Um Modelo de Programação Paralela Simples para Arquitecturas Distribuídas de Baixo-Custo

Hervé Paulino   Fernando Silva   Luís Lopes

Technical Report Series: DCC-99-5

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+6078830 – Fax: +351+2+6003654

`http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html`

Um Modelo de Programação Paralela Simples para Arquitecturas Distribuídas de Baixo-Custo*

Hervé Paulino, Fernando Silva & Luís Lopes

DCC-FC & LIACC, Universidade do Porto,
Rua do Campo Alegre 823, 4150-180 Porto, Portugal
{herve, fds, lblopes}@ncc.up.pt

Outubro 1999

Resumo

Este artigo descreve o desenho e implementação de um sistema de programação paralela para ambientes distribuídos, o di_pSystem. Este sistema proporciona aos utilizadores um modelo de programação próximo de um modelo de memória partilhada tornando a implementação de aplicações mais intuitiva e transparente. O modelo de programação é suportado por uma camada de software, entre a aplicação do utilizador e o sistema operativo, que gere a comunicação, a distribuição e balanceamento de trabalho no sistema. Esta camada permite esconder do utilizador muita da complexidade associada à implementação de aplicações em arquitecturas distribuídas mantendo no entanto o sistema competitivo em termos de performance.

As primitivas de comunicação do sistema são implementadas utilizando o MPI como *backbone*. Os resultados iniciais indicam que o sistema tem uma performance próxima do MPI, facto que se atribui à sua capacidade de balancear dinamicamente o trabalho no sistema.

Keywords: Programação Paralela, Scheduling Dinâmico, MPI.

1 Introdução

Recentemente, máquinas paralelas de baixo custo baseadas em componentes *off-the-shelf* têm sido propostas e contruídas. A arquitectura destas máquinas é predominantemente de memória distribuída, composta por *clusters* de multiprocessadores interligados por redes muito rápidas, como por exemplo a Myrinet [Bod95]. O sucesso deste tipo de arquitecturas está no entanto condicionado pelos ambientes de programação que oferecem aos utilizadores. Sistemas como o PVM [BDJ94] ou o MPI [MPI95], são de grande utilidade para a programação de arquitecturas de memória distribuída. Contudo, estes sistemas requerem dos programadores um conhecimento profundo sobre a arquitectura alvo e o controlo explícito de toda a comunicação. Uma alternativa consiste na utilização de uma camada de software, por

*Artigo apresentado no "11th Symposium on Computer Architecture and High Performance Computing" (SBAC-PAD'99), Natal, Brasil, Outubro, 1999.

exemplo o TreadMarks [KDC94], que forneça memória virtualmente partilhada, permitindo um modelo de programação mais próximo do de memória partilhada.

A facilidade de utilização e a eficiência de um sistema de programação paralela são dois aspectos chave que determinam a sua utilidade e aceitação. Neste sentido, Lopes e Silva [LS97] desenvolveram um sistema de programação paralela para arquiteturas de memória partilhada, o *pSystem*, que permite executar em paralelo programas desenvolvidos em C com um conjunto mínimo de anotações. As vantagens do pSystem residem na portabilidade, eficiência e simplicidade do seu modelo de programação. O sistema é responsável pela criação de agentes computacionais (processos) e pela distribuição e balanceamento do trabalho paralelo entre eles. A modularidade do sistema simplifica ainda a integração de outros algoritmos de scheduling implementados pelo utilizador.

Neste artigo, apresenta-se o desenho e implementação do *di_pSystem*, um sistema de programação paralela para arquitecturas de memória distribuída. Este sistema é uma evolução do pSystem para uma nova arquitectura, por forma a englobar uma classe maior de máquinas paralelas, preservando ao máximo o modelo de programação do pSystem bem como a sua arquitectura modular. Deste modo, cabe ao di_pSystem assegurar a distribuição dinâmica e o balanceamento do trabalho paralelo. Toda a comunicação é implícita e gerida automaticamente pelo sistema. Os utilizadores têm apenas de anotar as funções de um programa que devem ser tratadas pelo sistema como tarefas candidatas a execução paralela. A distribuição das tarefas pelos agentes computacionais é gerida internamente pelo sistema.

