WALTER FLÁVIO PIMENTA JÚNIOR

HIPERQUICKSORT: UMA ANÁLISE PRÁTICA COM IMPLEMENTAÇÃO EM MPI

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências da disciplina Projeto Orientado, para a obtenção do título de Bacharel em Ciência da Computação.

Orientador

Prof. Jones de Oliveira Albuquerque

LAVRAS
MINAS GERAIS - BRASIL
2002

WALTER FLÁVIO PIMENTA JÚNIOR

HIPERQUICKSORT: UMA ANÁLISE PRÁTICA COM IMPLEMENTAÇÃO EM MPI

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras, como parte das exigências da disciplina Projeto Orientado, para a obtenção do título de Bacharel em Ciência da Computação.

APROVADA em __ de _________ de _____.

Prof. André Luiz Zambalde

Prof. Jones de Oliveira Albuquerque

LAVRAS
MINAS GERAIS - BRASIL

Dedico este trabalho a todas as pessoas
me apoiaram, em especial à minha família,
minha tia Maria Tereza,
que tanto ajudou para que este sonho se con-
cretizasse,
Aos meus amigos e minha namorada Mariana.

LISTA DE FIGURAS ....*viii*

LISTA DE TABELAS ....*ix*

CAPÍTULO 1 *1*

INTRODUÇÃO *1*

1.1 Apresentação do tema 1
1.2 Objetivos 1
1.3 Estrutura do trabalho 2

CAPÍTULO 2 *3*

PRINCÍPIOS BÁSICOS DO PROCESSAMENTO PARALELO *3*

2.1 Introdução 3
2.2 O Advento do Processamento Paralelo 3
2.3 Principais Conceitos do Processamento Paralelo..4
2.4 Paralelismo de Dados x Paralelismo de Controle 6
2.5 Escalabilidade 7
2.6 O Modelo PRAM de Processamento Paralelo 8
2.7 Medidas de Desempenho para Algoritmos Paralelos 11
2.6 Arquiteturas Paralelas em Hardware 12

CAPÍTULO 3 *14*

O ALGORITMO HIPERQUICKSORT *14*

3.1 Introdução 14
3.2 O algoritmo de Ordenação Quicksort 14
3.2.1 Análise 17
3.3 Organização de processadores 18
3.3.1 –A topologia de rede hipercubo 19
3.4 O algoritmo de ordenação Hiperquicksort: Uma versão paralela do Quicksort implementável em rede hipercubo. 20
3.4.1 Um algoritmo PRAM para o Hiperquicksort 24
3.4.2 Hiperquicksort: Análise de Complexidade 26

CAPÍTULO 4 *28*

4.1 O MPI 28
4.1.1 Histórico 28
4.1.2 Funções Básicas 30
4.1.3 Pré Requisitos 31
4.2 – A Implementação 31

4.3 O Ambiente de Execução..33
4.4 Os Testes..34
4.5 Análise de Resultados...36
CAPÍTULO 5 *38*
CONCLUSÕES *38*

LISTA DE FIGURAS

Figura 2.1 – Paralelismo Lógico..5
Figura 2.2 – Paralelismo Real 6
Figura 2.3 – Modelo PRAM de processamento paralelo.....................................9
Figura 2.4 – Tempo de comunicação x Tempo de computação..........................12
Figura 3.1 – Procedimento Partição...16
Figura 3.2 – Funcionamento do Procedimento Quicksort..................................17
Figura 3.3 – Procedimento Quicksort...17
Figura 3.4 – Um Hipercubo de Dimensão quatro...20
Figura 3.5 - Comunicação entre os processadores em um hipercubo de três dimensões...23
Figura 3.6 – Exemplo da execução do algoritmo Hiperquicksort em um hipercubo de duas dimensões...24
Figura 3.7 – Um algoritmo PRAM para o Hiperquicksort.................................25
Figura 4.1: Programa MPI para a troca de valores entre dois processos............30
Figura 4.2 – Chamada à função MPI_Barrier( )..33

LISTA DE TABELAS

Tabela 4.1 – Vetor de 4.096 elementos 35
Tabela 4.2 – Vetor de 8.192 elementos 35
Tabela 4.3 – Vetor de 16.384 elementos 36
Tabela 4.4 – Vetor de 32.768 elementos 36

CAPÍTULO 1

INTRODUÇÃO

1.1 Apresentação do tema

O aumento da quantidade de dados a serem processados em conjunto com a necessidade de obtenção do resultado do processamento dos mesmos em um tempo cada vez menor, ocasionou um aumento significativo no custo computacional. Computadores seqüenciais mostraram-se computacionalmente ineficazes diante da realização desta tarefa.

