# Escalonamento Dinâmico de programas MPI-2 utilizando Divisão e Conquista

Guilherme P. Pezzi, Márcia C. Cera, Elton N. Mathias, Nicolas Maillard, Philippe O. A. Navaux
Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil
{pezzi, mccera, enmathias, nicolas, navaux}@inf.ufrgs.br

## Resumo

*MPI é um padrão para programação de aplicações científicas de alto desempenho e é muito utilizado em ambientes com recursos dedicados, como Clusters. A recente implementação da norma MPI-2 oferece mecanismos que permitem utilizar recursos computacionais cuja disponibilidade altera-se dinamicamente. Este trabalho estuda dois desafios que surgem com a utilização de ambientes dinâmicos: como programar as aplicações para se adaptarem aos recursos e como fazer um bom aproveitamento dos recursos disponíveis. O modelo de programação proposto para este trabalho é o D&C, pois é mais abrangente que o modelo Bag of Tasks, classicamente utilizado nesse tipo de ambiente. Para o bom aproveitamento dos recursos, propõe-se usar algoritmos de escalonamento on-line (Round-Robin e Escalonamento com lista). Por fim, para validar a proposta, são apresentadas aplicações desenvolvidas e resultados de execuções com diferentes algoritmos para escolha dos recursos utilizados.*

## 1 Introdução

A biblioteca MPI (*Message Passing Interface*) [9] é uma das mais populares bibliotecas de comunicação para programação paralela na área de Processamento de Alto Desempenho (PAD). Com o MPI, o emprego do paradigma de mensagens é bem definido e pode ser facilmente e eficientemente empregado em linguagens seqüenciais clássicas, tais como C/C++ e Fortran. Devido a suas características, MPI é uma interface de comunicação bastante empregada em máquinas paralelas e agregados (clusters) de computadores. Isso se comprova pela adaptação para MPI dos benchmarks de PAD (por exemplo, Linpack [8] e NAS [6]) e dos grandes desafios de PAD (previsão do tempo, astrofísica, química quântica, simulações nucleares, etc [2]).

Com o MPI, os processos da aplicação estão organizados em grupos (*communicator*) e cada um deles possui um identificador único (*rank*) dentro de seu grupo. É através desse identificador que o comportamento de cada um dos processos é diferenciado. Inicialmente MPI permitia apenas a criação estática de processos (ao disparar a aplicação), sendo este padrão conhecido como MPI-1.2. Nesse padrão, um conjunto de processos era definido no início da execução da aplicação e se mantinha o mesmo até seu término. A estaticidade na criação de processos, particularmente bem adaptada à programação em agregados, dificultava o uso de MPI em ambientes dinâmicos, onde o conjunto de computadores varia constantemente. Buscando contornar essa deficiência e aproximar-se do modelo de programação de PVM (Parallel Virtual Machine) [14], foi definida a norma MPI-2. Essa norma suporta a criação dinâmica de processos (ou seja, em tempo de execução), acesso remoto à memória (RMA - *Remote Memory Access*) e entrada e saída de dados paralela.

Para poder se adaptar à dinamicidade permitida pelo MPI-2, a aplicação deve ser programada seguindo um modelo paralelo que possibilite a criação dinâmica de processos. Uma opção possível é o modelo de Divisão e Conquista (*D&C*). Esse modelo caracteriza-se por dividir recursivamente um problema até que sua solução seja simples. Dentro dessa perspectiva, pode-se controlar o paralelismo e a granularidade das tarefas através do número de níveis de recursividade empregados.

Este artigo irá apresentar uma iniciativa de desenvolvimento de aplicações segundo o modelo de *D&C* empregando MPI-2. O restante do texto está organizado da seguinte forma: a seção 2 apresenta o modelo de *D&C*, sua complexidade paralela e considerações sobre como se escalona esse tipo de aplicação, além de trabalhos relacionados com este modelo de programação. A seção 3 mostra quais as dificuldades e possibilidades para a programação de aplicações *D&C* com MPI. Após, a seção 4 apresenta os resultados experimentais obtidos na iniciativa de programar aplicações *D&C* com MPI-2. Por fim, a seção 5 conclui o artigo.

## 2 Divisão e Conquista Paralela

Além da técnica de Divisão e Conquista ser classicamente usada em programação seqüencial para prover algoritmos altamente eficientes [15], ela pode também ser interessante para a programação paralela: resultados teóricos mostram que ela permite obter algoritmos paralelos eficientes. Em nível prático, vários ambientes de programação usaram este modelo, com excelentes resultados: é o caso, por exemplo, de Cilk [1], de Satin [18] ou ainda de Athaspascan [3]. Além disso, foi comprovado e testado experimentalmente que se pode escalonar eficientemente os processos de um programa paralelo *D&C*.