No di_pSystem, a interação do sistema com o módulo de comunicação está parametrizada por uma interface bem definida, permitindo desse modo isolar as partes do sistema que são dependentes da máquina onde o sistema está a executar. A vantagem desta organização é permitir a implementação de vários módulos de comunicação que podem ser activados ou desactivados sem afectar a parte restante do sistema. A implementação actual do módulo de comunicação recorre ao MPI para suportar as computações sobre redes heterogéneas de computadores. Contudo, o utilizador não se apercebe do uso deste sistema, pois todas as chamadas ao MPI estão encapsuladas em funções de mais alto nível que fazem parte da interface de comunicação do sistema com este módulo. A grande vantagem do di_pSystem sobre o MPI está na simplicidade de programação pois não requer do programador a codificação de mensagens explícitas no programa. O programador tem no entanto de definir a topologia da máquina paralela num ficheiro de configuração.

O artigo está estruturado da seguinte forma: a secção 2 descreve a arquitectura e modelo de execução do di_pSystem. A secção 3 apresenta um exemplo de um programa e explica o modelo de programação. Finalmente a secção 4 apresenta um estudo preliminar de performance do di_pSystem e uma comparação com o MPI.

2 O di_pSystem

O di_pSystem suporta computações distribuídas sobre uma rede cuja topologia é estática e definida pelo utilizador. Em cada nó da rede existe um agente do sistema que é responsável pela execução local dos programas. Os agentes comunicam entre si para trocar unidades de trabalho, denominadas tarefas, e para receber resultados da execução das mesmas. Tarefas correspondem a *closures* de funções anotadas em programas em C. A figura 1 ilustra a arquitectura de um agente do di_pSystem.

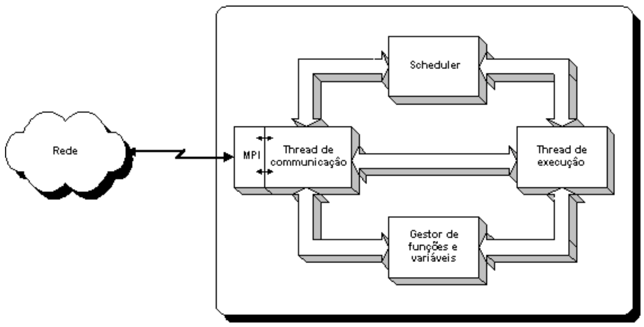

Figura 1: A estrutura interna de um agente do di_pSystem.

Cada agente do di_pSystem é composto por dois *threads*. O *thead de comunicação* gere as mensagens enviadas e recebidas pelo *thread de execução*. Por sua vez o thread de execução é responsável pela computação no agente e está implementado como um motor de execução compacto que executa sequencialmente as tarefas que recebe. Existem outros dois módulos com os quais estes threads interagem: o *scheduler* e o *gestor de funções e variáveis*. O scheduler implementa uma heurística de distribuição de trabalho que gere a fila de tarefas local e importa/exporta trabalho para/de o agente, de acordo com a estratégia escolhida. O gestor de funções e variáveis transforma nomes de funções e de variáveis de leitura global nos seus endereços locais em cada nó IP. Desta forma funções e variáveis são conhecidas pelos seus lexemas na rede e estes são transformados em endereços de memória quando atingem um nó para execução.

De seguida descreve-se em maior detalhe a implementação de cada uma das componentes e a sua interação.

