. Em seguida descreve o modelo geral de execução, para uma melhor compreensão do sistema. Por fim, apresenta os algoritmos desenvolvidos para implementar as fases de execução do sistema.

5.1 Estruturas de Dados

Para melhorar a interacção entre os dois *threads* foram introduzidas duas filas de tarefas: **fila de tarefas a executar** (`ExecTaskQueue`) e **fila de tarefas a enviar** (`ExportTaskQueue`). A `ExecTaskQueue` contém as tarefas a serem executadas localmente, a `ExportTaskQueue` contém as tarefas que poderão ser executadas remotamente, dependendo do balanceamento da carga de trabalho no ambiente de execução. Esta abordagem reduz a competição entre os *threads* no acesso à filas de tarefas, visto que apesar de ambos os *threads* poderem colocar tarefas nas duas filas, só o *thread* de execução pode retirar tarefas da `ExecTaskQueue` e apenas o *thread* de comunicação pode retirar tarefas da `ExportTaskQueue`. Além desta vantagem, esta abordagem permite um melhor controle sobre a atribuição de tarefas, visto que se todas as tarefas fossem colocadas na mesma fila, o balanceamento de carga do sistema poderia não ser tão eficaz. Este facto poderia se verificar, especialmente, em aplicações que criam tarefas que invocam um ponto de sincronização no inicio da sua execução. Pelo que a criação e consumo, pelo agente local, de várias destas tarefas poderia acontecer sem que o *thread* de comunicação tivesse conhecimento. Se o código a executar depois do ponto de sincronização for moroso, o sistema deixa de estar balanceado, porque depois de ser consumida por um agente, uma tarefa não pode ser entregue a outro.

As filas `ExportTaskQueue` e `ExecTaskQueue` consistem em filas de elementos do tipo `Task`. Cada um destes elementos representa uma tarefa e tem associado a informação que esta necessita para o seu envio, execução e retorno do resultado. Segue-se a descrição dos campos que compõem o tipo `Task`:

```
typedef struct task {
 int  id,              /* identificador da função anotada como
                          tarefa a executar em paralelo             */
 int  suspended,       /* número de subàrvores a executar abaixo
                          do nó da tarefa                           */
 long argument_size,   /* tamanho do argumento de entrada da tarefa */
 void *argument,       /* argumento de entrada da tarefa            */
 void *result,         /* resultado da execução da tarefa           */
 int  worker,          /* identificador do agente que gerou a tarefa */
 long result_size,     /* tamanho do resultado da tarefa            */
 void *address,        /* endereço onde deve ser colocado o resultado */
 struct task *parent,  /* tarefa que gerou esta tarefa              */
 struct task *next     /* apontador para a próxima tarefa na fila   */
}Task;
```

Os campos `result_size`, `address` e `parent`, não são utilizados na execução de uma tarefa, mas apenas no retorno do resultado para o endereço correcto no espaço de endereçamento do agente que gerou a tarefa. Em programação paralela, é comum que uma tarefa seja lançada antes que outra termine. Para manter a consistência no estado do programa é, por vezes, necessário aguardar até que sejam retornados os resultados de tarefas lançadas previamente. Cada tarefa conhece o endereço de retorno do seu resultado. Desta forma, se uma tarefa for executada localmente, o endereço está automaticamente disponível para a colocação do resultado, senão os campos `address`, `result`, `result_size` e `parent` da tarefa são utilizados para devolver o resultado ao agente correcto. Para que todos os resultados sejam enviados, é necessário manter uma lista que referencie as tarefas cujos resultados aguardem o envio. Esta fila, constituída por elementos do tipo `Result`, é denominada por **fila de resultados** (`ResultQueue`). Segue-se a descrição dos campos do tipo `Result`:

```
typedef struct result {
 Task *task,            /* apontador para a tarefa que terminou
                           e cujo resultado deve ser enviado          */
 struct result *next    /* apontador para o próximo resultado na fila */
}Result;
```

Uma aplicação desenvolvida no di_pSystem executa em todos os nós definidos pelo utilizador, portanto tem de ser compilada em todas as máquinas que os englobam. Para possibilitar a utilização de redes heterogéneas de computadores onde as máquinas possuem arquitecturas distintas, logo, conjunto de instruções distintos, é necessário considerar o problema de o endereço de uma função ou variável poder diferir de máquina para máquina. Para o resolver são introduzidas duas tabelas durante a transformação do ficheiro di_pSystem para um ficheiro C. Após uma inicialização, no arranque da aplicação, estas tabelas nunca são alteradas e portanto os seus acessos são exclusivamente de leitura.

A tabela de funções, `FUNCTION_TABLE`, é um vector de elementos do tipo `Function_table`, onde cada elemento contém o endereço de uma das funções declaradas como sendo tarefas do di_pSystem. A tabela de variáveis, `VARIABLE_TABLE`, é um vector de elementos de tipo `Variable_table` onde cada elemento contém o endereço (`address`) de uma das variáveis declaradas como de leitura global assim como o seu tamanho em bytes (`size`).

Para obter o endereço da função com identificador `i` basta aceder ao i-ésimo elemento do vector, ou seja, para obter o endereço da função associada a uma certa tarefa basta referir `FUNCTION_VARIABLE[task->id]` onde `task` é do tipo `Task`. Para se obter o endereço ou tamanho de uma certa variável o procedimento é análogo.

5.2 Modelo Geral de Execução

Para iniciar um agente o programador invoca a função `Start_di_pSysten()` que dá origem ao *thread* de execução do di_pSystem. Este inicializa as estruturas de controle de execução do agente (`ExecTaskQueue`, `ExportTaskQueue` e `ResultQueue`), lê as opções introduzidas pelo utilizador (parâmetros de controle de trabalho) e cria o *thread* de comunicação. A *cache* só é necessária nos agentes secundários (identificador diferente de 0), pois no principal é garantido que os valores das variáveis estão correctos. O algoritmo da função `Start_di_pSystem()` para um agente é o seguinte:

```
Start_di_pSystem() {
   Inicializar estruturas de informação;
   Leitura das opções do utilizador;
   se (myrank != 0)
      inicializar cache;
      introduzir variáveis de leitura global na cache;
      enquanto (di_pSystem_running)
         se (ExecTaskQueue != EMPTY)
            Handle_Task();
   }
}
```

Como se pode observar o agente principal (`myrank == 0`) tem um comportamento diferente dos restantes. Este, retorna da função `Start_di_pSystem()` para prosseguir a computação da componente sequencial do programa, enquanto os restantes agentes bloqueiam à espera de trabalho para executar. Sendo assim, numa fase inicial, este é o único agente em fase de execução, visto que todos os outros estão bloqueados à espera de trabalho. À medida que a computação avança e são criadas tarefas, estas são distribuídas por todos os agentes. Para controlar o fluxo de execução é criada uma árvore de execução por cada tarefa criada a partir da componente sequencial. Nestas árvores cada nó representa uma tarefa e cada caminho uma dependência de dados. À medida que são criadas novas tarefas são adicionadas novas subárvores ligadas à tarefa que as criou. A figura 5.1 ilustra o exemplo do cálculo no terceiro número de Fibonacci através da aplicação `Fib` (apêndice A).

Quando uma tarefa termina a árvore de computação relativa à sua execução é eliminada. Para que, num determinado ponto, uma tarefa aguarde pela conclusão da subárvore abaixo do seu nó, o programador pode suspender a sua execução através da função `Wait()`. Neste exemplo, utiliza-se este mecanismo para aguardar pelos resultados das tarefas lançadas, `Fib` tem de aguardar a conclusão de `Fib(2)` e de `Fib(1)` antes de calcular `Fib(3)`.

Uma árvore de execução pode estar espalhada pelos vários agentes, portanto estes têm de comunicar entre eles. A comunicação entre os agentes no di_pSystem está a cargo do thread de comunicação que é responsável pelo envio e recepção das mensagens.

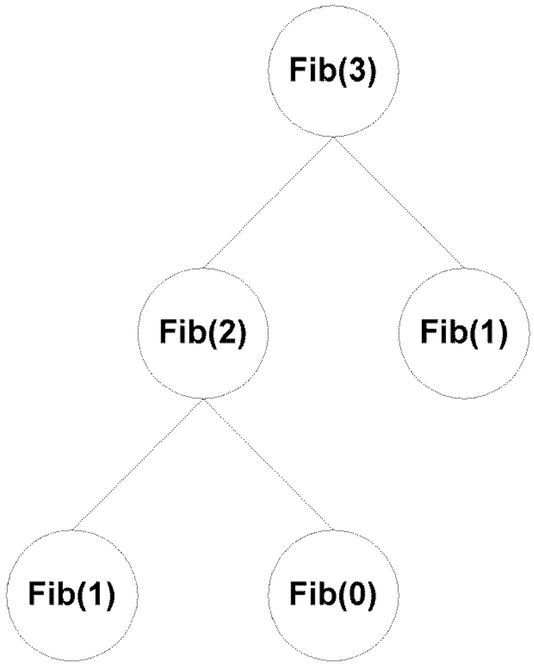


Figura 5.1: Árvore de execução de Fib(3).

Após uma fase de inicialização, o *thread* de comunicação executa um ciclo até que o di_pSystem seja finalizado. A inicialização arranca a módulo de comunicação, neste caso o MPI, e inicializa um conjunto de estruturas de dados nomeadamente: identificador do agente, número de agentes a executar, vector informativo da quantidade de trabalho de cada agente e, se estiver em modo de *debug*, um ficheiro que conterá a informação relativa à comunicação realizada entre este agente e os restantes. O ciclo é composto por quatro acções principais:

- detecção e recepção de mensagens
- envio das acções realizadas sobre variáveis de leitura global
- envio de resultados
- realizar operações relacionadas com a distribuição de trabalho

```
Communication_thread(void *args) {
   desempacotar args;
   inicializar MPI;
   myrank = obter identificador do agente;
   number_of_workers = obter número de agentes a executar;
   inicializar o  vector de quantidade de trabalho dos agentes;
   enquanto (di_pSystem_running) {
         ler mensagens;
         enviar acções sobre varáveis de leitura global;
         enviar resultados;
         operações relacionadas com distribuição de trabalho;
   }
}
```

Para distribuir uma árvore de execução pelos vários agentes, é necessário que uma tarefa criada por uma agente possa ser executada por um outro. O processo de definir onde uma tarefa vai ser executada principia na sua criação. O *thread* de execução, através da função `Sched_Publishing_Task()` entrega a tarefa ao *scheduler* que, baseado na quantidade de trabalho local, a coloca na `ExecTaskQueue` ou na `ExportTaskQueue`. Uma chamada à função `Sched_Management()`, a partir do *thread* de comunicação, conclui este mecanismo. `Sched_Management()` decide se uma tarefa na `ExportTaskQueue` deve executar localmente ou remotamente e, no segundo caso, qual o agente que a deve executar.

