di_pSystem: Um Ambiente de Programação Paralela para Arquitecturas de Memória Distribuída

Hervé Paulino   Fernando Silva   Luís Lopes

Technical Report Series: DCC-98-8

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal
Tel: +351+2+6001672 – Fax: +351+2+6003654
http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

di_pSystem: Um Ambiente de Programação Paralela para Arquitecturas de Memória Distribuída *

Hervé Paulino Fernando Silva Luís Lopes

DCC-FC & LIACC, Universidade do Porto
Rua do Campo Alegre, 823 4150 Porto, Portugal
herve, fds, lblopes@ncc.up.pt

November 1998

Resumo: *As arquitecturas paralelas de memória distribuída são, cada vez mais, utilizadas para a execução de programas com componentes sujeitas a execução paralela. No entanto, grande parte dos ambientes de programação disponíveis para este tipo de arquitecturas são de difícil programação, exigindo do programador um conhecimento profundo da arquitectura alvo, e obrigando-o a incorporar no seu algoritmo comunicação explícita, distribuição de trabalho e balanceamento de carga. Neste artigo apresenta-se um ambiente de programação para arquitecturas de memória distribuída, denominado* **di_pSystem**, *que contraponha estes modelos. Deste modo, no di_pSystem toda a comunicação, distribuição de trabalho e balanceamento de carga são geridas automaticamente pelo sistema. No entanto, o utilizador pode escolher a sua heurística de distribuição de trabalho, ou mesmo implementar a que deseja, visto que o sistema foi desenhado de forma a que seja fácil a incorporação de novas heurísticas.*

1 Introdução

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 seja a Myrinet [1]. Todavia, o sucesso deste tipo de arquitecturas está condicionado aos ambientes de programação que proporcionam. Ambientes de programação paralela, como o PVM [2] ou o MPI [3] 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 [4] que forneça memória virtualmente partilhada, permitindo um modelo de programação mais próximo do de memória partilhada.

Pensamos que a utilidade de um ambiente de programação paralela está relacionada, por um lado, com a sua facilidade de utilização. Por outro lado, não menos importante, está a eficiência com que permite a execução de programas em paralelo. Nesta linha, Lopes e Silva [5] desenvolveram um ambiente de programação paralela para arquitecturas de memória partilhada, **pSystem**, que permite a anotação de programas C para execução em paralelo. A vantagem do pSystem reside na sua portabilidade e eficiência de execução de programas, mas sobretudo no ambiente de execução

*Este trabalho foi suportado parcialmente pela Fundação da Ciência e Tecnologia através do projecto Dolphin (PRAXIS/2/2.1/TIT/1577/95).

que proporciona. Este é responsável pela criação dos agentes de execução paralela, pela distribuição dinâmica de trabalho (*scheduling*) e pelo balanceamento de carga nos vários agentes. A modularidade do sistema permite ainda que seja relativamente simples para um utilizador desenvolver e optimizar a sua heurística de distribuição de trabalho.

Neste artigo, apresentamos uma nova arquitectura para um ambiente de programação paralela para arquitecturas de memória distribuída, o **di_pSystem**. Este ambiente é uma evolução do pSystem e tem como um dos objectivos preservar, ao máximo, a facilidade de programação do pSystem, assim como a sua modularidade em termos de arquitectura. Deste modo, cabe ao di_pSystem realizar a distribuição e balanceamento dinâmico de todo o trabalho paralelo gerado durante a execução paralela. Na mesma linha toda a comunicação inerente à execução de um programa numa arquitectura de memória distribuída fica escondida do programador e é gerida automaticamente, quando necessário, pelo sistema. Com este modelo de execução o programador pode concentrar todos os seus esforços na elaboração do algoritmo, indicando apenas as tarefas candidatas a execução paralela, abstraindo-se portanto do facto de o seu programa executar num ambiente distribuído.

A comunicação no di_pSystem utiliza a interface paralela MPI para suportar a computação distribuída, sendo o módulo que a contém o único que depende do sistema utilizado. No entanto, a sua adaptação a outros sistemas é relativamente simples.