Tendo em vista esta ineficácia, a alternativa encontrada foi a programação paralela e distribuída, na qual diversos processadores podem processar simultaneamente um problema, tornando viável sua implementação.

Conforme visto em [4], o aumento de desempenho é a mais importante razão para o uso do processamento paralelo, tendo em vista que este tipo de processamento pode reduzir consideravelmente o tempo de resolução de um problema.

Desde o surgimento da programação paralela e distribuída, diversos mecanismos foram desenvolvidos para a realização da comunicação entre os processadores responsáveis pela execução do programa paralelo, como os mecanismos de trocas de mensagens, assunto deste trabalho.

Dentre os mecanismos de trocas de mensagens, é importante citar o MPI (Message Passing Interface), a ser abordado nessa monografia, como o mais amplamente utilizado atualmente.

1.2 Objetivos

O objetivo deste trabalho é realizar um estudo prático do algoritmo Hiperquicksort, um clássico do processamento paralelo, seguido da realização de

testes do algoritmo a fim de se comprovar o ganho de desempenho obtido pelo mesmo.

1.3 Estrutura do trabalho

A presente monografia esta organizada da seguinte forma: O Capítulo 1 apresenta uma introdução ao tema; O Capítulo 2 apresenta os principais conceitos de processamento paralelo, tanto em hardware quanto em software; O Capítulo 3 apresenta os algoritmo Quicksort e Hiperquicksort, com suas respectivas análises de complexidade; O Capítulo 4 apresenta a implementação MPI do algoritmo Hyperquiksort e sua análise de resultados. Por fim, no Capítulo 5, apresentaremos as conclusões referentes ao presente trabalho.

CAPÍTULO 2

PRINCÍPIOS BÁSICOS DO PROCESSAMENTO PARALELO

2.1 Introdução

Neste capítulo iremos explicar os principais conceitos do processamento paralelo, além de fazermos uma referência histórica do mesmo. Serão estudados os diferentes métodos de processamento paralelo, a medida de qualidade de um algoritmo paralelo e o modelo PRAM de programação paralela.

2.2 O Advento do Processamento Paralelo

A história da programação paralela seguiu os mesmos estágios de outras áreas experimentais da Ciência da Computação. Segundo [01], a programação paralela se originou nos anos 60, tendo como motivação, a invenção de unidades de hardware denominadas *controladoras de dispositivos.* Estes dispositivos permitem que as operações de ES (Entrada e Saída) sejam realizadas concorrentemente com as contínuas execuções de instruções de programas realizadas pelo processador central.

Pouco após a invenção das controladoras de dispositivo, engenheiros de hardware criaram o projeto de máquinas multi processadas. Porém, foi necessária uma espera de mais de vinte anos para que os computadores paralelos fizessem a transição do laboratório para o mercado de trabalho [03]. Segundo [01], esta demora se explica pelo alto preço dos computadores multi processados até então.

Em meados da década de 80, deu-se o advento dos computadores paralelos micro-processados comerciais.

Hoje em dia todas as grandes máquinas são multi processadas, sendo que as maiores possuem centenas de processadores, e são chamadas de MPP's – Massively Parallel Processors ou Processadores Paralelos Massivos.

Segundo [4], a mais importante razão para o uso do processamento paralelo é o aumento do desempenho, pois este tipo de processamento pode representar uma redução considerável no tempo de resolução de um problema. Outra importante razão que faz jus ao uso da programação paralela é que alguns problemas são de natureza paralela e sua implementação seqüencial não produz resultados satisfatórios [MONTE-MOR].

2.3 Principais Conceitos do Processamento Paralelo

A maioria dos computadores atuais de grande desempenho trata concorrência. Porém, segundo [3], não se deve chamar a todos os computadores modernos de computadores paralelos. O tratamento de concorrência em muitas máquinas é simulado e transparente ao usuário. Portanto, a seguinte nomenclatura é utilizada:

Processamento Paralelo é o processamento de informação que enfatiza a manipulação concorrente de dados relativos a um ou mais processos pertencentes a um único programa [3]. Deve-se aqui, explicitar a diferença entre processo e programa: Programa pode ser entendido como sendo o código fonte, de natureza estática, enquanto que o processo representa os programas em execução, possuindo natureza dinâmica, sendo que o mesmo programa pode disparar vários processos independentes.

Segundo [Ducan], citado por [JULIANO], uma **arquitetura paralela** fornece uma estrutura explícita e de alto nível para o desenvolvimento de soluções através do processamento paralelo, utilizando-se para isso de

múltiplos processadores, estes simples ou complexos, que cooperam para resolver problemas através da execução concorrente.