O *thread* de comunicação do agente que recebe a tarefa, coloca-a na sua `ExecTaskQueue` para que mais tarde esta seja executada pelo *thread* de execução. O resultado da execução é colocado na `ResultQueue` para que, posteriormente, seja enviado para o agente que criou a tarefa. Ao receber esta mensagem o *thread* de comunicação do agente receptor coloca o resultado no endereço respectivo. Este mecanismo está ilustrado na figura 5.2.

No final de toda a computação paralela deve invocar-se a função `End_di_pSystem()`. Esta garante que toda a computação paralela termina antes de sair do sistema. Isto corresponde a aguardar pela conclusão de todas as árvores de computação. Quando for seguro terminar o sistema, o *thread* de comunicação é informado através de uma variável (*flag*) partilhada. Este deve enviar a mensagem de terminação a todos os outros agentes, aguardar que todos os agentes atinjam um ponto de sincronização (mais especificamente uma barreira) e terminar a sua execução. Cada agente secundário,

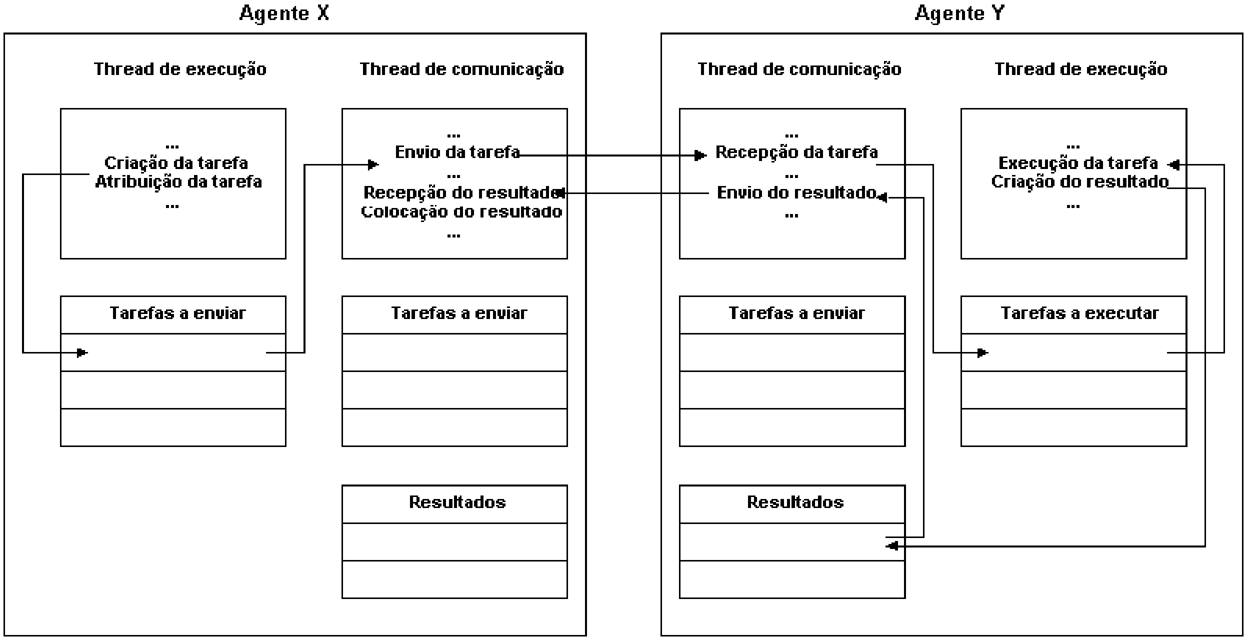


Figura 5.2: Execução remota de uma tarefa.

quando recebe esta mensagem e atinge o ponto de sincronização, termina os *threads* de execução e de comunicação e sai. Depois de terminado o di_pSystem, a aplicação pode prosseguir no agente principal, ou seja, no nó onde foi lançada. A figura 5.3 ilustra a comunicação necessária entre o agente principal (identificador 0) e os restantes agentes para terminar o di_pSystem.

5.3 Criação de uma Tarefa

A criação de uma tarefa a executar é anotada explicitamente pelo programador. Esta indicação é transformada durante a compilação numa chamada à função `Make_Task()` que cria uma nova estrutura do tipo `Task`. O primeiro campo a preencher nesta estrutura corresponde à identificação da função a executar. A sua obtenção depende da interacção com o gestor de funções e variáveis através da função `Get_Function_Index()`. De seguida, verifica-se se existe argumento de entrada para a função; se sim, este é copiado para o campo respectivo, senão, coloca-se o campo a zero. Os restantes argumentos de entrada são: o tamanho do argumento, o endereço do valor de retorno e o tamanho do resultado. Restam os campos `suspended`, que inicialmente é zero, `worker`, inicializado com número do agente (`myrank`) e `parent` que aponta para a tarefa a executar actualmente (`current_task`).

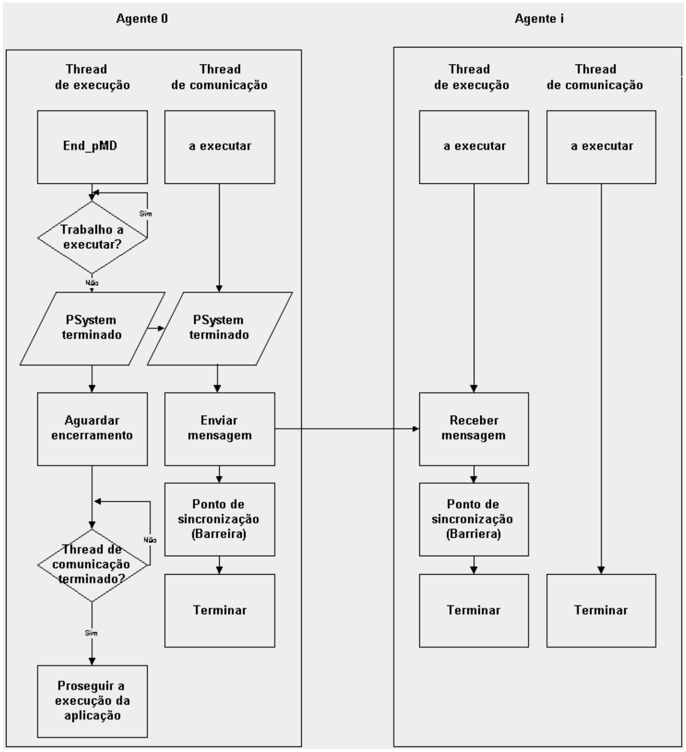


Figura 5.3: Terminação do di_pSystem.

O campo `suspended` da tarefa indica quantas subárvores se encontram sob o seu nó. Sendo assim, se uma a tarefa for criada a partir de outra, é necessário incrementar este campo na tarefa pai, para indicar que o número de subárvores por que esta tarefa deve aguardar, caso seja suspensa, aumentou. Se a tarefa for criada a partir da componente sequencial, é incrementada a variável global `suspended`, que tem o mesmo papel.

5.4 Suspensão de Execução

Sempre que o programador utiliza a função `Wait()`, a tarefa actual é suspensa passando a execução para uma nova tarefa obtida da `ExecTaskQueue`. Este processo é repetido enquanto a suspensão persistir. Consoante o local de invocação da suspensão, componente sequencial da aplicação ou uma tarefa, a variável que determina a sua continuidade varia. No primeiro caso, a suspensão depende da variável global `suspended` e no segundo do campo `suspended` da estrutura do tipo `Task` referente à tarefa. Esta abordagem evita que se tenha de criar uma tarefa *fantasma* para a componente sequencial da aplicação, quando o único campo útil é `suspended`. A suspensão persiste enquanto a variável de que depende for maior que zero, ou equivalentemente, existirem tarefas filho a executar. Quando termina a suspensão, a última tarefa suspensa volta a executar, sendo necessário actualizar a variável `current_task`. Se for a componente sequencial `current_task` fica com o valor `SEQ`.

Se não existirem mais tarefas a executar invoca-se a função `Free_Memory()` para libertar a memória que já não é necessária. O algoritmo seguinte descreve a função `Wait()`.

```
Wait() {
   se (current_task == SEQ) {
      enquanto (suspended) {
         se (ExecTaskQueue != NO_TASK) {
            Handle_Task();
            current_task = SEQ;
         }
         senão
            Free_Memory();
      }
   }
   senão {
      tarefa = current_task;
      enquanto (tarefa->suspended) {
         se (ExecTaskQueue != NO_TASK) {
            Handle_Task();
            current_task = tarefa;
         }
         senão
            Free_Memory();
      }
   }
}
```

O processo de retirar uma tarefa da `ExecTaskQueue` e activar a sua execução é da responsabilidade da função `Handle_Task()`. Se existir trabalho na fila de tarefas a executar, esta função retira uma tarefa, que passa a ser a tarefa actual, e chama a função `Run_Task()` para a executar. Quando a execução termina, é necessário colocar o resultado da computação no endereço de destino. Se a tarefa é local ao agente, o resultado é colocado automaticamente através da função `Put_Result()`, caso contrário é enviado ao *thread* de comunicação para ser colocado na fila de resultados a enviar. Esta operação, realizada através da função `Make_Result()`, é efectuada mesmo que não haja valor de retorno, pois é necessário informar o agente que gerou a tarefa que esta terminou para que este decremente o campo `suspended` na tarefa pai. A função em si, cria uma nova estrutura do tipo `Result` que aponta para a tarefa que terminou.

```
Handle_Task() {
   retirar tarefa da ExecTaskQueue;
   current_task = tarefa;
   executing_tasks++;
   Run_Task(tarefa);
   se (tarefa é local)
      Put_Result(tarefa);
   senao
      Make_Result(tarefa);
}
```

5.5 Execução de uma Tarefa

A execução de uma tarefa é feita através da função `Run_Task()`. Esta principia com o pedido do endereço da função a executar ao gestor de funções e variáveis (função `Get_Address()`), sendo depois necessário discriminar se a função tem ou não argumento de entrada e valor de retorno. Consoante o caso, a função é executada com zero, um ou dois argumentos de entrada, pois o valor de retorno é passado por referência como argumento. Assim uma função que normalmente é referida como

```
var = f(arg);
```

é representada no di_pSystem da seguinte forma: `f(arg, var)`.

```
Run_Task(Task *tarefa) {
    fp = Get_Address(tarefa->id);
    se (tarefa tem argumento)
       se (tarefa tem valor de retorno)
          (void)(*fp)(tarefa->argument, &tarefa->result);
       senão
          (void)(*fp)(tarefa->argument);
```

```
    senão
        se (tarefa tem valor de retorno)
          (void)(*fp)(&tarefa->result);
        senão
          (void)(*fp)();
}
```

5.6 Libertar Memória

A criação de um elemento das filas `ExecTaskQueue`, `ExportTaskQueue` e `ResultQueue` necessita de reservar espaço para guardar os dados relativos a cada elemento. À medida que a execução vai progredindo, as tarefas são executadas ou enviadas e os resultados enviados. Sendo assim, o espaço ocupado por estas tarefas e resultados, deixa de ser necessário, logo deve ser libertado. A libertação destes blocos de memória é feita sempre que o agente está parado, ou seja, quando está à procura de trabalho. Portanto, a função `Wait()` sempre que a `ExecTaskQueue` está vazia invoca a função `Free_Memory()` que liberta toda a memória alocada até aos elementos que ainda são necessários para a computação.