As secções incluídas neste artigo apresenta-se p modelo geral de execução do di_pSystem, a estrutura de um agente e o conceito de variável de leitura global. Indroduz-se ainda o conjunto de anotações disponibilizadas para a anotação de paralelismo, denominado interface de programação, detalhes relativos à comunicação entre agentes, e um pequeno estudo de performance.Por fim apresentam-se as conclusões relativas ao trabalho efectuado.

2 Modelo Geral de Execução

O di_pSystem permite a anotação, através de um conjunto de primitivas, de paralelismo existente em aplicações escritas na linguagem C. O seu modelo computacional é composto por um conjunto de agentes de cuja interacção e cooperação depende a evolução da execução. O utilizador é responsável pela definição da topologia de execução, mapeando agentes em máquinas ou processadores. O modelo de execução do di_pSystem é SPMD onde cada cópia do programa dá origem a um agente. Estes dividem-se em agente principal e agentes secundários, sendo o agente principal responsável pela execução da componente sequencial do programa e portanto responsável pelo lançamento do trabalho paralelo. É ainda este que centraliza a terminação do sistema, podendo depois prosseguir com a execução do resto do programa. Os restantes agentes têm como funcionalidade receber, executar e, possivelmente, criar e distribuir trabalho paralelo.

O trabalho paralelo do di_pSystem consiste em funções de C anotadas pelo programador como tarefas susceptíveis de execução paralela. A tarefa é a unidade elementar de execução, tendo o programador controle absoluto sobre a sua granulosidade, que depende apenas da complexidade temporal do corpo da função anotada. Cada tarefa é executada segundo uma árvore de execução em que cada nó representa uma tarefa e cada caminho uma dependência de dados. À medida que são criadas novas tarefas são adicionadas novas sub-árvores ligadas à tarefa que as criou. Para ilustrar este conceito segue-se um exemplo simples referente à execução da implementação da recorrência de Fibonacci.

Uma árvore de execução pode ser distribuída pelos vários agentes, ou seja, cada nó pode ser executado por agentes distintos.

O di_pSystem possibilita a execução assíncrona ou síncrona de uma tarefa. Por defeito a execução é assíncrona, mas, no entanto, o programador pode, explicitamente através de uma anotação, suspender a execução de uma tarefa até que toda a computação abaixo do seu nó esteja concluída. Este mecanismo permite a gestão das dependências de dados. Quando uma tarefa suspende, o agente responsável pela sua execução pode executar outras tarefas, retomando a primeira assim que esta esteja pronta a executar.

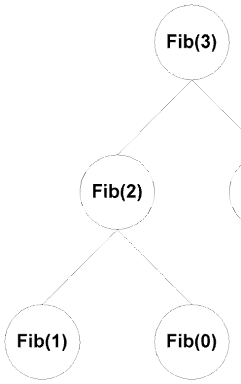

Figure 1: Árvore de execução de Fibonacci de 3.

3 Estrutura de um Agente

Cada agente é composto por um **thread de execução** e um **thread de comunicação**. A progressão da computação em cada agente depende da interacção destes threads com três módulos de funcionalidade: o **scheduler**, o **gestor de funções e variáveis** e o **módulo de comunicação**. A figura 2 ilustra as possíveis interacções entre os *threads* e os módulos.

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.

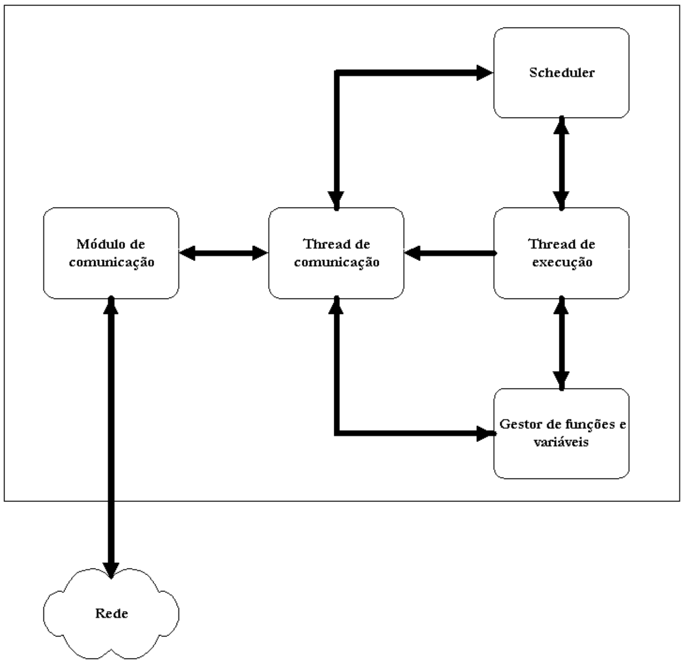

Figure 2: Estrutura de um agente.