A seção 2.1 detalha este modelo *D&C*. A seguir (seção 2.2), uma justificativa teórica e prática é dada por seu emprego em programação paralela. Por fim, a seção 2.3 detalha os mecanismos eficientes de escalonamento que existem para tais programas.

### 2.1 Divisão e Conquista

Dada uma instância de um problema, ela é decomposta em sub-instâncias menores, que são resolvidas separadamente. A decomposição é feita até que o sub-problema seja simples e sua solução imediata. As soluções parciais são então combinadas para se obter a solução da instância original do problema. Uma definição recursiva para um dado de entrada $e$ pode ser escrita como:

$$\text{Solução}(e): \begin{cases} \text{se simples}(e) \rightarrow \text{direto}(e) \\ \text{senão combina}\begin{pmatrix} \text{solução}(\text{parte}_1(e)), \\ \text{solução}(\text{parte}_2(e)) \end{pmatrix} \end{cases}$$

Uma forma comum de representar aplicações *D&C* é através de grafos acíclicos dirigidos, conhecidos como *Directed Acyclic Graphs* (DAGs) [15]. Em um DAG, os vértices e arestas representam, respectivamente, os processos e as dependências entre eles. A Figura 1 mostra uma aplicação vista como um DAG: a execução de $P_0$ depende de dados que vão ser calculados por $P_1$ e $P_2$. Vale ressaltar que os vértices não representam processadores, e sim processos, que podem ou não estar no mesmo processador. Neste trabalho, denomina-se como "tarefa" um processo de programa *D&C*, que inclui uma comunicação no início de sua execução - para receber os dados de entrada - e outra no final - para retornar os resultados.

### 2.2 Eficácia de Algoritmos *D&C* em Programação Paralela

Quando se pode usar mais de um processador para executar um algoritmo *D&C* paralelo, duas características dos mesmos possibilitam uma boa eficácia:

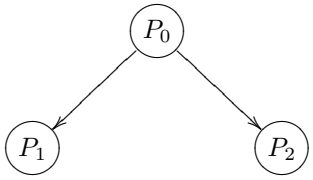


**Figura 1. Aplicação *D&C* vista como um DAG.**

1. o fato de se usar um algoritmo recursivo leva diretamente a um DAG que tem uma profundidade logarítmica em função do tamanho $n$ da entrada. Caso se tenha suficientes processadores para executar em paralelo as chamadas recursivas, pode-se obter dessa forma um tempo de execução logarítmico. Quando o número de recursos fica menor do que um polinomial de $n$, o algoritmo pertence à classe NC, que classifica na teoria PRAM algoritmos "altamente paralelos" [11, 12]. É o caso, por exemplo, de vários algoritmos de ordenação, do cálculo do prefixo ou da soma iterada.

   Nota-se, porém, que a complexidade obtida é logarítmica apenas se a operação de fusão dos resultados parciais ("combina") pode ser efetuada de forma eficiente em paralelo;

2. pelo fato de haver divisão recursiva em sub-problemas, pode-se controlar dinamicamente a profundidade das chamadas recursivas, assim adaptando-se o número de tarefas a serem executadas em paralelo. Dessa forma, pode-se adaptar dinamicamente o grau de paralelismo ao número de processadores disponíveis durante a execução. Nota-se que essa característica é bastante usada também em programação seqüencial, afim de controlar o tamanho da pilha de chamadas recursivas e de limitá-la às capacidades do sistema operacional.

Alguns ambientes de programação paralela foram concebidos para dar suporte à programação *D&C*. O Cilk e o Satin, por exemplo, obtiveram ótimos desempenhos com este modelo.

O Cilk [1] foi desenvolvido no *Massachusetts Institute of Technology* (MIT) e acrescenta à linguagem C três palavras chaves: `cilk`, inserido na frente da declaração de um procedimento, torna-o passível de ser chamado de forma assíncrona; `spawn` permite a criação dinâmica de novas tarefas para efetuar essas chamadas; por fim, o `sync` possibilita a sincronização entre as tarefas que foram criadas pelo `spawn`. O Cilk foi usado para implementar vários programas *D&C* que mostraram ótimo desempenho em arquiteturas com memória compartilhada. Destaca-se, entre outros, o programa de xadrez "Socrates" [7] que foi premiado.