A função `Free_Memory()` utiliza os apontadores `ExportQueueGarbage`, `ExecQueueGarbage` e `ResultQueueGarbage` que indicam o primeiro elemento de cada fila a ser libertado. Os elementos vão sendo apagados até se atingir o primeiro elemento em uso de cada fila `ExportQueuePointer`, `ExecQueuePointer` e `ResultQueuePointer`. O algoritmo seguinte ilustra o mecanismo utilizado.

```
Free_Memory() {
    tarefa = ExecQueueGarbage
    enquanto (tarefa != ExecQueuePointer) {
        ExecQueueGarbage = ExecQueueGarbage->next;
        free(tarefa);
        tarefa = ExecQueueGarbage;
    }
    tarefa = ExportQueueGarbage
    enquanto (tarefa != ExportQueuePointer) {
        ExportQueueGarbage = ExportQueueGarbage->next;
        free(tarefa);
        tarefa = ExecQueueGarbage;
    }
    resultado = ResultQueueGarbage
    enquanto (resultado != ResultQueuePointer) {
        ResultQueueGarbage = ResultQueueGarbage->next;
        free(resultado);
        resultado = ResultQueueGarbage;
    }
}
```

5.7 Gestão de Variáveis de Leitura Global

As operações sobre uma variável de leitura global resumem-se à leitura e escrita sob diferentes protocolos de consistência. Ambas dependem da estrutura `variable` do tipo `VariableInfo` cujos campos são os seguintes:

```
typedef struct {
 int  id,          /* id da variável anotada como de leitura global */
 int  wait,        /* indica se a acção já foi satisfeita           */
 int  action_type, /* tipo de operação a efectuar sobre a variável  */
 void *address     /* endereço da variável sobre a qual se vai
                      realizar a operação                           */
}VariableInfo;
```

A atribuição de um novo valor a uma variável de leitura global é da competência exclusiva do agente principal o qual tem informação completa e actualizada dos valores correctos das variáveis. Para que os outros agentes também possam ter esta informação, é necessário implementar um protocolo que garanta a coerência das cópias. O di_pSystem implementa dois protocolos, *invalidação da cópia* ou *actualização da cópia* (secção 4.3). O protocolo utilizado por defeito é a invalidação, mas o utilizador pode optar pelo outro indicando-o na linha de comandos. Internamente, ambos os protocolos são da responsabilidade das funções `Propagate_Value()` e `Inform_Copies()`. Sempre que ocorre uma atribuição a uma variável de leitura global é introduzida, em tempo de compilação, uma chamada à função `Propagate_Value()`. Esta função preenche a estrutura `variable` colocando no campo `action_type` o valor da variável global `protocol` que indica o protocolo escolhido. A estrutura `variable` é partilhada pelo *thread* de comunicação que detecta a acção e, através da função `Inform_Copies()`, propaga a informação da atribuição segundo o protocolo escolhido.

Se o protocolo escolhido for a actualização o agente que recebe a informação da atribuição invoca a função `Put_Value()`. A função coloca o novo valor no endereço da variável e coloca o endereço na `cache`, usando a função `Cache_Insert_Element()`, indicando que o valor da variável está actualizado. Se o protocolo usado for a invalidação é removido o endereço da respectiva variável da *cache*, invalidando assim a cópia existente no agente. Esta operação é efectuada através da função `Cache_Delete_Element()`,

Para garantir que o valor de uma variável de leitura global está correcto, o programador pode utilizar a função `Read()` da interface paralela. Se protocolo de coerência de cópias utilizado for a actualização, o acesso a uma variável de leitura global é trivial, pois os valores estão todos actualizados. Se o protocolo escolhido for a invalidação, o único agente que tem, garantidamente, todos os valores correctos, é o agente principal. Nos outros agentes é necessário verificar se a variável está em cache (função `Cache_Find_Element()`), ou seja, se o valor da variável que se encontra actualmente no agente é o correcto. Se a resposta for positiva, a computação prossegue, senão, a função `Read()` preenche a estrutura `variable` colocando o campo `action_type` com `GET_VALUE` e bloqueia o *thread* de execução à espera do valor correcto da variável. Como a estrutura é partilhada pelo *thread* de comunicação, este detecta a operação através de uma verificação periódica incorporada no seu ciclo de execução. Depois de detectar a operação este envia o pedido ao agente principal, recebe o novo valor e invoca a função `Put_Value()`. Esta função, além de colocar o novo valor no endereço da variável e colocar este endereço na *cache*, desbloqueia o *thread* de execução.

```
Read_Variable(void *variavel, long tamanho) {
  se (agente não é principal) {
    se (elemento não está na cache) {
      variable->address = variavel;
      variable->size = tamanho;
      variable->wait = 1;
      variable->action_type = GET_VALUE;
      variable->id = Get_Variable_Index(variavel);
    }
    enquanto (variable->wait) ;
  }
}
```

5.8 O Módulo de Comunicação

Tendo em vista preservar a portabilidade do sistema, a parte de comunicação realizada pelo *thread* de comunicação constitui um módulo independente, possibilitando assim a sua reescrita sem que tal afecte o motor de execução. O módulo de comunicação é constituído por um conjunto de funções responsáveis pela comunicação entre os agentes. Estas encontram-se todas no ficheiro `communication.c`. Para modificar o módulo de comunicação, adaptando-o a um outro sistema de comunicação, basta alterar e recompilar este ficheiro. Esta secção pretende descrever o propósito de cada função e realçar pormenores de implementação que melhoram a sua eficiência, reduzindo o tráfego na rede.

5.8.1 Funções de Comunicação

A detecção e recepção de mensagens está a cargo da função `Probe_Message()`. Em função do tipo de mensagem recebida, operações distintas são executadas.

A recepção de um novo resultado é da responsabilidade da função `Receive_Result()`. Depois de receber toda a informação relativa à colocação de um resultado (valor, endereço a colocar, tamanho do resultado e tarefa pai), verifica se este é nulo. Se a resposta for negativa coloca-o no endereço correspondente. Em seguida decrementa o campo `suspended` da tarefa pai, para que esta saiba que uma das subárvores abaixo do seu nó terminou. O envio de um resultado é efectuado pela função `Send_Result()` que empacota a informação e envia-a para o agente correspondente. O tipo de mensagens envolvido na recepção e envio de resultados é:

- `RESULT` – novo resultado.

A recepção de tarefas está a cargo da função `Receive_Task()` que as coloca no fim da `ExecTaskQueue`, actualizando a quantidade de trabalho local do agente. Quando é recebida uma mensagem relativa a um pedido de trabalho, a função `Receive_Work_Request()` verifica se existem tarefas a exportar (fila `ExportTaskQueue` não vazia). Se existirem, envia a primeira tarefa desta fila; senão envia uma mensagem com a informação de que não existe trabalho a enviar. O agente que recebe esta mensagem terá de recorrer a outro agente para obter trabalho. Um agente efectua um pedido de trabalho de cada vez.

Para se enviar uma tarefa utiliza-se a função `Send_Task()` que envia uma estrutura do tipo `Task`. Para se enviar um pedido de trabalho é utilizada a função `Send_Work_Request()`. Os tipos de mensagens envolvidos na comunicação relativa ao envio, recepção e pedido de trabalho são:

- `NEW_WORK` – nova tarefa a executar.
- `WORK_REQUEST` – pedido de uma tarefa.
- `NO_WORK` – não existem tarefas disponíveis para enviar.

Para que um agente possa informar os restantes sobre a sua quantidade de trabalho e possa obter, por sua vez, a quantidade de trabalho destes, existem duas funções: `Send_Info_Request()` e `Send_Work_Info()` respectivamente. As mensagens envolvidas são do tipo:

- `WORK_INFO` – informação sobre a quantidade de trabalho.
- `INFO_REQUEST` – pedido de informação sobre quantidade de trabalho.

A função `Inform_Copies()` tem como funcionalidade propagar a informação da alteração de uma variável de leitura global consoante o protocolo de consistência de cópias. Para pedir o valor actualizado de uma variável é invocada a função `Variable_Request()`. O agente principal ao receber esta mensagem obtém o endereço da variável e envia o valor que detém através da função `Send_Variable()`. O agente que efectuou o pedido ao detectar a mensagem referente ao valor actualizado recebe-o e invoca a função `Put_Value()`. O tipo de mensagens utilizado nestas comunicações são:

- `VAR_REQUEST` – pedido do conteúdo actualizado de uma variável global.
- `NEW_VAR` – envio do conteúdo actualizado de uma variável global.
- `INVALIDATE` – invalidação do conteúdo de uma variável global.
- `UPDATE` – actualização do conteúdo de uma variável global.

As mensagens do tipo `INVALIDATE` e `UPDATE` só podem ser enviadas pelo agente principal, visto que só este pode alterar, na execução da componente sequencial, variáveis de leitura global.

Para que o *thread* de comunicação do agente principal informe os restantes sobre o terminar da computação, invoca a função `Finalize()`. Esta envia uma mensagem do tipo `END` e invoca um barreira para sincronizar com os restantes agentes. Os outros agentes ao receberem a mensagem, invocam também a barreira. Depois de os *threads* de comunicação de todos os agentes se encontrarem sincronizados, isto é, tiverem atingido a barreira, terminam o MPI e a sua execução.

5.8.2 A Utilização do MPI

O MPI é inicializado com a função `MPI_Init()`, sendo depois obtido o identificador do agente e o número de agentes a executar através das funções `MPI_Comm_Rank()` e `MPI_Comm_Size()` respectivamente.

No di_pSystem existem dois tipos de mensagens: as simples que permitem o envio de, por exemplo, uma simples variável ou da informação sobre a quantidade de trabalho; e as compostas, que permitem o envio de uma tarefa ou de um resultado. Mensagens simples podem ser implementada por uma simples chamada à função `MPI_Send()`. Para se enviar uma mensagem composta, como seja, uma tarefa, não se envia uma mensagem por cada campo do tipo `Task`. Para reduzir o tráfego na rede e o custo do envio de uma mensagem, é empacotada toda a informação num vector, através da função `MPI_Pack()`, que depois é enviado. No entanto, para se desempacotar a mensagem no agente que a recebe é necessário saber o tamanho do vector. Como, por exemplo, o argumento de uma tarefa e um resultado não têm um tamanho predefinido, pois imporia uma restrição muito significativa ao programador, não existe forma de determinar antecipadamente o tamanho de uma mensagem relativa a uma tarefa ou resultado. A solução encontrada passa por preceder o envio desta mensagem pelo envio de uma outra, que contém o tamanho do vector a desempacotar.