O **thread de comunicação** é a entidade responsável por toda a comunicação entre um agente e os demais. Consequentemente, é a única componente a ter acesso ao **módulo de comunicação**, responsável pela comunicação entre agentes. Entre as competências do *thread* de comunicação contam-se a colocação, nos endereços respectivos, dos resultados das tarefas que foram executadas remotamente, o envio dos resultados das tarefas cuja colocação é da responsabilidade de outro agente e o envio e recepção de informação relativa à evolução da execução no sistema.

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, torna-se 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 **scheduler** é o responsável pela implementação da heurística de distribuição de trabalho utilizada por um agente. A distribuição de trabalho é descentralizada, sendo cada agente responsável pela atribuição das tarefas que cria. Deste modo o *scheduler*, através de informação relativa ao trabalho local e global, decide para onde deve enviar ou pedir trabalho.

O **módulo de comunicação** contém todas as funções de comunicação entre agentes. A sua implementação foi realizada tendo como base a norma standard de comunicação MPI. A sua separação do *thread* de comunicação visa uma modularidade que permita a introdução no sistema de novos módulos de comunicação, baseados em outros sistemas ou elaborados de raiz, sem que tal afecte o resto do sistema. A mesma filosofia foi aplicada ao *scheduler* possibilitando ao utilizador a escolha de uma heurística de distribuição de trabalho de um leque de heurísticas disponibilizadas, ou, se preferir, implementar a sua própria heurística. A implementação de uma nova heurística e de um novo módulo de comunicação depende apenas da programação das funções que compõem os respectivos módulos e da ligação (*link*) do ficheiro que as contém ao resto do sistema. No caso da implementação de heurísticas, esta é simplificada pela disponibilização de um conjunto de estruturas de dados e de variáveis que contêm informação relevante para a distribuição de trabalho.

Assim como no pSystem, o facto de se poderem incorporar novas heurísticas de distribuição de trabalho no di_pSystem torna-o um utilitário ideal para a realização de estudos sobre a performance de algoritmos de distribuição de trabalho.

4 Variáveis de Leitura Global

Num modelo de programação distribuído não existe um espaço de endereçamento global. Deste modo o envio e recepção de informação de uma tarefa tem de ser efectuada através dos seus argumentos. Este facto torna-se incomodo quando se pretende ler, na tarefa, um conjunto de valores. Apesar de a implementação actual não pretender oferecer um suporte completo, via software, para memória distribuída virtualmente partilhada, o di_pSystem permite a leitura de variáveis globais nas suas tarefas. Deste modo, os valores destas variáveis não têm de ser agregados aos argumentos de entrada das tarefas.

Antes do di_pSystem ser iniciado o conteúdo de cada variável é igual em todos os agentes. No entanto, uma variável de leitura global só pode alterada na componente sequencial do programa, ou seja, no agente principal. Para que os restantes agentes sejam notificados da atribuição foi necessário implementar um mecanismo que propague as atribuições, o que levanta o problema da coerência das cópias das variáveis. Para o resolver, o di_pSystem implementa dois protocolos: a *invalidação* e a *actualização* do conteúdo das variáveis [6]. Para implementar o protocolo de invalidação foi implementada em cada agente uma tabela associativa que retém a informação sobre quais as variáveis cujo conteúdo é válido ou não.

Com o conceito de variável de leitura global o programador poupa em comunicação, pois um agente só efectua comunicação quando existe uma atribuição à variável, em contraposição com o aumento do tamanho da mensagem necessária para enviar o valor da variável nos argumentos e a necessidade de enviar esta informação em cada tarefa, provavelmente para o mesmo agente.