O Satin [17, 18] é um ambiente de programação baseado em Java e focado na execução de aplicações do tipo *D&C* em ambientes com memória distribuída. Ele foi desenvolvido utilizando a plataforma Ibis [16], que objetiva melhorar o desempenho de aplicações Java distribuídas e permite o aproveitamento de recursos computacionais que se alteram dinamicamente. O Satin incorpora algumas das características do Cilk e, além disso, acrescenta funcionalidades necessárias para execução em arquiteturas sem memória compartilhada.

### 2.3 Escalonamento de Programas *D&C*

O escalonamento eficiente de programas paralelos é um problema NP-Completo em seu caso geral, ou seja, quando não se tem informações sobre a duração das tarefas, suas dependências ou sobre os recursos disponíveis. Nas soluções de escalonamento geralmente empregam-se heurísticas, que, no melhor dos casos, garantem um tempo de execução paralelo que aproximam-se (dentro de um fator multiplicativo) do tempo ótimo. No contexto de programas *D&C*, duas abordagens são possíveis e estão detalhadas a seguir.

Caso os recursos não sejam usados em exclusão mútua pelos processos, pode-se tentar balancear a carga, ou seja, alocar os processos de forma equilibrada entre os processadores. Um esquema de tipo Round-Robin pode ser empregado para que o processador $i = 0, \ldots, p-1$ receba $\lfloor n/p \rfloor$ ou $\lfloor n/p \rfloor + 1$ dentro dos $n$ processos recursivamente criados: o grau máximo de desequilíbrio é 1 processo. Nota-se que este esquema não é limitado ao modelo *D&C*.

Quando se quer usar os recursos de forma exclusiva, ou seja, executar a cada instante apenas um processo em um processador, o uso do modelo *D&C* possibilita um mecanismo de roubo de tarefas (*workstealing*). Foi comprovado matematicamente por Blumofe e Leiserson [1] que este algoritmo é o melhor possível, para tal modelo de programação. Foi por este motivo que o ambiente Cilk implementou o modelo *D&C*. O roubo de tarefa consiste de uma forma distribuída de se manter uma lista de tarefas prontas a serem executadas, sendo que logo que um processador se torna ocioso, ele retira uma das tarefas prontas da lista e passa a executá-la. Dessa forma, enquanto há tarefas prontas, garante-se que nenhum processador ficará ocioso. Por isso, o algoritmo é o melhor possível.

Na versão distribuída, a lista consiste em uma *deque* local a cada processador. Uma *deque* é uma pilha onde se pode empilhar/desempilhar tarefas normalmente, porém com a possibilidade de desempilhar tanto do topo como na base da pilha. Quando uma tarefa em execução em um determinado processador cria (dinamicamente) novas tarefas, elas serão empilhadas no topo da pilha local. Quando um processador se torna ocioso, ele primeiro desempilha uma tarefa pronta no topo de sua pilha. Quando a mesma está vazia, ele emite um pedido de roubo a um outro processador, escolhido pseudo-aleatoriamente. Ao receber tal pedido, um processador desempilha uma tarefa na base de sua pilha, e a manda para o processador que fez um pedido por tarefa. Assim, há roubo de trabalho e balanceamento de carga "sob demanda".

## 3 Divisão e Conquista com o MPI

Com o MPI, o uso de chamadas recursivas é relativamente técnico [13], mas possível. É necessário gerenciar, no programa, a pilha de chamadas recursivas. A seção 3.1 detalha como se pode programar segundo um modelo *D&C* com o MPI-1.2. A seção 3.2 explica como fazê-lo com o MPI-2.

### 3.1 Divisão e Conquista com MPI-1.2

MPI-1.2 não provê criação dinâmica de processos. Para efetuar chamadas recursivas em paralelo, deve-se executá-las em processos que já foram disparados. Uma possibilidade, no caso de programas *D&C*, é distribuir entre os processos (de forma estática) os primeiros nós da árvore de chamadas, e deixar cada processo executar recursivamente as sub-árvores tendo esses nós por raiz. Apesar de ser muito simples de implementar, essa opção induz possíveis desequilíbrios de carga, uma vez que nada garante que as sub-árvores tenham todas a mesma profundidade [13].

Uma solução consiste na programação explícita da pilha de chamadas recursivas. Em lugar de executar a chamada recursiva diretamente, se empilha um descritor do procedimento a ser chamado no topo de uma pilha interna. No caso do MPI, pode-se empilhar simplesmente os parâmetros de entrada do procedimento (por exemplo, no caso da computação de Fibonacci apresentado a seguir, o inteiro $n$ de entrada). Assim, cada processo pode executar um procedimento inicial, empilhar os que pode executar em paralelo e prosseguir com sua execução até encerrá-la. Neste momento, pode desempilhar uma das chamadas empilhadas e continuar a executar.