O MPI permite o envio de estruturas directamente através da construção de um novo tipo de dados, ou seja, uma tarefa poderia ser enviada como sendo do tipo, por exemplo, `MPI_TASK`. No entanto, esta facilidade obriga as estruturas a terem um tamanho fixo. Como já foi referido, tal não acontece no di_pSystem, pelo que, esta hipótese foi abandonada.

A função do di_pSystem `Probe_Message()` tem como base a função `MPI_Probe()`. Esta detecta se alguma mensagem foi recebida e se sim coloca nos campos da estrutura do tipo `MPI_Status`, passada como argumento por referência, informação relativa à mensagem recebida (e.g., o valor da *tag* e identificador do agente). Deste modo, é fácil discriminar qual a operação a efectuar em resposta à recepção de uma mensagem.

Para se evitar *deadlocks* o envio de qualquer mensagem é precedido pela invocação da função `Probe_Message()`. Desta forma, obriga-se que todas as mensagens *bufferizadas* sejam lidas antes de se efectuar um envio. Pois, apesar de o di_pSystem usar a função não bloqueante `MPI_Isend()`, é necessário completar o envio de uma mensagem, através da função `MPI_Wait()`, sempre que se efectua um novo envio, ou seja, garantir que o primeiro foi recebido. Este processo evita que a informação do primeiro envio seja sobrescrita, sem que seja recebida. Se estiverem dois agentes a completar um envio em simultâneo, ambos estão bloqueados à espera que o receptor leia a mensagem. Se um agente X estiver à espera da recepção por parte de outro agente Y e vice-versa temos uma situação de bloqueio em que nenhum dos agentes poderá sair.

A recepção de uma mensagem simples é implementada com uma chamada à função `MPI_Recv()`. A recepção de uma mensagem composta implica duas chamadas a esta função, para receber o tamanho do vector e o vector em si, implica ainda o desempacotamento do vector, através da função `MPI_Unpack()`.

A junção de mensagens numa só, pode ser muito importante para minimizar o número de mensagens na rede. No di_pSystem sempre que se envia uma mensagem junta-se-lhe a informação sobre a quantidade de trabalho actual do agente, evitando assim o envio de uma nova mensagem. A quantidade de trabalho do agente também é propagada sempre que uma mensagem não contenha nenhum valor, ou seja, quando a única informação extraída da mensagem é a sua *tag*. Esta situação verifica-se nas mensagens do tipo `NO_WORK` e `WORK_REQUEST`. Isto permite manter informação mais precisa sobre a distribuição de trabalho no sistema num dado momento, sem penalização de comunicação.

Quando a função `Finalize()` é executada o *thread* de comunicação termina invocando a função `MPI_Finalize()` que por sua vez termina o MPI.

Capítulo 6

Análise de Performance

Este capítulo apresenta o estudo de performance realizado no di_pSystem. O estudo foi elaborado através da execução de três aplicações num conjunto de arquitecturas diferentes. As aplicações foram executadas utilizando as quatro heurísticas de distribuição de trabalho implementadas (secção 3.3). Os resultados são complementados por uma comparação entre o sistema implementado e o utilizado para auxilio na comunicação, i.e., o MPI, e por um estudo de particularidades a que o sistema é sensível: granulosidade das tarefas e velocidade da rede de interligação.

6.1 Aplicações de Teste

Para a realização de um estudo inicial à performance do di_pSystem foram escolhidas três aplicações simples, cujo comportamento é amplamente conhecido. Um estudo mais profundo, onde serão desenvolvidas aplicações com um grau de complexidade superior (cadeias de Markov, simulação de fluídos), fazem parte do trabalho futuro a realizar sobre este sistema. Este estudo mais pormenorizado não foi possível de ser realizado durante a tese, devido à escassez de tempo, que foi essencialmente conduzido para o desenho modular do sistema e sua eficiente implementação. As aplicações de teste desenvolvidas foram as seguintes:

Fibonacci – Fib(n) – Aplicação que calcula o n-ésimo número de Fibonacci, através da definição

$$Fib(n) = \begin{cases} 1 & \text{se } 0 \leq n \leq 1, \\ Fib(n-1) + Fib(n-2) & \text{se } n > 1. \end{cases} \tag{6.1}$$

Uma instância de `Fib()` é uma tarefa executada em paralelo. Cada uma destas tarefas gera mais duas, e assim sucessivamente, portanto a geração de tarefas é regular e dependente de `n`. No entanto, foi introduzido um parâmetro `Gr` que define a granularidade das tarefas. Este, define a partir de que `n` as tarefas não devem ser entregues ao *scheduler* para execução paralela, devendo, portanto, ser executadas localmente no agente que as gerou.

Multiplicação de matrizes – Mmat(n) – Aplicação que calcula a multiplicação de uma matriz de tamanho **n x n**, por ela própria. A matriz é criada aleatoriamente por uma pequena aplicação externa e colocada num ficheiro para ser lida por esta aplicação. A criação das tarefas é efectuada, na sua totalidade, pela componente sequencial da aplicação. O número de linhas, da matriz resultado, a calcular por cada tarefa é indicada pela divisão de `n` pela variável `dpS_SIZE`, fornecida pelo sistema, que indica o número de agentes a executar. O número de linhas resultante do resto da divisão são calculadas, localmente, pelo agente principal.

Cálculo de $\boldsymbol{\pi}$ – Pi(n) - Aplicação que calcula o número $\boldsymbol{\pi}$, através da relação

$$\frac{\pi}{4} = \int_0^1 \frac{1}{1+x^2} \tag{6.2}$$

O número `n` indica o número de partições do intervalo [0,1], onde a função vai ser integrada, usando a regra do trapézio. As tarefas são todas geradas na componente sequencial do programa. O número de tarefas geradas é igual ao número de partições do intervalo, `n`. A granulosidade de cada tarefa é dada pelo parâmetro `Gr` que indica quantas subpartições tem cada partição. A geração de trabalho paralelo desta aplicação difere da aplicação `Mmat`, pois o número de tarefas é diferente, provavelmente superior, do que o número de agentes a executar, portanto, a componente de comunicação tem um peso mais significativo na performance total.

6.2 Heurísticas de Distribuição de Trabalho

As heurísticas de distribuição de trabalho utilizadas são as descritas na secção 3.3. A sua implementação no di_pSystem é ligeiramente diferente, à realizada no pSystem, devido à utilização de dois *threads* e de duas filas de trabalho. Segue-se uma descrição algorítmica de cada uma das heurísticas.

Sender Initiated

```
Sched_Publishing_Task(tarefa) {
  se (quantidade de trabalho local < limite_inferior)
    inserir tarefa na ExecTaskQueue;
  senão
    inserir tarefa na ExportTaskQueue;
}

Sched_Management() {
  se (ExportTaskQueue != EMPTY)
    se (quantidade de trabalho local < limite_inferior)
      retirar tarefa da ExportTaskQueue;
      colocar tarefa na ExportTaskQueue;
    senão {
      encontrar o agente para enviar tarefa;
      se (agente != agente local)
        enviar tarefa ao agente;
      senão
        colocar tarefa no agente local;
    }
  }
}
```

Receiver Initiated

```
Sched_Publishing_Task(tarefa) {
   colocar tarefa na ExecTaskQueue;
}

Sched_Management() {
  se (não existe pedido de trabalho) {
    se (quantidade de trabalho local < limite_inferior) {
      encontrar agente para enviar pedido;
      se (agente != agente local) {
        enviar pedido de trabalho a agente;
        colocar flag de pedido de trabalho a true;
      }
      senão
        enviar pedido de informação de quantidade de trabalho.
    }
  }
}
```

Adaptive I

```
Sched_Publishing_Task(tarefa) {
  se (agente é RECEIVER)
    Sched_Publishing_Task(tarefa) do Receiver Initiated;
  senão
    Sched_Publishing_Task(tarefa) do Sender Initiated;
  actualizar estado do agente;
}

Sched_Management() {
  se (agente é RECEIVER)
    Sched_Management() do Receiver Initiated;
  else
    Sched_Management() do Sender Initiated;
  actualizar estado do agente;
}
```

Adaptive II

```
Sched_Publishing_Task(tarefa) {
  se (agente é RECEIVER)
    Sched_Publishing_Task(tarefa) do Receiver Initiated;
  senão
    se (agente é SENDER)
      Sched_Publishing_Task(tarefa) do Sender Initiated;
    senão
      se (quantidade de trabalho mais um mantém o agente NEUTRAL)
        coloca tarefa na ExecTaskQueue;
      senão
        coloca tarefa na ExportTaskQueue;
  actualizar estado do agente;
}

Sched_Management() {
  se (agente é RECEIVER)
    Sched_Management() do Receiver Initiated;
  else
    Sched_Management() do Sender Initiated;
  actualizar estado do agente;
}
```

6.3 Ganhos de Velocidade

Os resultados desta secção foram obtidos num *cluster* denominado **Dolphin 1**, constituído por quatro *dual* Pentium II a 300MHz, 128 MBytes de RAM e 4GBytes de disco UW_SCSI, ligados por uma rede Myrinet a 1 GBit/segundo. O sistema operativo utilizado foi o RedHat 5.1 Linux/smp. É de salientar que este *cluster* foi desenvolvido recentemente e ainda não está optimizado. Obviamente, este facto irá reflectir-se nos resultados obtidos.

A tabela 6.1 indica o número de tarefas geradas e o número de sincronizações utilizadas na execução paralela da cada aplicação, utilizando o di_pSystem. A escolha da granulosidade das tarefas das aplicações foi sujeita a estudo. Foi concluído que nesta arquitectura o di_pSystem pode obter *speedups* com tarefas cuja granulosidade seja superior 0.01 segundos, dependendo essencialmente do peso da comunicação.

Aplicações	Num. tarefas	Num. sincs
Fib(n=40, Gr=30)	287	145
Mmat(n=500)	dpS_SIZE	1
Pi(n=100, Gr=5000000)	n	1

Tabela 6.1: Número de tarefas e sincronizações por aplicação.

Aplicações	1 agente	sequencial	custo adicional
Fib(n=40, Gr=30)	6,17	6,16	0,16%
Mmat(n=500)	17,32	17,32	0,00%
Pi(n=100, Gr=5000000)	18,72	18,65	0,38%

Tabela 6.2: Custo adicional da execução com o di_pSystem.

A tabela 6.2 indica o custo adicional da execução de cada aplicação por 1 agente do di_pSystem relativamente à execução sequencial usual em linguagem C, sem recorrer a qualquer ambiente de programação paralela.

O custo adicional verificado é de certa forma insignificante, o que é bom, pois os *speedups* a obter serão também ganhos sobre a execução sequencial da aplicação. A figura 6.1ilustra os ganhos de velocidade obtidos em cada aplicação com a utilização do di_pSystem.