2.1 O Thread de Execução

O thread de execução é o thread principal do agente e é responsável pela criação, execução e suspensão de tarefas, bem como pela colocação de resultados de tarefas locais. Cada agente tem um thread de execução que começa por inicializar as estruturas de dados do sistema (por exemplo, as filas de trabalho) e lança o thread de comunicação, como ilustrado no pseudo-código da função `Start_di_pSystem()` apresentado abaixo. No agente principal (com o identificador `myrank=0`), o thread de execução prossegue a execução do programa. Nos outros agentes, o thread bloqueia até que o scheduler lhe sinalize a chegada de novo trabalho.

```
Start_di_pSystem() {
  initialize_data_structures();
  process_user_options();
  if (myrank != 0) {
    if (coherence_protocol == INVALIDATE) {
      initialize_cache();
      insert_read_only_variables_in_cache();
    }
    launch_communication_thread();
    while (di_pSystem_running)
      if (ExecTaskQueue != EMPTY)
```

```
        Handle_Task();
 }
 else
   launch_communication_thread();
}
```

As tarefas, criadas localmente ou recebidas de um outro agente, podem ser colocadas em duas filas: a *ExecTaskQueue* guarda as tarefas a executar localmente e a *ExportTaskQueue* guarda as tarefas candidatas a execução remota. O uso de duas filas de trabalho tem a vantagem de reduzir e, em muitos casos, evitar as zonas críticas no código permitindo uma gestão de tarefas mais estruturada e eficiente.

A execução paralela de um programa no di_pSystem pode ser vista conceptualmente como uma árvore computacional onde cada nó representa uma tarefa e cada aresta uma dependência de dados (figura 2). Os nós desta árvore são distribuídos pelos agentes durante a execução de uma forma que depende da política de scheduling adoptada. Apesar de por defeito a execução de tarefas ser assíncrona, o sistema inclui um mecanismo de sincronização, necessário para a gestão de dependências de dados. A anotação `Wait()` permite ao programador suspender a execução de uma tarefa, obrigando que toda a sub-árvore abaixo do seu nó seja concluída antes de prosseguir. Sempre que uma tarefa suspende, o thread de execução procura na fila `ExecTaskQueue` novas tarefas para executar.

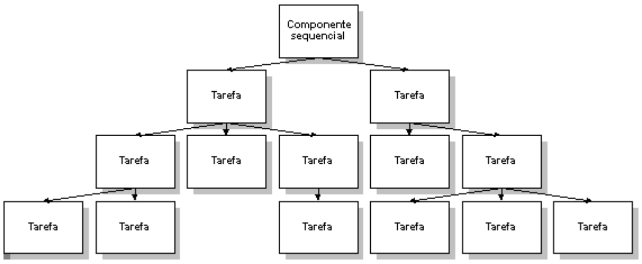

Figura 2: Árvore de execução de uma programa no di_pSystem.

2.2 O Thread de Comunicação

O *thread* de comunicação tem duas camadas e é responsável pelas seguintes acções: recepção e colocação de resultados de tarefas executadas remotamente; envio de resultados de tarefas executadas localmente para agentes remotos, e; recepção e envio de informação sobre carga de trabalho. A camada de mais baixo nível para suporte de comunicação está actualmente implementada em LAM [BDV94], uma das distribuições do MPI.

A execução de um thread de comunicação corresponde basicamente a um ciclo no qual se detecta se existem mensagens a receber ou a enviar.

```
Communication_thread(void *args) {
  unpack_args();
  initialize_communication_layer();
  initialize_work_load_structures();
  while (di_pSystem_running) {
    read_messages();
```

```
    send_read_only_variable_operations();
    send_remote_tasks_results();
    scheduling();
  }
}
```

O thread de comunicação começa por inicializar a camada de comunicação, neste caso o MPI, para que a comunicação entre agentes possa ter lugar. Em seguida entra num ciclo infinito dentro do qual verifica quando chegam novas mensagens e envia mensagens do agente local para o exterior. Existem vários tipos de mensagens: operações sobre variáveis de leitura global (por exemplo, invalidações), actualizações, pedidos de conteúdos ou conteúdos por si só; resultados de tarefas remotas executadas localmente, e; tarefas provenientes ou destinadas a outros agentes.