Além disso, essa programação com pilhas explícitas possibilita o emprego de roubo de tarefas: quando um processo tem sua pilha vazia, ele pode mandar uma mensagem para outro, que retornar parte de sua pilha local para executar. No entanto, essa troca de mensagens necessita que, durante sua execução, um processo teste se está recebendo mensagens de roubo. Para não bloquear na recepção, deve ser empregado um `MPI_Irecv` (recepção assíncrona). A freqüência de teste de recepção deve ser acertada para que um pedido de roubo seja notificado rapidamente e atendido, porém sem ser muito alta para não prejudicar a execução normal do processo.

Os parágrafos acima mostram que, apesar da norma MPI-1.2 não prever a criação dinâmica de processos, é possível empregá-la na programação *D&C*. Porém, a programação não é trivial, pois engloba uma série de detalhes e o problema agrava-se quando aplicado a ambientes dinâmicos como as grades. Por isso, é mais adequado o uso do MPI-2 para a programação *D&C*.

### 3.2 Divisão e Conquista com MPI-2

MPI-2 provê uma interface que permite a criação dinâmica de processos durante a execução de um programa MPI, que podem se comunicar através de troca de mensagens. Apesar da norma MPI-2 prover outras funcionalidades, este trabalho foca a criação dinâmica de processos. Maiores informações sobre os recursos do MPI-2 podem ser encontrados em [10].

A primitiva introduzida que cria processos durante a execução de um programa MPI é o `MPI_Comm_spawn`. Os principais argumentos dessa primitiva são: nome do executável, que deve ser um programa MPI padrão (com as instruções `MPI_Init` e `MPI_Finalize`); os parâmetros passados pela linha de comando; o número de processos que deve ser criado; o comunicador que será enviado para os processos criados e o comunicador que será retornado para que o processador possa se comunicar com os processos criados.

Como este trabalho trata problemas recursivos de divisão-e-conquista, pode-se limitar a comunicação: os processos podem se comunicar apenas com seus processos criados (chamados de filhos) e seu criador (chamado de pai). Dessa forma o comunicador enviado para o filho deverá conter apenas o processo pai, evitando a sobrecarga de criação e sincronização de grupos que não precisam se comunicar.

Outra limitação que se pode fazer é quanto aos momentos de troca de mensagens entre pai e filho: é feita sincronização na criação dos processos, para envio dos dados de entrada, e na finalização, para retorno dos resultados. Os dados de entrada podem ser enviados no momento da criação do novo processo, utilizando o argumento de linha de comando do executável MPI. No caso de dados complexos, pode-se utilizar o recurso `MPI_Datatype` para criar um tipo de dados complexo e enviar os dados de entrada por troca de mensagens MPI. A seguir, serão apresentadas execuções de aplicações com diferentes formas de sincronização: a primeira utiliza utiliza troca de mensagens para enviar um vetor simples (seção 4.1), a segunda utiliza o argumento de linha de comando (seção 4.2) e a terceira utiliza `MPI_Datatype` para enviar vetores com diferentes tipos de dados (seção 4.3). Já para o retorno dos resultados ao final da execução, a única opção é utilizar troca de mensagens MPI.

### 3.3 Exemplos de programação D&C com MPI-2

Esta seção apresenta duas aplicações *D&C* e como foram implementadas utilizando MPI-2.

**Fibonacci:** esta aplicação faz cálculo do *n-ésimo* número de *Fibonacci* seguindo a definição recursiva. Caso o $n$ pedido seja menor que $2$, o processo retorna para o pai o próprio $n$ através de uma mensagem de `envio`. Caso contrário, o processo cria $2$ novos processos para calcular $n-1$ e $n-2$, aguarda os processos retornarem o resultado através de dois `recebimentos` bloqueantes e, então, soma os resultados parciais e retorna para o pai o valor através de uma mensagem de `envio`.

```
cilk int fib (int n){
  if (n < 2) return n;
  else{
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;

    return (x+y);
  }
}
```

```
n=atoi(argv[1]);
if (n < 2 )  MPI_Send(n, parent);
 }else{
    int x,y;
    MPI_Comm_spawn(command, n-1);
    MPI_Comm_spawn(command, n-2 );
    MPI_Recv(x, children_comm[0]);
    MPI_Recv(y, children_comm[1]);
    n=x+y;
    MPI_Send(n, parent);
}
```

**Figura 2. Trecho de código do cálculo de Fibonacci, (a) com Cilk (b) com MPI-2**

A Figura 2 compara um trecho de código para cálculo do *n-ésimo* número de *Fibonacci*: na esquerda (a) está representado o código de uma função com criação de processos no Cilk e na direita (b) a implementação em MPI-2. Nos dois ambientes a criação de processo é feita de forma semelhante, através de uma primitiva `spawn`. A sincronização no Cilk é feita com a primitiva `sync` e no MPI-2 é feita através de troca de mensagens.