Os ganhos de velocidade foram relativamente melhores para a aplicação `Pi`. Este facto deve-se a uma granulosidade constante e à criação estática de tarefas. Na aplicação `Mmat` a granulosidade diminui com o número de agentes, visto que o número de linhas a computar com cada agente resulta da divisão do número de linhas pelo número de agentes. Sendo assim, é natural que o ganho de velocidade diminua com o aumento do número de agentes. Acresce-se ainda, o facto de com mais de quatro processadores os *threads* de execução e comunicação passam a ser concorrentes dentro de cada processador. Isto não acontece com menos de quatro, visto que cada *thread* é mapeado num processador. Este facto também se nota nos resultados obtidos na aplicação `Fib` que tem uma componente elevada de comunicação relativa à distribuição e procura de trabalho.

Consultando a figura 6.1 pode-se verificar que o comportamento das heurísticas foi praticamente idêntico. No caso geral pode-se afirmar que as heurísticas *adaptive* têm um comportamento superior. Para simplificar a leitura dos gráficos, daqui para diante

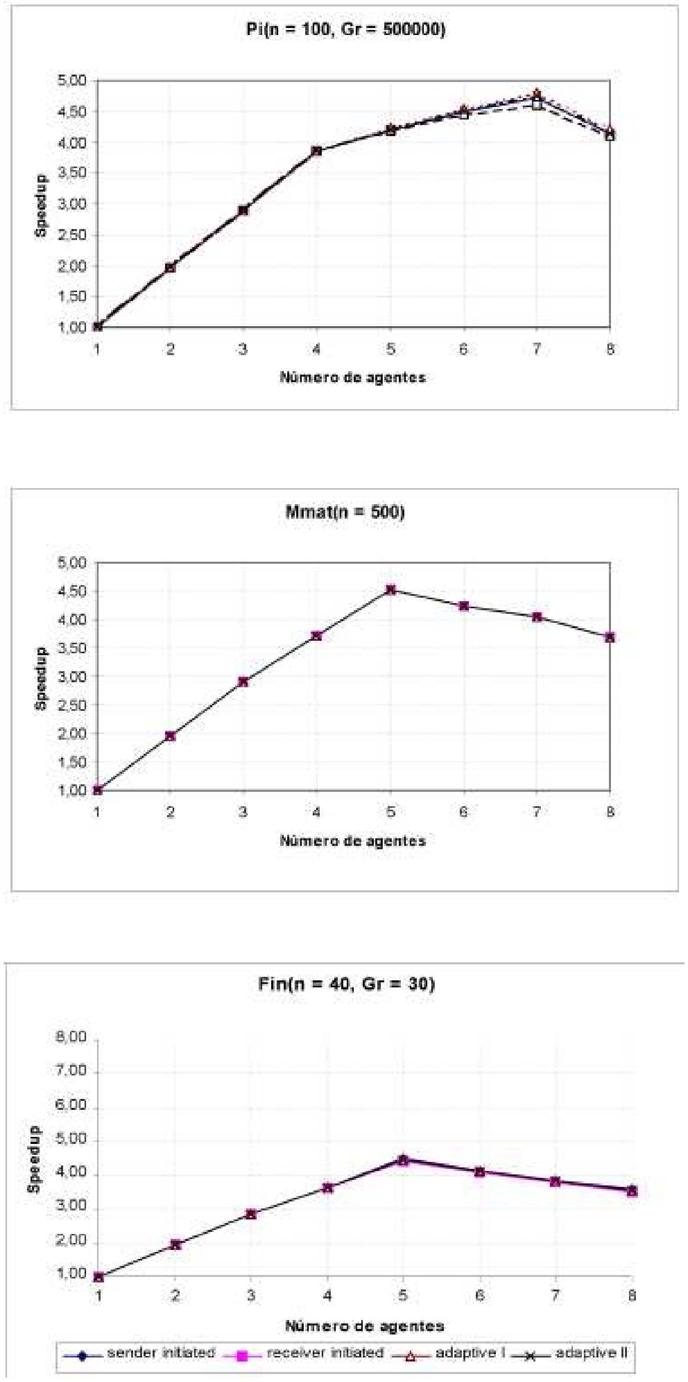
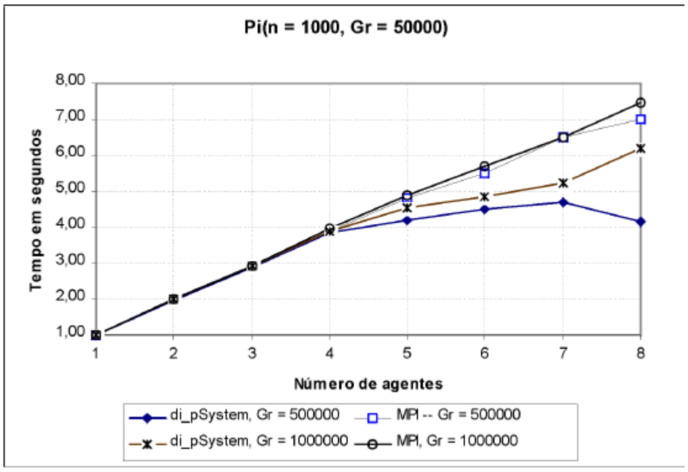


Figura 6.1: Ganhos de velocidade.

a única heurística ilustrada será *adaptive I*.

6.4 di_pSystem versus MPI

O di_pSystem pretende retirar a carga da comunicação explícita e da gestão de distribuição de trabalho do programador de arquitecturas distribuídas. Por outro lado, pretende que as suas prestações sejam o mais próximas possível das obtidas pelos sistemas que não se debruçam sobre estes problemas. Como o seu módulo de comunicação foi concebido tendo como base as funções de comunicação do MPI, é natural verificar se não houve perdas, significativas, de performance relativamente ao mesmo. A figura 6.2 indica o tempo de execução verificado pelo di_pSystem e pelo MPI no *cluster* referido acima.



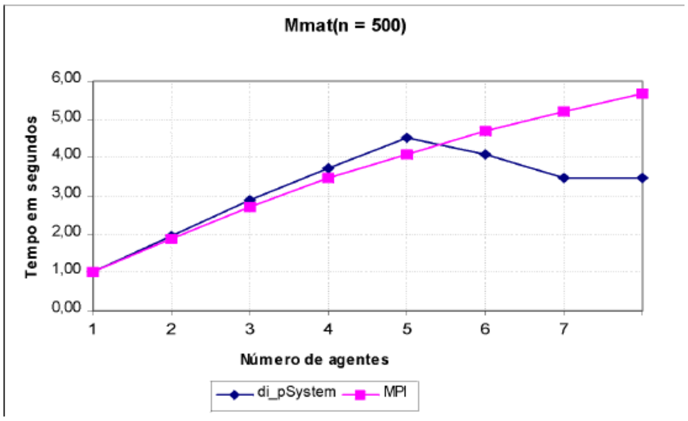


Figura 6.2: di_pSystem vs. MPI.

Os resultados obtidos demostram que o di_pSystem obtem melhores performance para menos de cinco agentes. Isto deve-se ao aproveitamento dos processadores parados para receber mensagens, sobrepondo assim comunicação com computação. Todavia, quando tal deixa de ser possível o MPI revela-se mais eficiente, pois não tem concorrência nos processadores. No entanto, esta diferença revela-se menor quando a granulosidade aumenta. A granulosidade de uma tarefa de `Pi(100, 1000000)` é de cerca de 0,13 segundos, enquanto que a granulosidade de uma tarefa de `Pi(100, 500000)` é de 0,7 segundos. O que antevê que com tarefas de granulosidade maior esta diferença passe despercebida.

6.5 A Granulosidade

A granulosidade das tarefas é um factor muito importante na execução paralela de aplicações. Tal é ainda mais importante quando a execução se efectua em arquitecturas de memória distribuída, pois o custo do envio das tarefas e dos resultados através da rede tem de ser compensado. Para realizar um estudo sobre este parâmetro de performance, foi escolhida a aplicação `PI`. O gráfico 6.3 ilustra a execução paralela da aplicação com três níveis de granulosidade, 200000, 500000 e 1000000. O tempo de execução de uma tarefa para as duas últimas execuções já foi referida, para a primeira é cerca de 0,3 segundos.

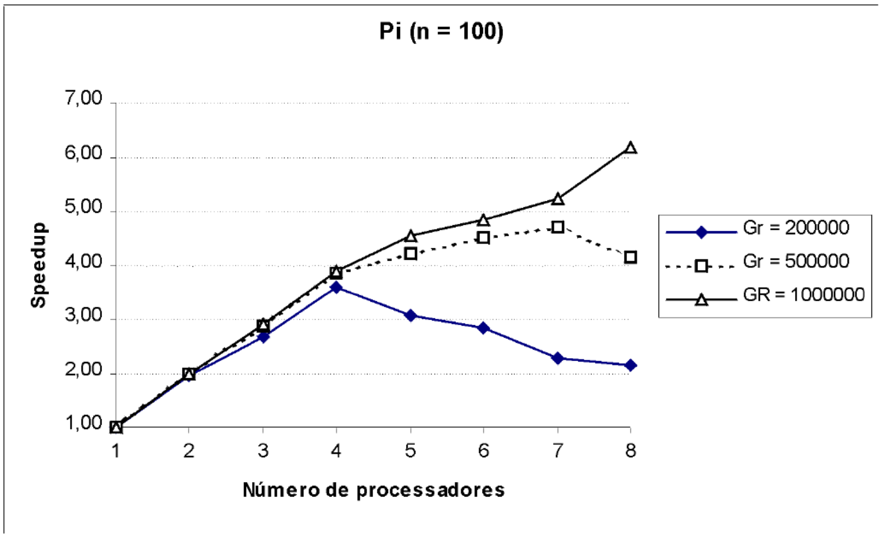


Figura 6.3: Performance vs. granulosidade das tarefas.

Pode-se verificar que com o aumento da granulosidade o ganho de velocidade é maior e a concorrência dos *threads* é menos notada.

6.6 Velocidade das Redes de Interligação

A velocidade das redes também é um factor muito importante na execução de aplicações em ambientes distribuídos. Nesta secção mostram-se resultados obtidos no *cluster* descrito anteriormente, e num novo *cluster* denominado **Dolphin 2** composto por dois *quad* Pentium Pro a 200MHz, 128 MBytes de RAM e 2,5 GBytes de disco EIDE, ligados por uma Fast Ethernet a 100MBits/segundo. O sistema operativo utilizado é o Slackware Linux/smp.

A figura 6.4 ilustra o custo do envio e recepção de uma mensagem nas redes dos dois *clusters*. Este teste é realizado enviando uma mensagem, de tamanho desejado, de um agente para outro, esperando depois que o segundo a envie de volta. O tempo cronometrado consiste no tempo passado entre o envio e a recepção da mensagem.