Em computadores seqüenciais existe o que chamamos de concorrência lógica, ou seja, o tempo de processamento é compartilhado entre os processos concorrentes, de modo que apenas um processo é executado por vez. O usuário, por sua vez, tem a impressão de um processamento paralelo, já que o escalonamento dos processos não é visível. A Figura 2.1 ilustra essa situação:

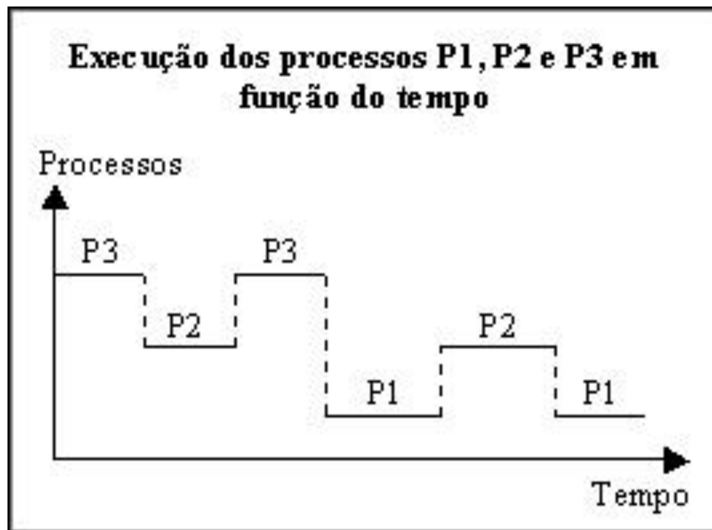

Figura 2.1 – Paralelismo Lógico

Em computadores paralelos, todavia, existe uma concorrência real entre os processos já que cada processador executa simultaneamente um processo distinto. É importante salientar que ao dividirmos as tarefas entre os diferentes processadores, temos uma redução no tempo de processamento (speedup), e a divisão destas tarefas entre seus respectivos processadores fica a cargo do programador. A Figura 2.2 demonstra a execução de processos com o paralelismo real:

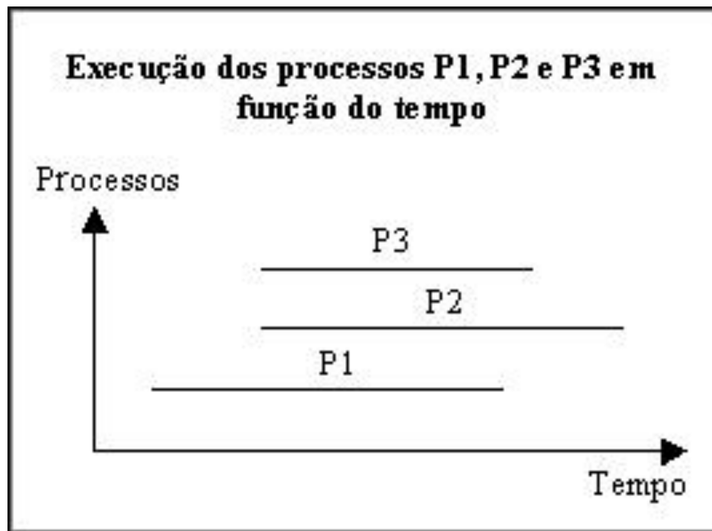

Figura 2.2 – Paralelismo Real

Um **supercomputador** é um computador de uso geral capaz de resolver problemas em um tempo computacionalmente muito rápido [3], se comparado aos outros computadores.

2.4 Paralelismo de Dados x Paralelismo de Controle

Pode-se classificar a programação paralela entre dois paradigmas:

O **paralelismo de dados** que representa o uso de múltiplas unidades para se aplicar a mesma operação simultaneamente em um dado conjunto de elementos. Segundo [3], K unidades de processamento adicionais geram um aumento de vazão de K vezes no sistema. Por vazão indica-se o número de resultados obtidos por ciclo de tempo. A execução deste tipo de algoritmo pode ser verificada, por exemplo, em algoritmos paralelos de multiplicação de matrizes.

O **paralelismo de controle,** no qual, diferentemente do paralelismo de dados onde o paralelismo é atingido através de diversas unidades de processa-

mento executando uma única instrução, atinge-se o paralelismo através da aplicação de diferentes operações a diferentes conjuntos de dados simultaneamente. Conforme [3], citado por [MONTE-MOR], o fluxo de dados sobre este processo pode ser arbitrariamente complexo. No paralelismo de controle, a computação é dividida em passos, chamados segmentos ou estágios, que são distribuídos entre os processadores. Cada segmento realiza uma parte do processamento, e pode ser possível que a entrada de um segmento seja a solução gerada na saída do segmento anterior.