Como foi já referido a implementação actual utiliza o MPI no módulo de comunicação, tirando vantagem de uma norma de comunicação standard e eficiente. As mensagens entre agentes podem ser simples ou compostas. Mensagens simples contêm valores de tipos *builtin*, e.g inteiros, e são implementadas com simples mensagens assíncronas (no MPI, `MPI_Isend()`). Por sua vez, as mensagens compostas contêm estruturas de dados arbitrárias de tamanho desconhecido, necessitando de operações de empacotamento. Sempre que uma mensagem composta é enviada, aproveita-se para propagar com a mensagem informação sobre quantidade de trabalho do agente local. Isto permite evitar o envio de mensagens específicas para a actualização da informação sobre a carga de trabalho global. Esta informação é utilizada pelo scheduler do sistema.

Apesar de serem utilizadas mensagens assíncronas, é necessário completar o envio anterior antes de começar um novo, sob pena de se destruir a mensagem anterior. Este requisito poderia, por vezes, resultar em *deadlock* caso existissem operações por completar entre os agentes que comunicam. Para evitar essa possibilidade, cada envio de mensagem é precedido de uma detecção e possível recepção de mensagens referentes ao agente para o qual se deseja efectuar o envio.

2.3 O Gestor de Funções e Variáveis

O gestor de funções e variáveis implementa, através de tabelas de *hash*, o mecanismo utilizado pelo di_pSystem para suportar resolução de nomes de funções e variáveis num ambiente distribuído. Este módulo converte os lexemas de funções e variáveis nos seus endereços locais dentro de um agente. Para além disso, também implementa os protocolos necessários para a assegurar a coerência dos valores das variáveis de leitura global. Estes permitem que de dentro de uma tarefa se possam efectuar acessos de leitura a variáveis partilhadas por todos os agentes de computação. Desta forma é possível minimizar o número de argumentos enviados nas tarefas, reduzindo o tráfego na rede. As mensagens relativas a variáveis de leitura global são *demand driven*, ou seja, efectuadas só quando é estritamente necessário.

O agente principal, i.e., o que inicia a computação, mantém os valores correctos de todas as variáveis de leitura global, uma vez que estas são instanciadas na componente sequêncial do programa. Para que os outros agentes possam aceder a estes valores implementaram-se dois protocolos de coerência de cópias de variáveis: *invalidação* e a *actualização* [LH89]. O protocolo de invalidação, utilizado por defeito requer uma *cache*, em cada agente secundário, para gerir as variáveis invalidadas. A anotação `Read()` devolve o valor actualizado de uma variável. Internamente este processo envolve uma pesquisa à *cache* para verificar se o valor correcto se encontra no agente. Em caso afirmativo este é prontamente utilizado, senão é enviado um pedido do valor actualizado ao agente principal.

2.4 O Scheduler

O scheduler implementa a política de distribuição de trabalho adoptada pelo sistema. As decisões sobre a distribuição do trabalho baseiam-se na informação sobre balanceamento de carga recolhidas pelo agente durante a execução. O scheduler e as estruturas de dados relevantes para o balanceamento de carga são implementados de forma modular o que simplifica a implementação e integração de novas estratégias de scheduling. O utilizador do di_pSystem pode escolher uma das quatro estratégias fornecidas pelo sistema ou, se desejar, pode implementar a sua própria estratégia e adicioná-la ao sistema sem interferir nos restantes módulos.

O scheduler interage com ambos threads de um agente. A interação com o thread de execução dá-se quando uma tarefa é criada, devendo decidir se esta deve ser colocada na fila de trabalho local (`ExecTaskQueue`), ou se deve ser candidata a execução remota (`ExportTaskQueue`). Esta decisão baseia-se na quantidade de trabalho local. Por outro lado a interação com o thread de comunicação ocorre no momento em que é necessário decidir se as tarefas da `ExportTaskQueue` devem ser realmente enviadas para um outro agente ou se devem ser colocadas na `ExecTaskQueue`. Esta decisão depende do balanceamento e quantidade de carga de trabalho de todo o sistema. Como exemplo, apresentamos o pseudo-código da implementação da estratégia *Sender Initiated*.

```
Sched_Publishing_Task(task) {
  if (local_work_load < lower_limit)
    insert_task(task,ExecTaskQueue);
  else
    insert_task(task,ExportTaskQueue);
}

Sched_Management() {
  while (ExportTaskQueue != EMPTY) {
    task = get_next_task(ExportTaskQueue);
    if (local_work_load < lower_limit)
        insert_task(task,ExecTaskQueue);
    else {
      agent = find_agent_with_less_work();
      if (agent != myrank)
        send_task(task,agent);
      else
        insert_task(task,ExecTaskQueue);
    }
  }
}
```