**N-Queens:** O problema N-Queens consiste na colocação de $n$ rainhas em um tabuleiro de xadrez, de tamanho $n \times n$, de forma que nenhuma rainha capture uma a outra. Isto é, a busca de configurações nas quais não existam mais de uma rainha em cada linha, coluna e diagonal. Existem muitas formas de solução desse problema, sendo o algoritmo de *backtrack*, desenvolvido através do modelo *D&C*, a forma mais utilizada. Este trabalho apresenta duas implementações *D&C* do N-Queens: a primeira (seção 4.1) que retorna a primeira solução encontrada e a segunda (seção 4.3) que busca todas as possíveis soluções utilizando *backtrack*.

O algoritmo de *backtrack* desenvolvido consiste na colocação ordenada e exaustiva de rainhas linha a linha, até que configurações válidas sejam encontradas. Toda vez que determinado posicionamento possua rainhas em posição inválida, o algoritmo retorna à última configuração válida e prossegue avaliando novas posições.

A abordagem *D&C* desenvolve-se através da divisão do problema do posicionamento das rainhas em problemas menores, pela colocação da primeira rainha em cada uma das posições possíveis da primeira linha. Depois disso, tarefas são subdivididas recursivamente pela colocação de cada uma das rainhas seguintes. Cada uma dessas subdivisões gera novas tarefas, que são executada por um novo processo, lançado em um dos nós disponíveis no ambiente de execução. Cada um desses processos recebe a configuração atual do tabuleiro, e segundo sua posição na árvore de recursão ou retorna o número de soluções encontradas (nós folha), ou gera novas tarefas posicionando a próxima rainha e espera pelo retorno dos processos criados por este. Para controlar o paralelismo, pode-se escolher até qual nível (profundidade) da recursão novos processos devem ser criados e quantos processos em cada nível.

### 3.4 Escalonamento On-line com MPI-2

A norma MPI-2 não prevê mecanismos de escalonamento, isto é, meios para escolha de qual recurso receberá cada processo criado. Essa escolha deve ser feita diretamente no código da aplicação e pode, por exemplo, utilizar mecanismos oferecidos pelas implementações MPI. O LAM-MPI oferece um mecanismo que faz um balanceamento Round-Robin dos processos em cada chamada `MPI_Comm_spawn` entre os recursos disponíveis. Porém, foi constatado que quando as aplicações criam apenas um processo por chamada, esse mecanismo não faz balanceamento adequado [4]. Isto ocorre pois não são consideradas as decisões anteriores de escolha dos recursos, e em todas chamadas `MPI_Comm_spawn` inicia-se um Round-Robin a partir do mesmo recurso.

Para permitir um balanceamento de carga nesse tipo de aplicação, é preciso escolher em qual nó deve-se iniciar o Round-Robin. Esta escolha deve ser feita antes de cada chamada `MPI_Comm_spawn` e através da primitiva `MPI_Info_set`. Vale destacar que esta solução não é genérica e não garante um balanceamento eficiente, já que cada processo fará a escolha do recurso onde será criado o novo processo sem conhecimento das escolhas dos demais processos. Porém, esta solução já permite uma melhor utilização dos recursos, que podem se alterar dinamicamente, comparado com a utilização do mecanismo de escalonamento oferecido pelo LAM-MPI.

Um maneira de melhorar o balanceamento é armazenar as decisões de escalonamento, de modo que seja possível fazer uma distribuição Round-Robin dos processos nos recursos disponíveis. Uma possível solução para resolver este problema é utilizar uma biblioteca de escalonamento [5]. Essa abordagem utiliza redefinição de algumas primitivas do MPI, fazendo com que a aplicação, para efetuar o escalonamento, execute código da biblioteca nos pontos onde sejam chamadas essas primitivas. Objetivando facilitar o escalonamento para o programador e evitar redefinição de primitivas, que podem gerar intrusão na aplicação, foi implementado um escalonador independente da aplicação. Esse escalonador é centralizado e pode ser consultado sempre que um novo processo é criado. O escalonador utiliza primitivas MPI para publicar uma porta de conexão e a aplicação pode requisitar recursos através da porta publicada.