2.5 Escalabilidade

Um algoritmo é escalável se o nível de paralelismo aumenta pelo menos linearmente com o tamanho do problema [3]. Uma arquitetura é escalável se o desempenho de cada processador não se altera quando se aumenta o número de processadores [3]. Como regra geral, pode-se dizer que tanto a escalabilidade algorítmica quanto a arquitetural são importantes, visto que, se pode resolver problemas maiores por meio da aquisição de um computador paralelo com mais processadores.

Os algoritmos baseados no paralelismo de dados são mais escaláveis que os baseados no paralelismo de controle, visto que, no caso do paralelismo de controle o nível de paralelismo é geralmente uma constante, enquanto no paralelismo de dados, o nível de paralelismo é uma função crescente com o aumento do tamanho do problema.

2.6 O Modelo PRAM de Processamento Paralelo

Conforme visto em [3], o modelo PRAM (Parallel Random Acess Machine) permite aos programadores tratar o poder computacional como um recurso ilimitado. Apesar de ser o mais conhecido modelo de processamento paralelo ele é não é realista, pois ignora a complexidade de comunicação entre os processadores.

Como a complexidade de comunicação é ignorada, os criadores de algoritmos paralelos podem focalizar suas atenções ao desenvolvimento do paralelismo inerente à computação[3].

Segundo [(Fortune and Wyllie 1978)] citado por [3], o modelo PRAM consiste de uma unidade de controle, uma memória global e um número ilimitado de processadores, cada qual com sua memória local, conforme ilustrado na Figura 2.3.

Uma computação PRAM inicia com a entrada armazenada na memória global e apenas um processador ativo. Durante cada ciclo da computação um processador ativo pode ler um valor da memória local ou global, realizar uma operação sobre este valor e escrevê-lo na memória local ou global. Alternativamente, ele pode ativar outro processador. Cada processador ativo deve executar a mesma instrução em diferentes locais da memória. A computação chega ao fim quando o último processador termina a execução.

É importante salientar que como um algoritmo PRAM começa sua execução com apenas um processador ativo, ele é dividido em duas partes: na primeira fase é realizada a ativação de um número suficiente de processadores e na segunda fase é realizado o processamento.

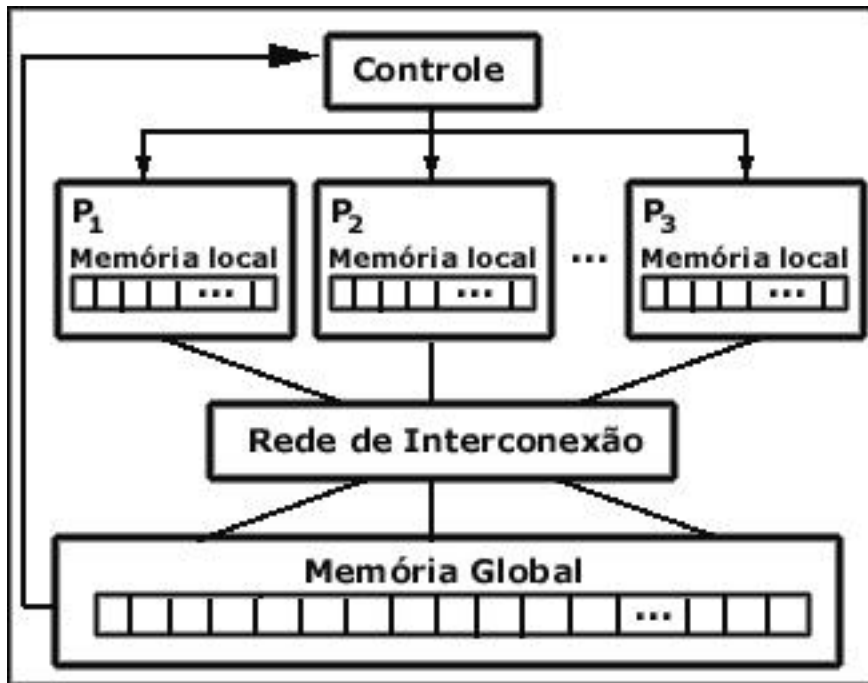

Figura 2.3 – Modelo PRAM de processamento paralelo

Segundo [3], o custo de uma computação PRAM é o produto da complexidade de tempo paralelo e o número de processadores envolvidos, isto é, se, por exemplo, um algoritmo que tenha complexidade de tempo O(log n), usando p processadores possui complexidade O (p log n).