5 A Interface de Programação

A interface de programação do di_pSystem consiste num conjunto de anotações que permitem ao utilizador anotar funções C como tarefas candidatas a execução paralela e gerir as variáveis de leitura global.

di_pSystem_task ¡função¿ ([tipo1][,tipo2?]) declara ao sistema que a função **função** pode ser executada em paralelo e qual o tipo dos seus argumentos.

di_pSystem_var tipo variável declara uma variável como sendo de leitura global.

Start_di_pSystem(int, char **) inicia um agente no nó que está a executar o programa. O agente pode ser configurado através dos argumentos de entrada desta função.

End_di_pSystem() termina o di_pSystem aguardando que toda a computação de trabalho paralelo termine.

new task ¡função¿ ([arg1][,?arg2]) constrói uma tarefa correspondente à função **função**. Existem variantes para a passagem, como argumentos, de vectores de elementos.

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.

Read(&variável) garante que o valor da variável de leitura global **variável** está correcto.

Como exemplo de um programa implementado com o di_pSystem, encontra-se em anexo uma implementação paralela do problema de multiplicação de matrizes.

6 A 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. Deste modo o di_pSystem beneficia 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.

No di_pSystem existem dois tipos de mensagens: as simples que permitem o envio de, por exemplo, uma 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 são implementadas por um simples envio, enquanto as compostas necessitam de operações de empacotamento e desempacotamento.

Para se evitar *deadlocks* o envio de qualquer mensagem é precedido pela detecção de recepção de mensagens. Desta forma, obriga-se que todas as mensagens em **buffer** sejam lidas antes de se efectuar um envio. Pois, apesar de o di_pSystem usar a função não bloqueante do `MPI: 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 re-escrita, 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 agregação de varias mensagens numa só pode ser muito importante para minimizar o número de mensagens na rede. No di_pSystem sempre que se envia uma mensagem agrega-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*. Isto permite manter informação mais precisa sobre a distribuição de trabalho no sistema num dado momento, sem penalização de comunicação.

7 Performance

O di_pSystem foi já testado num pequeno conjunto de programas em arquitecturas paralelas de baixo custo, nomeadamente num *cluster* de 4 dual-pentium II a 300MHz interligados por uma rede Myrinet. Foram ainda realizadas experiências numa rede Fast-Ethernet de 6 Pentium e 2 Quad-PentiumPro a 200 MHz. Estes resultados são preliminares, estando actualmente a ser implementadas aplicações de maior complexidade para fundamentar a viabilidade do sistema e melhorar ainda mais as suas performances. Os resultados apresentados referem-se a uma aplicação, denominada **Pi**, que calcula o número (π) através da relação

$$\frac{\pi}{4} = \int_0^1 \frac{1}{1+x^2} \qquad (1)$$

Para este estudo foram utilizadas duas heurísticas de distribuição de trabalho estáticas: **Sender initiated** e **Receiver initiated**, e duas adaptativas [7]: **Adaptive I** e **Adaptive II** que, consoante o estado do agente, se adaptam alternando entre Sender e Receiver initiated. A figura 3 ilustra os ganhos de velocidade obtidos para a aplicação, onde n representa o número de partições do integral, assim como o número de tarefas criadas. Neste exemplo cada tarefa tem um tempo de execução na ordem dos 0.07 segundos.

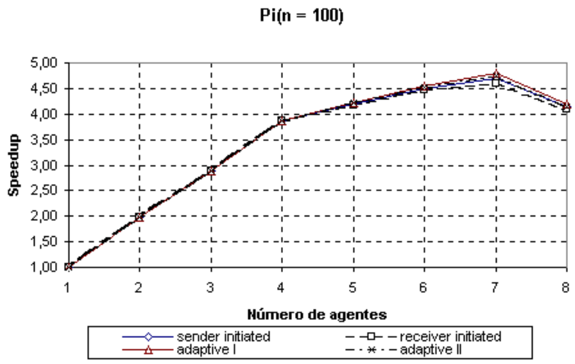

Figure 3: Ganhos de velocidade.

Os ganhos são lineares até quatro agentes e consideráveis até sete. No entanto, com o aumento do número de agentes aumenta a comunicação e com tarefas de granulosidade pequena e constante é normal que os ganhos se venham a ressentir com o aumento do número de agentes.