3 Programar com o di_pSystem

Programar com o di_pSystem envolve a anotação explícita de programas na linguagem C, identificando desta forma paralelismo potencial no programa. Em seguida ilustra-se um programa em di_pSystem que multiplica uma matriz quadrada por si própria.

```
   #include  <di_pSystem.h>

   int N;
   double A[N][N];
(1)di_pSystem task <mul>(int,double*?);

   void  main(int argc,char *argv[]){
     int i,j;
     double R[N][N];

     read_matrix(argv[1]);
(2)  Start_di_pSystem(argc,argv);
     for (i=0; i<N; i++)
(3)    new task <mul>(i,?R[i],N);
(4)  End_di_pSystem();
     print_matrix(result_matrix);
   }

(5)task <mul>(int i,double *r){
     int j,k;

     for (j=0; j<N; j++) {
       *r=0;
       for (k=0; k<N; k++)
         *r = *r + A[i][k]*A[k][j]);
       r++;
     }
   }
```

O programa começa por declarar cada instância da função `mul` (1) como sendo uma tarefa candidata para execução paralela. Na função `main()` faz-se a leitura, a partir de um ficheiro, da matriz de entrada e inicializa-se o di_pSystem com os argumentos introduzidos na linha de comando (2). A partir deste ponto o agente principal continua a execução do programa, enquanto os demais entram em ciclo aguardando trabalho. Com a continuação da execução, o agente principal cria `N` tarefas (3) que são geridas pelo scheduler e eventualmente distribuídas por todo o sistema. Continua depois até (4) onde sincroniza aguardando a conclusão de todas as tarefas geradas em (3). Neste ponto (4), a função `main()` é suspensa pelo sistema como se tivesse sido sujeita a um `Wait()`. A condição de suspensão é testada periodicamente e enquanto perdurar são executadas outras tarefas. Entretanto os restantes agentes recebem dos seus schedulers tarefas para executarem, computando linhas da matriz resultado. Os resultados, neste caso, são enviados automaticamente para o agente principal. Quando todos tiverem sido recolhidos, a função `main()` retoma a sua execução, permitindo ao agente principal continuar para imprimir a matriz resultado.

4 Trabalho Relacionado

O di_pSystem apresenta-se como um sistema de programação paralela para arquitecturas de memória distribuída, especialmente *clusters* de baixo custo. Para a programação deste tipo de arquitecturas os sistemas mais utilizados são o PVM [BDJ94] e o MPI [MPI95]. O PVM foi o principal impulsionador da utilização de redes de computadores como máquinas paralelas virtuais. O MPI é o standard de comunicação do momento. Estes sistemas, apesar de permitirem um bom escalonamento dos recursos computacionais, são difíceis de programar, exigindo do utilizador um conhecimento do padrão de comunicação da aplicação e da arquitectura subjacente. O di_pSystem destaca-se neste ponto com a sua facilidade de programação ao suprimir a necessidade de comunicação explícita, de distribuição e de balanceamento de trabalho.

Mais recentemente surgiram os sistemas baseados em modelos DSM (Distributed Shared Memory), como por exemplo, o Treadmarks [KDC94] e o OpenMP [OMP97]. Apesar destes sistemas terem alguns mecanismos de suporte para memória partilhada mais sofisticados que o di_pSystem, no caso do TreadMarks o modelo de programação obriga o utilizador a fazer o balanceamento e distribuição do trabalho paralelo. O OpenMP pretende surgir como um novo standard para arquitecturas DSM e tem como vantagem principal sobre o di_pSystem a possibilidade de paralelização incremental de aplicações. As principais vantagens do di_pSystem relativamente aos sistemas descritos consistem em: a) distribuição e gestão dinâmica de trabalho paralelo; b) possibilidade de escolha do scheduler para a execução de uma aplicação e ; c) o facto de a camada de comunicação ser modular permitindo a sua implementação utilizando diferentes *backbones* (por exemplo, MPI, PVM, TreadMarks).