Um algoritmo PRAM de custo ótimo é um bom ponto de partida para o desenvolvimento de um algoritmo paralelo real eficiente. Por algoritmo PRAM de custo ótimo entende-se um algoritmo PRAM cujo custo está na mesma classe de complexidade do algoritmo seqüencial.

Um dos obstáculos à paralelização de um método qualquer são os problemas denominados conflitos de escrita e leitura, que ocorrem quando dois ou mais processadores tentam ler ou escrever na mesma posição de memória simultaneamente.

Conforme [3], diversos modelos PRAM diferem no modo de tratamento dos conflitos de leitura e escrita na memória global. Cita-se:

EREW – Exclusive Read Exclusive Write, ou seja, leitura e escritas exclusivas. Neste modelo, não são permitidas leituras ou escritas concorrentes. Apenas um processador pode acessar a região de memória por vez, tanto para a escrita, quanto para a leitura.

CREW – Concurrent Read Exclusive Write, ou seja, leitura concorrente e escrita exclusiva. Dois processadores podem acessar a mesma região de memória simultaneamente caso o objetivo seja a leitura. Neste modelo a escrita concorrente ainda não é tratada.

CRCW – Concurrent Read Concurrent Write. Este é o mais completo modelo PRAM dentre os citados, já que trata tanto leitura quanto escrita concorrente. Segundo [3], existem diversos modelos PRAM CRCW, dentre os quais citamos: **CRCW COMMON,** no qual todos os processadores escrevendo na mesma região de memória global devem estar escrevendo o mesmo valor; **CRCW ARBITRARY,** onde, quando múltiplos processadores tentam escrever na mesma região de memória um deles é arbitrariamente escolhido e escreve seu valor; e o modelo **CRCW PRIORITY**, de acordo com o qual os processadores possuem um índice de prioridade e ao ocorrer a múltipla tentativa de escrita no mesmo local da memória, o processador com o menor índice escreve seu valor na memória global.

De acordo com [3], o modelo EREW é o mais fraco dentre os citados. Claramente, é possível a um PRAM modelo CREW, executar qualquer algoritmo EREW na mesma quantidade de tempo, simplesmente descartando a facilidade de leitura concorrente. Similarmente, um PRAM CRCW pode executar qualquer algoritmo CREW na mesma quantidade de tempo. Segundo (Eckstein [1789] e Vishkin [1983]) citado por QUINN em [3], uma máquina PRAM

PRIORITY com p processadores pode ser simulada em uma máquina EREW PRAM com um incremento de O (log p) na complexidade de tempo.

2.7 Medidas de Desempenho para Algoritmos Paralelos

Em se tratando de algoritmos paralelos, as medidas de qualidade ficam a cargo de dois fatores: *Speedup* e *eficiência.*

Segundo [3], o ganho de desempenho obtido com a paralelização do algoritmo é chamado de *speedup*, que é a razão entre o tempo de execução do algoritmo seqüencial mais eficiente e o tempo de execução do algoritmo paralelo correspondente utilizando p processadores.

Embora teoricamente o *speedup* ótimo seja diretamente proporcional ao número de processadores utilizados, na prática esta provou ser uma hipótese errada, visto que o *speedup* não se mostrou proporcional ao número de processadores. Apesar disso, segundo [3], um *speedup* acima do linear, ou superlinear é possível, desde que a escolha do algoritmo seja feita antes da escolha do problema e que o algoritmo paralelo trate especificamente alguns casos tratados genericamente pelo algoritmo seqüencial.

Segundo [3], a queda no *speedup* ao se aumentar muito o número de processadores pode ser explicada pela sobrecarga (overhead) de tempo de comunicação entre os processadores – fato ignorado pelo modelo PRAM – e a existência de tarefas inerentemente seqüenciais, mesmo em algoritmos paralelos.Com o aumento do número de processadores o tempo de computação torna-se cada vez menor, ao passo que o excesso de tempo de comunicação torna o processamento final mais lento.O tempo de computação é inversamente proporcional ao número de processadores utilizados enquanto que o tempo de comuni-

cação entre os processadores cresce linearmente com o aumento dos mesmos. Conforme [3], atingido o limite, ou seja, o *speedup* máximo, o aumento do tempo de comunicação é maior que a diminuição do tempo de computação e o tempo total começa a aumentar. Na Figura 2.4, temos um gráfico segundo o exemplo da Figura 2.4 em [3] que ilustra essa situação.