## 4 Resultados obtidos

O objetivo dos experimentos apresentados a seguir é validar o uso de MPI-2 para programar aplicações *D&C* recursivas e dinâmicas em ambientes com memória distribuída. A validação foi dividida em três etapas distintas: adequação dos mecanismos de programação oferecidos pelo ambiente para programação *D&C*; possibilidade de utilização de recursos dinâmicos com MPI-2 e resultados de desempenho em ambiente com memória distribuída. Vale lembrar que a utilização de recursos dinâmicos não é essencial para executar aplicações *D&C*. No entanto, essa possibilidade acrescenta flexibilidade e pode permitir executar as aplicações em ambientes de grade.

A seção 4.1 apresenta uma comparação dos ambientes Cilk e MPI-2, através da execução de uma aplicação semelhante nos dois ambientes. Após, a seção 4.2 mostra uma solução para a utilização de recursos dinamicamente disponibilizados ao longo da execução de uma aplicação MPI-2. Por último (seção 4.3), são apresentados resultados de desempenho com uma aplicação *D&C* recursiva utilizando MPI-2. Os experimentos foram realizados utilizando LAM MPI-2 versão 7.1.2 em um cluster de computadores Linux. Todos os experimentos utilizaram o escalonador Round-Robin centralizado, apresentado na seção 3.2. Essa decisão foi tomada pois o mecanismo oferecido pelo LAM não faz um balanceamento de carga adequado em aplicações que criam apenas um processo por chamada da primitiva `MPI_Comm_Spawn` [5].

### 4.1 Comparando Cilk e MPI-2 com N-Queens

Para comparar os dois ambientes, foi utilizado o exemplo do problema N-Queens fornecido pelo ambiente Cilk (versão 5.4.2.3). Essa implementação retorna apenas a primeira solução encontrada e foi programada de forma recursiva. A função recebe como argumento um tabuleiro com

as rainhas posicionadas até o momento e, para cada nova posição válida de rainha, executa recursivamente a função com a nova configuração de tabuleiro. Quando $n$ rainhas estiverem posicionadas, a função retorna a solução e termina os demais fluxos de execução. No Cilk, cada fluxo é implementado por uma *thread* com memória compartilhada e, no MPI-2, por um processo que se comunica por troca de mensagens.

**Tabela 1. Execuções do N-Queens MPI-2 baseado no exemplo do ambiente Cilk ($N = 7$).**

| | Num. processos | tempo(ms) | Tempo/criação(ms) |
|---|---|---|---|
| | 64 | 1100 | 17,19 |
| | 74 | 1120 | 15,14 |
| | 84 | 1100 | 13,1 |
| | 83 | 1100 | 13,25 |
| | 89 | 1120 | 12,58 |
| | 92 | 1200 | 13,04 |
| | 111 | 1260 | 11,35 |
| Média | 85,29 | 1142,86 | 13,66 |

A execução do N-Queens com $n = 7$ no ambiente Cilk tem tempo médio de $3ms$ em uma máquina bi-processada. Já com MPI-2, a execução tem um tempo médio de $1142ms$ utilizando 5 máquinas bi-processadas. A Tabela 1 mostra a quantidade de processos MPI-2 criados, o tempo total de execução e uma estimativa do tempo para a criação de cada processo MPI-2. Essa estimativa foi feita desconsiderando-se o trabalho executado por cada processo, que é muito pequeno comparado ao custo de criação remota de processos - que envolve transferência do executável e troca de mensagens pela rede.

O objetivo do experimento apresentado é validar a utilização do modelo *D&C* com MPI-2 através da comparação com um ambiente já consolidado e específico para *D&C*. Com os resultados obtidos foi possível adaptar uma aplicação restrita a ambientes com memória compartilhada para ambientes com memória distribuída, que possuem maior escalabilidade. Porém, os tempos apresentados evidenciam um fator crucial na paralelização de aplicações: é necessário fazer um balanceamento adequado entre o tempo de processamento e o tempo de criação de processos.

## 4.2 Utilização de recursos dinâmicos

Nesta seção, validamos o aproveitamento de recursos dinâmico, com um programa *D&C* MPI-2. O programa gera uma grande quantidade de processos, os quais podem ser escalonados dinamicamente a medida que surgem novos processadores. A aplicação escolhida para este teste é o Fibonacci, descrito em 3.3. No entanto, devido a uma limitação do LAM MPI quanto ao número de processos que podem executar em paralelo (atualmente em torno de 500 processos por máquina), não é possível obter uma execução dessa implementação do Fibonacci com valores muito altos. A solução adotada foi executar o Fibonacci acrescentando uma chamada `sleep` antes da criação de novos processos. Assim, foi possível obter um tempo de execução suficiente para acrescentar novos recursos e verificar sua utilização (Figura 3).