A figura 4 ilustra a comparação entre o di_pSystem e o MPI, onde o parâmetro Gr representa o tempo de execução, em segundos, das tarefas. Neste exemplo é apenas ilustrada a heurística Adaptive I.

Podemos verificar que à medida que a granulosidade das tarefas aumenta o fosso entre o di_pSystem e o MPI diminui. Este facto antevê que em aplicações com tarefas de maior granulosidade este fosso passe despercebido diante da facilidade de programação que o sistema oferece.

8 Conclusões

Neste artigo descrevemos o modelo execução e os conceitos que suportam o ambiente de programação paralela para arquitecturas de memória distribuída, di_pSystem. O ambiente visa simplificar a tarefa

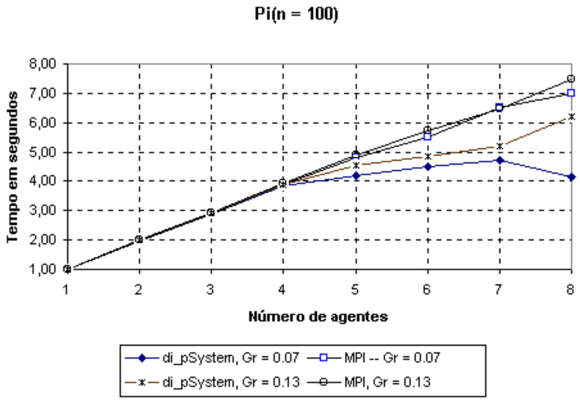

Figure 4: di_pSystem.

do programador de aplicações paralelas, para o que integra toda a comunicação inerente à execução de programas em arquitecturas de memória distribuída e funcionalidades como distribuição dinâmica de trabalho, balanceamento de carga, suporte parcial para memória virtual partilhada e experimentação com algoritmos de scheduling. Actualmente pretende-se elaborar um estudo aprofundado das suas capacidades, implementando uma série de aplicações complexas que beneficiem largamente de execução paralela.

Referências

[1] 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.*

[2] 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.*

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

[4] 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, 115–131, January 1994.*

[5] L. Lopes and F. Silva. Thread- and Process-Based Implementations of the pSystem Parallel Programming Environment. *Software Practice and Experience, 27(3), 329 – 351, March, 1997.*

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

[7] N. G. Shivaratri and P. Krueger, Two Adaptive Location Policies for Global Scheduling Algorithms. *In 10th Intl Conf. on Distributed Computing Systems, Paris, France, pages 502-509, May 1990.*

Anexo: Multiplicação de Matrizes no di_pSystem

```
#include  "di_pSystem.h"
#include  <stdio.h>

int n;           /* Dimensão da matriz    */
int n_lines,     /* No. linhas por agente */
double *matrix;  /* Matriz a multiplicar  */

/* Função que lê a matriz */
void read_matrix(char *);

/* Tarefa: calculo da multiplicação */
di_pSystem task <mul_columns>(int, double*?);

void  main(int argc, char *argv[]) {
  int i,j;
  int first;  /* 1? linha a calc por tarefa */
  double *result_matrix;/* Matriz resultado */


  /* Ler matriz */
  read_matrix(argv[1]);
  /* Alocar espaço para matriz resultado */
  result_matrix = (double *)malloc(sizeof \
                (double)*n*n);
  /* Iniciar o di_pSystem */
  Start_di_pSystem(argc,argv);
  /* Calcular número de linhas por tarefa */
  n_lines = n/pS_size;
  /* Lançar tarefas */
  for (i=0; i<pS_size; i++) {
   first = n_lines*i;
new task <mul_columns>(first, /
    result_matrix+first*n, n_lines*n);
  }
  /* Linhas computadas sequencialmente */
  first = first+n_lines; n_lines = n-first;
  mul_columns(first,?(result_matrix+first*n));
  /* Terminar di_pSystem */
  End_di_pSystem();
  /* Imprimir matriz resultado */
  print_matrix(result_matrix);
}

/* Função C declarada como tarefa */
task <mul_columns>(int first, double *result) {
  int i,j,k; /* Variáveis de indução */

  /* Ler número de linhas a calcular */
  Read(n_lines);
  /* Calcular elementos da matriz resultado */
  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++;
    }
}
```