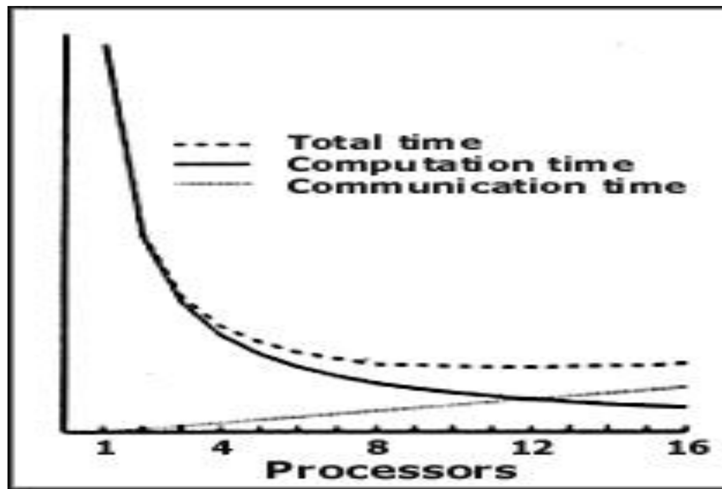

Figura 2.4 – Tempo de comunicação x Tempo de computação

Podemos definir a eficiência de um algoritmo paralelo executado em uma máquina com p processadores como sendo a razão entre o *speedup* alcançado e o número de processadores utilizados.

2.6 Arquiteturas Paralelas em Hardware

As arquiteturas paralelas se podem distinguir-se em dois grandes grupos: Um destes grupos é a **Arquitetura de memória compartilhada,** na qual os processadores estão conectados por algum tipo de rede de comunicação interna e

os processos dividem a memória principal, embora cada qual tenha sua memória cache particular, de acesso exclusivo.

Em maquinas multi processadas pequenas – possuindo de 2 a aproximadamente 30 processadores – a rede de comunicação entre os processadores pode ser um implementada por um barramento de memória. Este modelo de arquitetura é denominado UMA (Uniform Memory Access – Acesso Uniforme à Memória), visto que, existe um tempo uniforme de acesso à memória entre cada processador e cada endereço de memória.

Em máquinas multi processadas de grande porte, a rede de comunicação dos processadores é composta por um conjunto de *switches* e memórias estruturados em árvore. Cada processador tem acesso tanto à memória que está próxima quanto à memória que está longe. A estruturação da memória em árvore evita o congetionamento que poderia ocorrer se muitos processadores acessassem o barramento simultaneamente [1]. Esta estrutura, por sua vez, não possui um tempo uniforme de acesso à memória e é chamada de NUMA (Non-Uniform Memory Acess – Acesso não Uniforme à Memória).

O outro grupo é a **Arquitetura de memória distribuída,** na qual os processadores também estão conectados por uma rede de comunicação interna possuindo, porém tanto memória cache particular quanto memória principal própria.

CAPÍTULO 3

O ALGORITMO HIPERQUICKSORT

3.1 Introdução

Segundo [3], a ordenação é uma das atividades mais freqüentemente realizadas em computadores seriais. Muitos algoritmos incorporam uma ordenação a fim de permitir um acesso mais coerente aos dados posteriormente. A Ordenação tem uma importância adicional para desenvolvedores de algoritmos paralelos; ela é freqüentemente usada para a realização de permutações de dados em computadores de memória distribuída [3]. Essas operações de movimentação de dados podem ser usadas para resolver problemas na teoria de grafos, geometria computacional, e processamento de imagens em tempo ótimo. Uma máquina PRAM CRCW não trivial pode ordenar n elementos em tempo constante, desprezando-se o tempo necessário à ativação de n*n processadores. Neste capítulo será feita uma análise do algoritmo de ordenação Quicksort, base para a implementação do algoritmo Hiperquicksort, estudo dessa monografia, além de uma visão geral sobre a topologia de redes hipercubo, sobre a qual é implementado o Hiperquicksort.

3.2 O algoritmo de Ordenação Quicksort

O Quicksort é um algoritmo de ordenação freqüentemente utilizado em computadores seriais, devido ao seu comportamento assintoticamente ótimo em O (n log n) para o caso médio [(Bease 1978)], citado por [3].

Segundo [10], o Quicksort é o algoritmo de ordenação interna mais rápido de que se tem conhecimento. O algoritmo foi inventado por C.A.R. Hoare em 1960, ainda como estudante na Universidade de Moscou. Dois anos mais tarde, em 1962, o algoritmo foi publicado, após uma série de refinamentos.