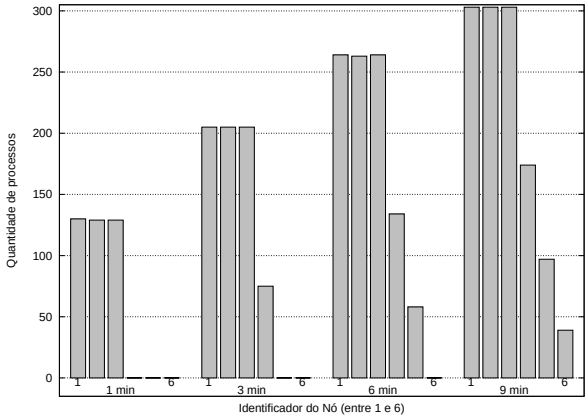


**Figura 3. Utilização de recursos dinâmicos com lamgrow**

O ambiente de execução, composto inicialmente por três nós, recebeu o incremento de novos nós durante a execução da aplicação aos 3, 6 e 9 minutos. Em cada incremento foi adicionado um nó utilizando a primitiva `lamgrow`, que insere o novo nó na lista de recursos disponíveis ao ambiente MPI. A Figura 3 mostra uma fotografia do número de processos lançados em cada um dos recursos disponíveis no instante imediatamente anterior à inserção de novo recurso.

É possível notar que o mecanismo de Round-Robin desenvolvido permitiu uma distribuição homogênea entre os processadores que fizeram parte do ambiente desde o princípio. Assim, pode ser suprida a carência na distribuição eficiente de processos em ambientes distribuídos. A medida que novos recursos foram inseridos, estes puderam ser utilizados de forma automática e imediata, sendo que receberam mais tarefas os recursos que permaneceram maior tempo no ambiente. No entanto, o mecanismo de Round-Robin pode não ser adequado para todas as aplicações e ambientes.

Uma forma de melhorar a distribuição é considerar outras métricas para fazer o balanceamento. Em [5] são apresentados desempenhos de aplicações executadas com um gerenciador de recursos que faz o balanceamento de cargas baseado na coleta informações sobre utilização de CPU dos nós disponíveis.

### 4.3 Desempenho do N-Queens com MPI-2

Como partiu-se de um ambiente com memória compartilhada para memória distribuída, é necessária atenção especial a granularidade do trabalho, considerando o custo de criação de novos processos. A aplicação N-Queens de exemplo do Cilk apresentada na seção 4.1 possui um tempo muito pequeno de execução e, além disso, possui granularidade de trabalho pequena em relação ao tempo de criação de processos MPI-2. Uma vez que essa aplicação não serve para fins de obtenção de um desempenho com MPI-2, foi desenvolvida outra versão do N-Queens. A nova implementação, que está descrita em 3.3, calcula todas as possíveis soluções de uma instância do problema, ao contrário da outra implementação, que retorna apenas a primeira solução encontrada.

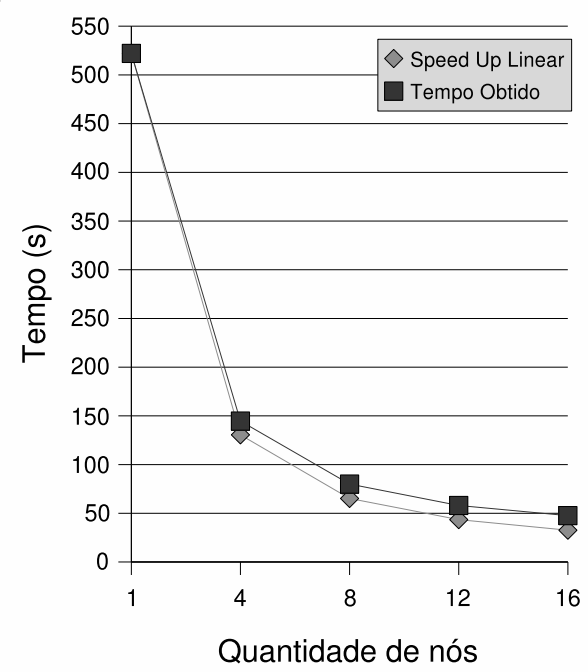


**Figura 4. Tempos do N-Queens ($n = 18$) obtidos no Cluster e previsão linear do Speed Up**

Para este experimento foram utilizadas 5 configurações de ambientes com 1, 4, 8, 12 e 16 nós do cluster. Em cada configuração foi executado o cálculo de soluções possíveis em um tabuleiro com $18 \times 18$ posições. Para obtenção dos melhores tempos foram feitas execuções utilizando diferentes cargas de trabalho por processo, isto é, para cada ambiente, foi encontrado até qual nível da árvore de recursão deve-se criar novos processos. A Figura 4 mostra o tempo médio obtido em 5 execuções com o tamanho de tarefa com o melhor desempenho. O outro valor apresentado é uma estimativa de tempo com *Speed up* linear, calculada a partir da divisão do tempo obtido com um nó pelo número de nós utilizados.