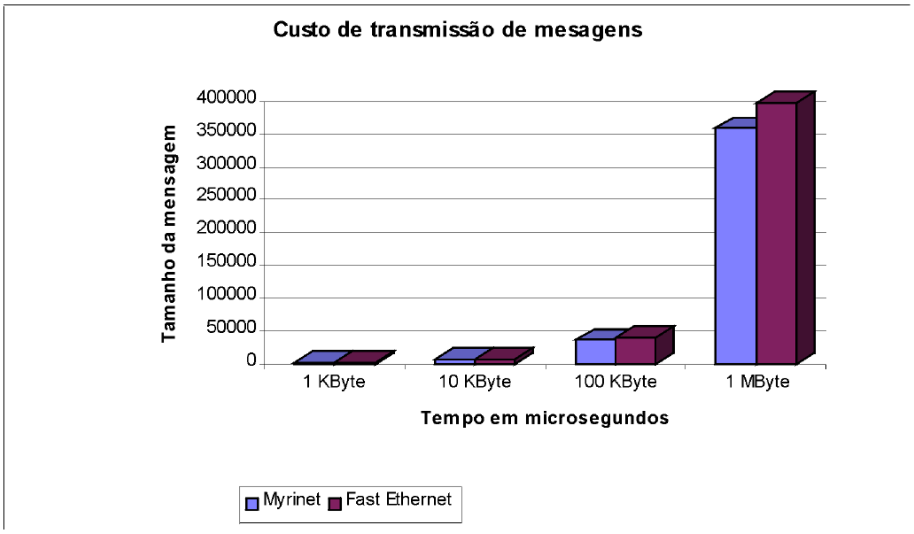
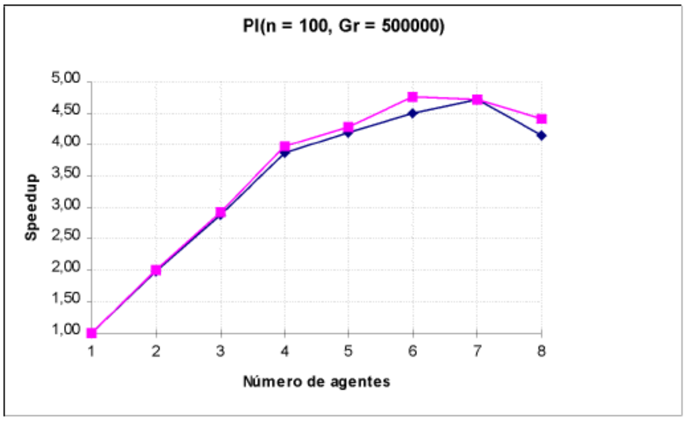


Figura 6.4: Custo de transmissão de uma mensagem.

Como se pode verificar o custo do envio e recepção de uma mensagem em ambas as redes é muito similar. Verificando-se mesmo que para mensagens de 1KByte a Fast Ethernet ser mais rápida que a Myrinet. Este facto deve-se à não optimização da Myrinet, devido à sua recente instalação. A figura 6.5 ilustra os resultados obtidos em ambos os *clusters*.

Como o custo de transmissão de uma mensagem nos dois *clusters* é semelhante, a arquitectura do Dolphin 2 é menos distribuídae e os seus processadores são mais lentos, a granulosidade das tarefas neste *cluster* é maior. Sendo assim, os resultados obtidos no Dolphin 2 tendem a ser melhores do que os obtidos no Dolphin 1. Com a optimização da rede Myrinet o panorama deverá ser diferente.



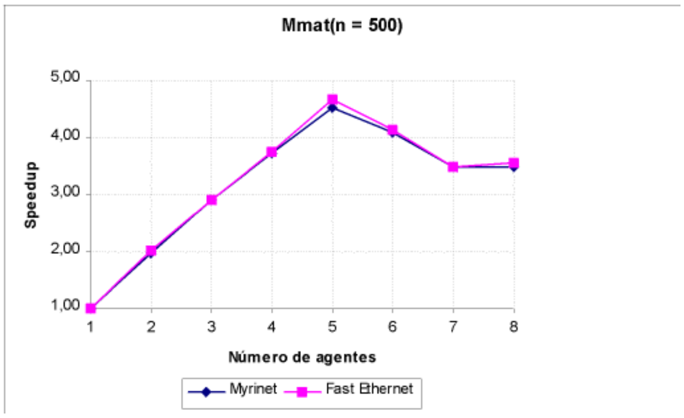


Figura 6.5: Dolphin 1 vs. Dolphin 2.

Como curiosidade mostra-se ainda uma figura 6.6 que ilustra a execução da aplicação `Pi` numa rede heterogénea de computadores. O gráfico mostra alguns pontos irregulares que se devem à diferença de velocidade de processamento de cada computador. Nesta rede encontram-se três Pentium a 200MHz, dois a 233MHz, dois a 166MHz e um a 133MHz.

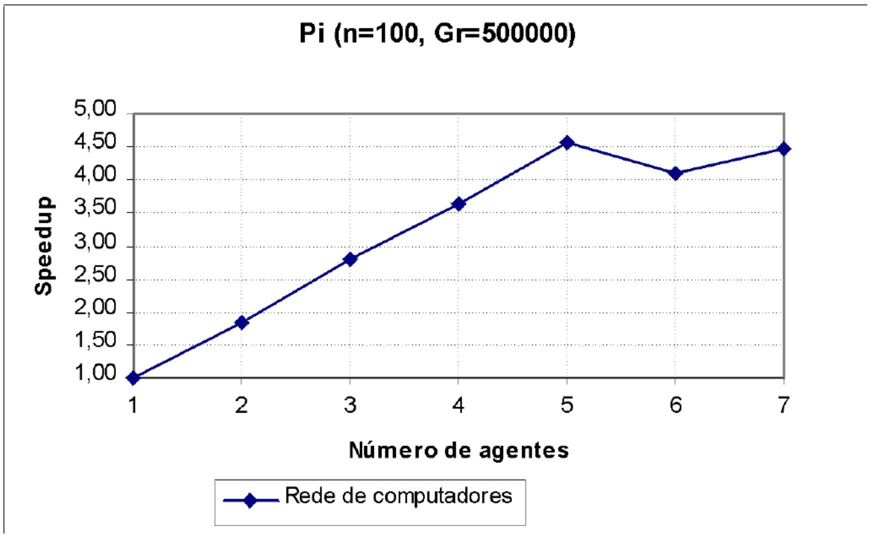


Figura 6.6: Execução numa rede heterogénea de computadores.

Capítulo 7

Conclusões

Os objectivos iniciais deste trabalho compreendiam o desenvolvimento de um ambiente de programação, o di_pSystem, para arquitecturas de memória distribuída. Tendo em vista aferirmos, de forma sumária, o trabalho realizado, começamos por relembrar os objectivos que nos propusemos atingir:

- desenvolver um ambiente que permitisse a anotação de programas, na linguagem C, para execução paralela em arquitecturas de memória distribuída.
- preservar o mais possível o modelo de programação de memória partilhada, utilizado no pSystem, escondendo a camada de comunicação do programador;
- não implementar o módulo de comunicação de raiz; recorrer a um outro sistema que proporcionasse essa funcionalidade como o MPI ou o PVM;
- incorporar a distribuição do trabalho no sistema, para possibilitar ao programador concentrar-se apenas na construção da aplicação;
- desenhar o ambiente de uma forma modular, isolando o módulo de distribuição de trabalho assim como o de comunicação, para permitir a experimentação de vários algoritmos de distribuição de trabalho e a adaptação da camada de comunicação a vários sistemas;
- realizar um estudo de performance do ambiente utilizando vários programas de teste e várias heurísticas de distribuição de trabalho.

Nas secções seguintes descreve-se o modo como se alcançaram estes objectivos, os resultados obtidos e o trabalho futuro a realizar no seguimento deste trabalho.

7.1 Os Objectivos

O di_pSystem foi desenhado, implementado e está actualmente a funcionar. Consiste num ambiente de programação paralela que permite a programação, utilizando a linguagem de programação `C`, de arquitecturas de memória distribuída. Para tal foi introduzido um pequeno conjunto de anotações que permitem expressar o paralelismo existente em cada aplicação. Desta forma, o objectivo de construir um ambiente que permita a programação deste tipo de arquitecturas foi alcançado.

A intenção de manter, o mais possível, o modelo de programação utilizado no pSystem passou pela ocultação da camada de comunicação ao programador. Para o conseguir, a comunicação entre os agentes é toda efectuada pelo sistema, recorrendo a um *thread* dedicado para o efeito. O envio e recepção de mensagens referentes a tarefas, resultados, balanceamento de carga e consistência de cópias de variáveis de leitura global, ocorre de forma transparente para o programador. Libertando-o para que se concentre exclusivamente na concepção do algoritmo a implementar.

O di_pSystem encarrega-se igualmente da distribuição e balanceamento da carga de trabalho no ambiente de execução. Este mecanismo é implementado através da introdução de um *scheduler* que se encarrega destas tarefas. Este módulo pode ser escolhido, ou mesmo implementado pelo utilizador, evitando assim a restrição da execução de aplicações ao mesmo algoritmo de distribuição de trabalho, que poderia não ser o melhor para certo tipo de aplicações. Um dos objectivos do di_pSystem era evitar a implementação das funções de comunicação de raiz, recorrendo à utilização de funções de um sistema externo standard. O sistema escolhido para o efeito foi o MPI, tendo sido utilizadas algumas das suas funções e algumas das suas potencialidades, como o envio não bloqueante de mensagens e a agregação de várias mensagens numa só.

O desenho modular do di_pSystem permite a incorporação de novas heurísticas de distribuição de trabalho e a adaptação da camada de comunicação a novos sistemas, como por exemplo ao PVM, sem para que tal seja necessário alterar qualquer outro módulo do sistema. No que se refere à distribuição de trabalho, este objectivo foi alcançado separando o módulo *scheduler* do resto do sistema. A introdução de uma nova heurística depende apenas da implementação das duas funções que compõem este módulo `Sched_Publishing_Task()` e `Sched_Management()`. No que se refere à camada de comunicação, todas as funções utilizadas na comunicação entre agentes formam um módulo do sistema (módulo de comunicação). Para a sua adaptação basta reprogramar as funções que o compõem. Depois de se alterar um destes módulos, para o integrar no sistema, basta a sua compilação e ligação ("*link") ao sistema.*

Devido à escassez de tempo não foram implementadas aplicações de grande envergadura, em termos de complexidade, para se obter um estudo de performance. No entanto, este estudo foi realizado baseado em três aplicações mais pequenas e em quatro heurísticas de distribuição de trabalho.

Apesar dos resultados obtidos serem apenas preliminares, visto ser necessário implementar mais aplicações e com maior grau de complexidade, estes mostram que o sistema é viável e que pode obter resultados bastante satisfatórios. A introdução de optimizações, nomeadamente ao nível do módulo de comunicação permitirá melhorar substancialmente a performance do sistema.