O Quicksort é um algoritmo recursivo, baseado na estratégia da divisão e conquista. Sua idéia é particionar o problema da ordenação de uma lista de *n* elementos em dois problemas menores. A seguir, os problemas menores são ordenados independentemente e os resultados gerados são combinados a fim de se obter a solução do problema original.

A parte mais delicada do método Quicksort corresponde ao processo de partição, o qual tem que rearranjar o vetor A[*Esq...Dir*] através da escolha arbitrária de um item *x* do vetor chamado normalmente de *pivô*, de maneira que ao final do procedimento o vetor esteja particionado em duas partes: A esquerda, com valores menores ou iguais a *x,* e a direita, com valores maiores que *x*.

Este processo pode ser descrito pelo algoritmo:

1. Escolha arbitrariamente um item do vetor e coloque-o em *x;*
2. Percorra o vetor à partir da esquerda até que um item A[i] > x é encontrado; analogamente, percorra o vetor a partir da direita até um item A[j] <= *x* é encontrado.
3. Como os dois itens A[i] e A[j] estão fora de lugar no vetor final, troque-os de lugar.
4. Continue este processo até que os apontadores *i* e *j* se cruzem em algum ponto do vetor.

A Figura 3.1 ilustra uma implementação do procedimento Partição.

```
procedure Partição (Esq, Dir : Índice; var i, j : Índice);
var x, w : Item;
begin
  i := Esq; j := Dir;
  x := A[(i+j) div 2]; {— obtém o pivô x —}
  repeat
    while x.Chave > A[i].Chave do i := i+1;
    while x.Chave < A[j].Chave do j := j−1;
    if i ≤ j
    then begin
           w := A[i]; A[i] := A[j]; A[j] := w;
           i := i+1; j := j−1;
           end;
  until i > j;
end;
```

Figura 3.1 – Procedimento Partição

No procedimento da Figura 3.1, Esq e Dir são apontadores para delimitar o subvetor dentro do vetor original *A*, o qual deve ser particionado. Os índices *i* e *j* retornam as posições finais das partições, onde A[Esq], A[Esq +1], ..., A[j] são menores ou iguais ao pivô *x,* e A[i], A[i+1] e A[Dir] são maiores ou iguais a *x*. O vetor A é considerado global ao procedimento partição.

Podemos observar que o anel interno do procedimento partição consiste apenas em incrementar um apontador e comparar um item do vetor com um valor fixo em *x*. Este anel é extremamente simples, razão pela qual o algoritmo Quicksort é tão rápido [10].

Após obter dois sub vetores depois da aplicação do procedimento partição, cada pedaço é ordenado recursivamente. O refinamento final do procedimento Quicksort é apresentado na Figura 3.2. O procedimento Ordena é recursivo e o vetor A é global ao procedimento partição.

```
procedure Quicksort (var A : Vetor);
{— Entra aqui o procedimento partição —}
  procedure Ordena (Esq, Dir : Índice);
  var i, j : Índice;
  begin
    partição (Esq, Dir, i, j);
    if Esq < j then Ordena (Esq, j);
    if i < Dir then Ordena (i, Dir);
  end;
begin
  Ordena (1, n);
end;
```

Figura 3.2 – Procedimento Quicksort

A Figura 3.3 demonstra o funcionamento de um vetor de seis elementos após cada chamada recursiva do procedimento ordena, onde o elemento *pivô* é mostrado em negrito.

Chaves iniciais:	*O*	*R*	*D*	*E*	*N*	*A*
1	*A*	**D**	*R*	*E*	*N*	*O*
2	**A**	*D*				
3			**E**	*R*	*N*	*O*
4				**N**	*R*	*O*
5					*O*	**R**
	A	*D*	*E*	*N*	*O*	*R*

Figura 3.3 – Funcionamento do Procedimento Quicksort

3.2.1 Análise

Pior caso:

Uma importante característica do quicksort é sua ineficiência para vetores já ordenados quando a escolha do pivô é inadequada [10]. O pior caso possível seria e escolha sistemática dos extremos de um arquivo já ordenado. Neste

caso, as partições são de tamanhos completamente diferentes e função ordena será recursivamente chamada n vezes, eliminando apenas um item em cada chamada. Essa situação é extremamente indesejada, visto que o número de comparações passa a cerca de $\frac{n^2}{2}$ e a pilha necessária á execução das chamadas recursivas passa a ter tamanho n.