## 5 Considerações finais e trabalhos futuros

Com os resultados dos experimentos foi possível verificar o uso de *D&C* com MPI-2 através da comparação com um ambiente já consolidado e específico para *D&C*. Também foi apresentada uma forma de utilizar recursos disponibilizados ao longo da execução de uma aplicação. Por fim, foi demonstrado o bom desempenho de uma aplicação *D&C* com MPI-2 em ambiente com memória distribuída.

Os resultados obtidos mostram que pode ser interessante utilizar MPI-2 para aplicações *D&C*, que necessitam muito poder computacional. O Cilk, apesar do seu bom desempenho, limita-se a arquiteturas com memória compartilhada. Já o Satin, permite o uso de memória distribuída. No entanto é um ambiente baseado em Java e, apesar dos inúmeros avanços em questões de desempenho, ainda sofre com a sobrecarga decorrente da interpretação, inerente à linguagem Java.

Pode-se perceber que o escalonador Round-Robin melhorou a distribuição de processos, porém essa distribuição pode não ser boa para todas aplicações e ambientes. No caso de aplicações cujos processos tenham carga irregular de trabalho ou em ambientes heterogêneos (ex. grades), é provável que para obter um bom balanceamento seja preciso um mecanismo mais elaborado. Uma possível solução é utilizar um gerenciador de recursos que use outras métricas, além da quantidade de processos. Uma alternativa para aumentar a escalabilidade do escalonador centralizado é torná-lo distribuído, onde ainda pode-se empregar técnicas avançadas para obter um balanceamento de cargas, como por exemplo *Work Stealing*. Outra possibilidade para tornar transparente o balanceamento é integrar o escalonador diretamente à distribuição MPI.

## Referências

[1] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[2] R. Buyya. *High Performance Cluster Computing: Programming and Applications*. Prentice Hall, NJ, USA, 1999.

[3] G. G. H. Cavalheiro, F. Galilée, and J.-L. Roch. Athapascan-1: Parallel Programming with Asynchronous Tasks. In *Proceedings of the Yale Multithreaded Programming Workshop*, Yale, USA, June 1998.

[4] M. Cera, G. Pezzi, M. Pilla, N. Maillard, and P. Navaux. Scheduling dynamically spawned processes in mpi-2. In

*Workshop on Job Scheduling Strategies for Parallel Processing*, Saint-Malo, France, jun 2006.

[5] M. C. Cera, G. P. Pezzi, E. N. Mathias, N. Maillard, and P. O. A. Navaux. Improving the dynamic creation of processes in mpi-2, sep 2006a. accepted to publication in 13th European PVMMPI Users Group Meeting, set, 2006.

[6] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-91-002, NAS Systems Division, Jan. 1991.

[7] D. Dailey and C. E. Leiserson. Using Cilk to write multiprocessor chess programs. *The Journal of the International Computer Chess Association*, 2002.

[8] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

[9] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, Massachusetts, USA, Oct. 1994.

[10] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2 Advanced Features of the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, USA, 1999.

[11] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[12] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*, chapter 17, pages 871–941. Elsevier Science Publishers, 1990.

[13] P. Pacheco. *Parallel Programming With MPI*. Morgan Kaufmann Publishers Inc, 1996.

[14] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339, Dec. 1990.

[15] R. L. R. e. C. S. Thomas H. Cormen, Charles E. Leiserson. *Algoritmos: teoria e prática*. Ed. Campus, 2002.

[16] R. V. van Nieuwpoort et al. Ibis: an efficient java-based grid programming environment. In *Proc. ACM-ISCOPE conference on Java Grande*, pages 18–27, New York, NY, USA, 2002. ACM Press.

[17] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Satin: Efficient parallel divide-and-conquer in java. In *Proc. Euro-Par*, pages 690–699, Munich, Germany, Aug. 2000. Springer.

[18] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, T. Kielmann, and H. E. Bal. Adaptive load balancing for divide-and-conquer grid applications. *Journal of Supercomputing*, 2006.