5 Performance

Para podermos fazer um estudo de performance inicial, implementamos um conjunto de pequenas aplicações. Actualmente estão em fase de implementação duas aplicações maiores que envolvem *data mining* e simulações de dinâmica de fluídos. Estas aplicações permitirão um estudo mais elaborado onde poderemos fundamentar a utilidade do sistema bem como melhorar a sua performance.

Os resultados [Pau98] foram obtidos em arquitecturas paralelas de baixo custo, nomeadamente um *cluster* de quatro dual-Pentium II a 300MHz interligados por uma rede Myrinet e um outro *cluster* de dois quad-Pentium Pro a 200MHz ligados por uma rede Fast-Ethernet. Por uma questão de espaço, apresentamos apenas resultados relativos às aplicações `Mmat` – multiplicação de matrizes, e `Pi` – cálculo de π por integração numérica através da regra dos trapézios. Os resultados apresentados referem-se apenas aos obtidos no *cluster* interligado por Myrinet.

O primeiro gráfico, figura 3, refere-se a um estudo de ganhos de velocidade elaborado com as quatro estratégias de distribuição de trabalho disponíveis. O parâmetro `n` indica o número de partições do integral. Cada tarefa tem um tempo de execução na ordem dos 0.07 segundos.

Como esta implementação do di_pSystem recorre às funções de comunicação do MPI, é interessante comparar implementações de di_pSystem com implementações do MPI. Um programador experiente que implemente uma aplicação que envolve paralelismo regular obterá certamente melhores resultados. No entanto, a facilidade de programação do di_pSystem

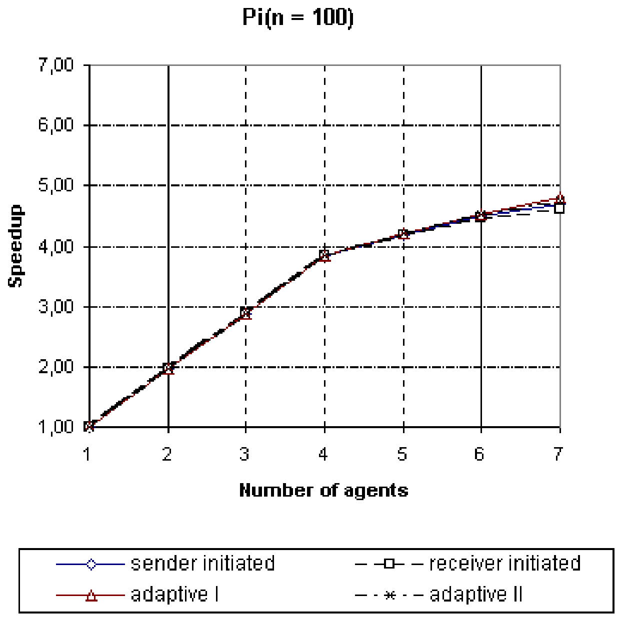

Figura 3: Ganhos de velocidade.

compensa alguma perda de performance que possa acontecer. A figura 4 ilustra esta comparação para as duas aplicações, tendo sido utilizada a política de distribuição de trabalho *Adaptive I*.

O gráfico relativo à aplicação `Mmat` revela que até cinco agentes o di_pSystem se mostra melhor que o MPI, este facto deve-se à sobreposicão de computação com comunicação que o di_pSystem possibilita ao aproveitar processadores parados. A queda posterior dá-se pela falta destes. O gráfico relativo à aplicação `Pi` revela que com o aumento da granulosidade das tarefas (parâmetro `GR`), a diferença entre as duas implementações diminui, o que parece indicar que para tarefas de maior granulosidade esta se torna desprezável.