O objectivo fundamental deste projecto foi alcançado, isto é, desenhar e implementar um ambiente de programação paralela que permita a programação de arquitecturas distribuídas com um modelo de programação semelhante ao utilizado na programação de arquitecturas de memória partilhada. O estudo de performance realizado sugere que o custos adicionais relativos à gestão de carga de trabalho do sistema e à incorporação de um modelo de comunicação implícita não são significativos quando se compara com os resultados obtidos pelo MPI. Esta diferença é tanto menor quanto maior for a granulosidade das aplicações. Em conclusão, podemos considerar que o di_pSystem, além da sua vantagem ao nível da facilidade de programação, consegue ser eficiente, pelo menos a um nível muito próximo de outros ambientes, como sejam o MPI.

7.2 Trabalho Futuro

O pSystem surgiu da necessidade de se construir um sistema com uma estrutura modular que possibilitasse o estudo de heurísticas de distribuição de trabalho e o seu desenvolvimento poderá seguir a mesma filosofia, ou seja, possibilitar novos estudos. Os pontos seguintes não são projectos, são apenas as ideias que se encontram actualmente em discussão para os futuros projectos a desenvolver utilizando como base o di_pSystem:

- *Estudo que visa comparar a performance do modelo de troca de mensagens com a performance do modelo de memória partilhada em arquitecturas de memória partilhada e de memória distribuída mas virtualmente partilhada. Para tal seria necessário desenvolver uma implementação para memória virtualmente partilhada, tendo como base a de memória distribuída. Sendo apenas necessário introduzir um modulo que implemente um modelo eficaz para simular a memória* *partilhada.*

- *Implementação do modelo de troca de mensagens em C++, e estudo de problemas associados com a introdução de mecanismos como herança em sistemas distribuídos.*
- *Modificar a camada de comunicação de modo a suportar outros sistemas como, por exemplo, o PVM e variações deste, ou conjugações do PVM e do MPI como, o PVMPI.*
- *Desenvolver aplicações mais complexas no di_pSystem como forma de melhorar a robustez do sistema e ilustrar a sua aplicabilidade e eficiência.*

7.3 Fecho

Este trabalho foi para mim uma experiência enriquecedora, numa àrea que, na minha perspectiva, é uma àrea de futuro, essencialmente devido às limitações crescentes que o modelo computacional de von Neumman apresenta. Neste cenário, os ambientes de programação paralela eficientes, de fácil utilização, têm um papel de crescente importância. O di_pSystem não pretende, pelo menos por agora, um ambiente de programação paralela de uso comercial, mas, no meu entender, o conceito em que se baseia é inovador e pode ser implementado eficientemente. O meu trabalho futuro visa este mesmo objectivo: melhorar a robustez e eficiência do di_pSystem. Para o realizar pretendo implementar um conjunto de aplicações complexas, que me indiquem todos os pontos do sistema onde a eficiência está ser comprometida ou uma nova solução pode ser introduzida. A meu ver, este projecto tem potencialidade e deve ser continuado, por si só ou englobado num projecto de maiores dimensões.

Apêndice A

Aplicações de Teste Implementadas com o di_pSystem

A.1 Fibonacci – Fib(n, Gr)

```
#include  "di_pSystem.h"
#include  <stdio.h>
pSystem task <fib>( long, long? );
int Gr;

void  main( int argc, char *argv[] ){
  long  n, fibn;

  /* Ler parâmetros de entrada */
  n  = atoi(argv[1]);
  Gr = atoi(argv[1]);
  /* Iniciar di_pSystem */
  Start_di_pSystem( --argc, ++argv );
  /* Lançar tarefa fib(n) */
  new task  <fib>( n, ?fibn );
  /* Terminar di_pSystem, aguardar pelo resultado */
  End_di_pSystem();
  printf("Fibonacci(%ld) = %ld\n", n, fibn);
}

task  <fib>( long n, long ?fibn ){
```

```
  long pr[2];

  if( n < 2 )
    fibn = 1;
  else {
    if (n < Gr)
      /* Lançar tarefas fib(n-1) e fib(n-2) */
      n--;
      new task <fib>(n, ?pr[0]);
      n--;
      new task <fib>(n, ?pr[1]);
      /* Aguadar a conclusão das tarefas */
      Wait();
   } else {
      /* Executar fib(n-1) e fib(n-2) localmente */
      n--;
      fib(n, ?pr[0]);
      n--;
      fib(n, ?pr[1]);
   }
   fibn = pr[0] + pr[1];
}
```

A.2 Multiplicação de Matrizes Mmat(n)

```
#include  "di_pSystem.h"
#include  <stdio.h>
int n;
int n_lines,
double *matrix;

void read_matrix(char *);
pSystem task <multiply_columns>(int, double*?);

void  main(int argc, char *argv[]) {
  int i,j;
  int first;
```

```
  double *result_matrix;

  /* Ler matriz */
  read_matrix(argv[1]);
  /* Alocar espaco para a matriz resultado */
  result_matrix = (double *)malloc(sizeof(double)*n*n);
  /* Iniciar o di_pSystem */
  start_di_pSystem(argc,argv);
  /* Calcular o numero de linhas a executar por cada tarefa */
  n_lines = n/pS_size;
  /* Lancar tarefas */
  for (i=0; i<pS_size; i++) {
    first = n_lines*i;
    new task <multiply_columns>(first, result_matrix+first*n, n_lines*n);
  }
  /* Computar o resto das linhas localmente */
  first = first+n_lines;
  n_lines = n-first;
  multiply_columns(first, ?(result_matrix+first*n));
  /* Terminar di_pSystem */
  End_di_pSystem();

  /* Imprimir resultado */
  printf("\nResult matrix:\n");
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      printf ("%lf ", *result_matrix);
      result_matrix++;
    }
    printf("\n");
  }

}

void read_matrix(char *file) {
  FILE *fp;
  int i,j;

  /* Abrir ficheiro */
  fp = fopen(file, "r");
```

```
  /* Ler o numero de linhas */
  fscanf(fp,"%lf",&n);
  /* Inicializar a matriz */
  matrix = (double *)malloc(sizeof(double)*n*n);
  /* Ler a matriz */
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      fscanf(fp,"%lf",(matrix+i*n+j));

}

task <multiply_columns>(int first, double *result) {
  int i,j,k;

  /* Ler o valor de n_lines */
  Read(n_lines);
  /* Multiplicar as linhas das matriz */
  for (i=first; i<first+n_lines; i++) {
    for (j=0; j<n; j++) {
      *result = 0;
      for (k=0; k<n; k++)
        *result = *result + *(matrix+i*n+k)**(matrix+k*n+j);
      result++;

    }
  }
}
```

A.3 Cálculo de Pi - Pi(n, Gr)

```
#include  <math.h>
#include  <stdio.h>
#include  "di_pSystem.h"
#define   MAXSLICES       1024
#define   LOWER           ((double)0.0)
#define   UPPER           ((double)1.0)
#define   FUNCTION(X)     ( 1.0 / (1.0 + (X)*(X)) )
```

```
typedef struct{
   double   lower;      /* lower and upper limits */
   double   upper;      /*  of partition element  */
}Work;
int subslices;
pSystem task  <trapezium>( Work, double? );

void  main( int argc, char *argv[] ){
 register int   i, nslices;
 register double   x, y, h, Pi = 0;
 double   width = UPPER - LOWER;
 double   values[MAXSLICES];
 Work     work;

 nslices = atoi( argv[1] );
 subslices = atoi( argv[2] );
 h = width/(double)nslices;
 x = LOWER;
 y = x + h;
 Start_di_pSystem( argc, argv );
 for( i = 0 ; i < nslices ; i++ ){
    work.lower = x;
    work.upper = y;
    x = y; y += h;
    new task  <trapezium>( work, ?values[i] );
 }
 Wait();
 for( i = 0 ; i < nslices ; i++ )
    Pi += values[i];
 Pi *= 4.0;
 printf("pi=%lf\n",Pi);
 End_di_pSystem();
}

task  <trapezium>( Work work, double ?sum ){
  register double   h, x, y;

  h   = ( work.upper - work.lower )/((double)work.subslices);
  sum = ( FUNCTION( work.lower ) + FUNCTION( work.upper ) )/2.0;
```

```
  x   = work.lower + h;
  for(  ; x < work.upper ; x += h )
     sum += FUNCTION( x );
  sum *= h;
}
```

Apêndice B

Aplicações de Teste Implementadas com o MPI

B.1 Multiplicação de Matrizes Mmat(n)

```
#include <stdio.h>
#include <mpi.h>
#include "time.h"
#define RESULT 0

void read_matrix(char *);
int n;
double *matrix;

main (int argc, char *argv[]) {

  int i, j, k,           /* Inducao */
  int first, n_lines;    /* Parte da matriz a computar por uma tarefa */
  int size, myrank;      /* Identificacão do agente e numero de
  agentes */
  double *result_matrix; /* Matriz resultado */
  double *array;         /* Vector resultado a enviar ao agente mestre */
  double *pos;           /* Variavel auxiliar de calculo */
  MPI_Status status;     /* Status retornado pelo MPI */
```

```
/* Iniciar MPI */
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
/* Ler matriz */
read_matrix(argv[1]);
/* Calcular o numero de linhas a computar por uma tarefa */
n_lines = n/size;
/* Iniciar matriz resultado */
result_matrix = (double *)malloc(sizeof(double)*n*n);
if (myrank) {
  /* Agente escravo */
  if (n_lines != n && n_lines > 0) {
    /* Calcular parte da matriz resultado */
    array = (int *)malloc(sizeof(int)*n_lines*n);
    first = n_lines*(myrank-1);
    for (i=first; i<first+n_lines; i++) {
      for (j=0; j<n; j++) {
        *array = 0;
        for (k=0; k<n; k++)
          *array = *array + *(matrix+i*n+k) * *(matrix+k*n+j);
        array++;
      }
    }
    array = array-n_lines*n;
    /* Enviar resultado */
    MPI_Send(array, n_lines*n, MPI_DOUBLE, 0, RESULT, MPI_COMM_WORLD);
  }
  /* Terminar MPI */
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
} else {
  /* Agente mestre */
  /* Calcular parte da matriz resultado */
  first = n_lines*(size-1);
  for (i=first; i<n; i++) {
    for (j=0; j<n; j++) {
      pos = result_matrix+i*n+j;
      *pos = 0;
      for (k=0; k<n; k++)
```

```
          *pos = *pos + *(matrix+i*n+k) * *(matrix+k*n+j);
      }
    }
    /* Receber a computacao dos outros agentes */
    if (n_lines != n && n_lines > 0) {
      array = (int *)malloc(sizeof(int)*n_lines*n);
      for (i=1; i<size; i++) {
        MPI_Recv(array, n_lines*n, MPI_DOUBLE, MPI_ANY_SOURCE,RESULT, //
                 MPI_COMM_WORLD, &status);
        (void)memcpy((void *)(result_matrix+n_lines*n*(status.MPI_SOURCE-1)), //
                     (void *)array, n_lines*n*sizeof(double));
      }
    }
    /* Terminar MPI */
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    /* Imprimir resultado */
    printf("\nResult matrix:\n");
    for (i=0; i<n; i++) {
      for (j=0; j<n; j++) {
        printf ("%lf ", *result_matrix);
        result_matrix++;
      }
      printf("\n");
    }
  }
}

void read_matrix(char *file) {
  FILE *fp;
  int i,j;

  /* Abrir ficheiro */
  fp = fopen(file, "r");
  /* Ler numero de linhas */
  fscanf(fp,"%lf",&n);
  /* Inicializar matriz */
  matrix = (int *)malloc(sizeof(int)*n*n);
  /* Ler matriz */
  for (i=0; i<n; i++)
```

```
    for (j=0; j<n; j++)
      fscanf(fp,"%lf",(matrix+i*n+j));

}
```