6 Conclusões

Neste artigo apresentou-se o modelo de execução, desenho e implementação de um sistema de programação paralela para arquitecturas de memória distribuída – o di_pSystem. O sistema visa simplificar a tarefa do programador de aplicações paralelas. Para o efeito integrou-se no sistema toda a comunicação inerente à execução de programas, funcionalidades como distribuição e balanceamento dinâmico de trabalho e suporte parcial para memória virtual partilhada. Depois de ter sido elaborado um estudo de performance preliminar, actualmente estão em fase de implementação duas aplicações que beneficiam largamente de execução paralela e que têm um de grau de complexidade que nos permitirá tirar elações mais concretas sobre a utilidade do sistema.

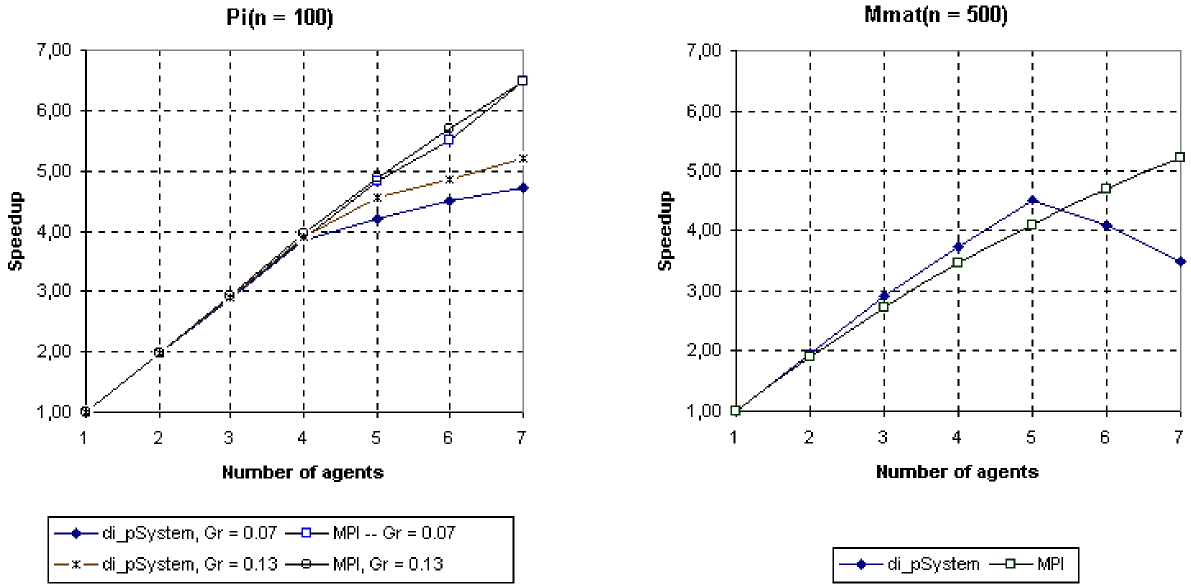

Figura 4: di_pSystem vs MPI.

Agradecimentos

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

Referências

[Bod95] Nanette J. Boden, et. al., Myrinet - A Gigabit-per-Second Local-Area Network. *IEEE-Micro*, 15(1), 29–36, February 1995.

[BDJ94] 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.

[BDV94] G. Burns, R. Daoud and J. Vaigl. LAM: an Open Cluster Environment for MPI. Ohio Supercomputer Center, 1994.

[KDC94] 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.

[LH89] K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4) 321-359, November, 1989.

[LS97] L. Lopes and F. Silva. Thread- and Process-Based Implementations of the pSystem Parallel Programming Environment. *Software Practice & Experience*, 27(3), 329 – 351, March, 1997.

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

[OMP97] OpenMP: A Proposed Industry Standard API for Shared Memory Programming, October 1997. At `www.openmp.org`.

[Pau98] H. Paulino, Desenho e Implementação do pSystem para Arquitecturas de Memória Distribuída. Tese de Mestrado, Faculdade de Ciências da Universidade do Porto, 1998.

[SK90] N. G. Shivaratri and P. Krueger, Two Adaptive Location Policies for Global Scheduling Algorithms. *In 10th International Conference on Distributed Computing Systems*, 502-509, May 1990.