B.2 Cálculo de Pi - Pi(n, Gr)

```
#include <stdio.h>
#include <mpi.h>
#include  <math.h>
#include "time.h"
#define   MAXSLICES       1024
#define   LOWER           ((double)0.0)
#define   UPPER           ((double)1.0)
#define   FUNCTION(X)     ( 1.0 / (1.0 + (X)*(X)) )
#define RESULT 0

double trapezium (double);
double h, subslices;

void  main(int argc, char *argv[]) {

  int   i, slices;
  double   x, y;
  double   width = UPPER - LOWER;
  double   value, Pi = 0;
  int myrank, size, flag;
  MPI_Status status;

  /* Ler argumentos de entrada */
  slices = atoi( argv[1] );
  subslices = atoi( argv[2] );
  /* Calcular tamanho do intervalo */
  h = width/(double)slices;
  /* Iniciar MPI */
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

```
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if (myrank) {
    /* Agente escravo */
    /* Calcular parte do integral */
    for (i = myrank; i<slices; i+= size) {
      Pi += trapezium(LOWER + i*h);
      MPI_Send(&Pi, 1, MPI_DOUBLE, 0, RESULT, MPI_COMM_WORLD);
      Pi = 0;
    }
    /* Terminar MPI */
    MPI_Finalize();
  } else {
    /* Agente mestre */
    /* Determinar intervalo */
    i = 0;
    x = LOWER;
    y = 0;
    while (i<slices) {
      /* Determinar se existem mensagens */
      MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);
      if (flag) {
        /* Receber mensagem */
        MPI_Recv(&value, 1, MPI_DOUBLE, MPI_ANY_SOURCE, RESULT, //
                  MPI_COMM_WORLD, &status);
        Pi += value;
        i++;
      }
      /* Calcular parte do integral */
      if (x < UPPER) {
Pi += trapezium(x);
y = y + size;
i++;
x += size*h;
      }
    }
    Pi *= 4.0;
    /* Terminar MPI */
    MPI_Finalize();
    /* Imprimir resultado */
    printf("pi=%lf\n",Pi);
```

```
  }
}


double trapezium(double x) {
  register double sum, new_h, y;

  /* Calcular integral de um intervalo */
  y = x + h;
  new_h   = h /((double)subslices);
  sum = ( FUNCTION( x ) + FUNCTION( y ) )/2.0;
  for(  x = x+new_h ; x < y ; x += new_h )
    sum += FUNCTION( x );
  sum *= new_h;
  return(sum);
}
```

Referências

[ABC95] *A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie and D. Yeung. The MIT Alewife Machine: Architecture and Performance. ISCA 1995*

[ABL95] *J. Árabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey and P. Stephan. Dome: Parallel Programming in a Heterogenous Multi-User Environment. School of Computer Science, Carnegie Mellon University, April 1995*

[ABM94] *R. Alasdair, A. Bruce, J. G. Mills and A. Gordon Smith S.. Chimp/MPI User Guide. University of Edinburg, June 1994*

[BC93] *R. Black and S. Crosby. University of Cambridge, Computer Laboratory, 1993*

[Che93] *D. Y. Cheng. A Survey of Parallel Programming Languages and Tools, Report RND-93-005, March, 1993*

[KDCZ94] *P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In* Proceedings of the Winter 94 Usenix Conference*, pages 115–131, January 1994.*

[BCF+95] *Nanette J. Boden, Danny Cohen, Roberte ER. Feiderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Scizovic, and Wen-King Su. Myrinet - A Gigabit-per-Second Local-Area Network.* IEEE-Micro*, 15(1):29–36, February 1995.*

[BDV] *G. Burns, R. Daoud and J. Vaigl. LAM: Na Open Cluster Environment for MPI. Ohio Supercomputer Center, Columbus, Ohio*

[Blu95] *C. Blumofe. Cilk: An Efficient Multithreaded Runtime System. In ACM, editor, Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Santa Barbara, California, July 1995, MIT Laboratory for Computer Science.*

[CCK93] *Peter Carlin, Mani Chandy, Carl Kesselman - The Compositional C++ Language Definition, California Institute of Technology, 1993*

[CDG] *D. E. Culler, A. Dusseau, S C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken and K. Yelick. Parallel Programming in Split-C. Computer Science Division, University of California, Berkeley.*

[CK] *K. Mani Chandy, Carl Kesselman - Compisitional C++: Compositional Parallel Programing, California Institute of Technology.*

[CK88] *T. Casavan and J. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. IEEE Transactions on Software Engeneering, Ferbruary 1988*

[DSW95] *Jack Dongarra, Steve Otto, Marc Snir and David Walker. An Introduction to the MPI Standart. 1995.*

[ECG92] *T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In International Symposium on Computer Architecture, 1992.*

[Fly72] *M. Flynn. Some Computer Organizations and their Effectiveness. IEEE Transactions on Computers. Vol. C-21. No 9. September 1972*

[FS94] *A. Ferrari and V. Sunderam. TPVM: A Threads-Based Interface and Subsystem for PVM. Technical Report CSTR-940802, Emory University Departement of Mathematics and Computer Science, Atlanta, GA, August, 1994*

[Geh94] *Narain Gehani, Ada: Concurrent Programming, Prentice-Hall, Englewood Cliffs, NJ, 1994.*

[GB94a] *A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderan. PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. MIT Press, 1994.*

[GB94b] *A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderan. PVM 3 user's guide and reference manual. Technical Report ORNL, September 1994.*

[GL96] *W. Gropp and E. Lusk. User's Guide for MPICH, a Portable Implementation of MPI, ANL/University of Chicago*

[GR86] *N. H. Gehani and W. D. Roome. Concurrent C. In Software-Practice and Experience Vol. 16(9), 821-844, September 1986*

[GR88] *N. H. Gehani and W. D. Roome. Concurrent C++. In Software-Practice and Experience Vol. 18(12), 1157-1177, December 1988*

[Gr90] *N. H. Gehani and W. D. Roome. Concurrent C - A language for Programming Multiprocessor Systems. In Byte December 1990.*

[Gri93] *A. Grimshaw. Easy-to-Use Object-Oriented Parallel Processing with Mentat.* IEEE Computer*, 26(5):39–51, May 1993.*

[HB85] *K. Hwang and F. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill, 1985*

[KK93] *L. V. Kale and S. Krishnam. Charm++: A Portable Concurrent Object Oriented System based on C++. Technical report, University of Illinois, Urbana-Champaign, February 1993.*

[KOH94] *J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, J. Hennessy. The Stanford FLASH Multiprocessor. In* Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, April 1994*

[KRS] *L .V. Kalé, B. Rambumar, A. B. Sinha and A. Güsoy. The Charm Parallel Programming Language and System: Part I - Description of Language Features, Dept. of Computer Science, University of Illinois and Dept. of Electrical and Computer Engineering, University of Iowa.*

[Lop95] *Luís Lopes: Um Ambiente de Programação Paralela para Avaliação de Heurísticas de Scheduling, Provas de Aptidão Pedagógica e Capacidade Científica, Universidade do Porto, Janeiro, 1995.*

[LM92] *Thomas J. LeBlanc and Evangelos P. Markatos. Shared Memory vs. Message Passing in Shared-Memory Multiprocessors. Computer Science Department, University of Rochester, 1992.*

[LP89] *K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, ACM Transactions on Computer Systems, 1989, 7, 4, 321-359, "November"*

[LR91] *M. Lam and M. Rinard: Coarse-Grain Parallel Programming in Jade. ACM SIGPLAN Notices, July, 1991*

[LS97] *L. Lopes and F. Silva. Thread- and Process-Based Implementations of the pSystem Parallel Programming Enviroment. Software Practice and Experience, 27, B, 329 – 351, March, 1997*

[M94] *The ADAM Architecture and it's simulation. O. Maquelin, TIK-Schriftenreihe Nr. 4, 1994, IBSN 3 7281 21113 4*

[ML92] *E. P. Markatos and T. J. LeBlanc. Shared-memory multiprocessor trends and the implications for parallel program performance. Technical Report 420, University of Rochester, Computer Science Department, March 1992.*

[MPI95] *MPI: A Message-Passing Interface Standard. Message Passing Interface Forum. June 12, 1995*

[MPI2-97] *MPI-2: Extensions to the Message-Passing Interface. Message Passing Interface Forum. July 18, 1997*

[Nev96] *N. Nevin. The Performance of Lam6.0 and MPICH 1.0.12 on a Workstation Cluster. Ohio Supercomputer Center Tech. Report, Mars 1996.*

[NB92] *P. Newton and J. C. Browne, The Code 2.0 Graphical Parallel Programming Language, Proc. ACM Int. Conf. On Supercomputing, July, 1992*

[Par94] *Craig Partridge.* Gigabit Networking. *Addison-Wesley Professional Computing Series, 1994.*

[RSC97] *Ricardo Rocha, Fernando Silva and Vítor Santos Costa, YapOr: an Or-Parallel Prolog System based on Environment Copying, Technical Report DCC-97-14, DCC & LIACC, Universidade do Porto, 1997.*

[RSS] *B. Rambumar, A. B. Sinha, V. A. Saletore and L .V. Kalé. The Charm Parallel Programming Language and System: Part I - Description of Language Features, Dept. of Computer Science, University of Illinois, Dept. of Computer Science, Oregon State University and Dept. of Electrical and Computer Engineering, University of Iowa.*

[S95] *F. Silva, Dolphin: Ambiente de Computação Paralela para Sistemas de Alto-Desempenho, projecto PRAXIS XXI (contracto PBIC/C/TIT/2495/95).*

[Sap] *W. Saphir. Sorting Out Communication Libraries: A comparison of NX, CMMD, PVM and MPI. NAS Scientific Computing Center, NASA Ames Research Center, Computer Sciences Corporation.*

[SL94]

[T94-6] *Concurrent Programming with TreadMarksTM. ParallelTools, L.L.C., 1994-96.*

[Vok95] *Rajeev Vokkarne. Distributed Execution Environments for the Code 2.0 Parallel Programming System. The University of Texas at Austin, 1995*

[ZG95] *H. Zhou and A. Geist. LPVM: A Step Towards Multithread PVM. Mathematical Sciences Section, Oak Ridge National Laboratory, Oask Ridge, 1995*