Melhor caso:

O melhor caso para o algoritmo ocorre com a partição do arquivo em duas metades iguais. Logo, $C(n) = 2C(\frac{n}{2}) + n$ onde $C(\frac{n}{2})$ ilustra o custo de ordenar uma das metades e n é o custo de cada item. Resolvendo esta recorrência chegamos a $C(n) \leq 1{,}4\ n \log n$, de onde ocorre que, em média, o tempo de execução do algoritmo Quicksort é $O(n \log n)$.

3.3 Organização de processadores

Um assunto a ser tratado em processamento paralelo é a organização dos processadores, isto é, como está estruturada a rede de processadores e de que maneira eles se comunicam. Uma organização de processadores pode ser representada por um grafo no qual os nós representam os processadores e as arestas representam os caminhos de comunicação entre pares de processadores. Conforme [3], a avaliação das diversas variações de organização de processadores é realizada de acordo com quatro critérios. Estes critérios são:

1. Diâmetro. O diâmetro de uma rede é a maior distância entre dois nós (processadores). Menores diâmetros são melhores, pois, altos diâmetros pioram significamente a complexidade do algoritmo paralelo, quando o mesmo requer muitas comunicações entre pares arbitrários de processadores.

2. Largura de bisecção da rede. A largura de bisecção de uma rede é o menor número possível de arestas que devem ser removidas a fim de se dividir a rede em duas metades. Uma maior largura de bisecção é desejada, pois, em algoritmos que requerem uma grande quantidade de movimentação de dados, o tamanho do conjunto de dados dividido pela largura de bisecção piora a complexidade do algoritmo paralelo.
3. O número de arestas por nó. É melhor que o número de arestas por nó seja uma constante independente do tamanho da rede, pois desta maneira, torna-se mais fácil portar o algoritmo para redes com maior número de nós.
4. Tamanho da maior aresta. Por razões de escalabilidade, é melhor que os nós e arestas de uma rede de processadores possam ser **"laid out"** em um espaço tri-dimensional de modo que o maior comprimento de uma aresta seja uma constante independente do tamanho de rede.

A seguir, discutiremos a organização de processadores em hipercubo, sobre a qual é implementado o algoritmo Hiperquicksort, de acordo com os critérios de avaliação vistos acima.

3.3.1 –A topologia de rede hipercubo

Uma rede hipercubo, é uma rede formada por 2^k nós formando um hipercubo k-dimensional. Os nós são numerados de 0 a $2^k - 1$ e dois nós são adjacentes se, e somente se, diferem entre si por exatamente um bit. A Figura 3.4 ilustra um hipercubo de quatro dimensões.

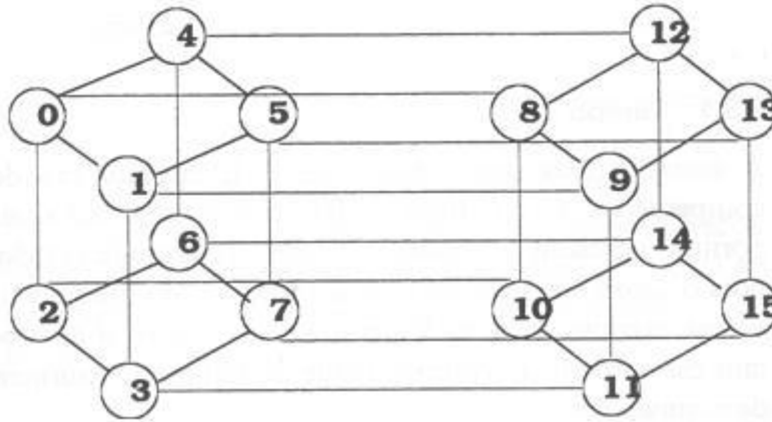

Figura 3.4 – Um Hipercubo de Dimensão 4

Conforma visto em [3], o diâmetro de uma rede hipercubo com 2^k nós é k e a largura de bisecção desta rede é de 2^{k-1}. A organização de redes em hipercubo consegue um baixo diâmetro e uma grande largura de bisecção (exponencial ao numero de nós) ao custo do número de arestas por nó e do tamanho da maior aresta. O número de arestas por nó é dado por (*k - o logaritmo do número de nós da rede)* e o comprimento da maior aresta aumenta de acordo com o aumento do numero de nós da rede.

3.4 O algoritmo de ordenação Hiperquicksort: Uma versão paralela do Quicksort implementável em rede hipercubo.

Conforme visto anteriormente, o Quicksort é um algoritmo que utiliza a estratégia da divisão e conquista. Não é difícil notar que depois da primeira partição de uma lista em duas sub listas são gerados dois novos problemas que podem ser resolvidos simultaneamente. Conforme [3], desenvolver algoritmos paralelos torna-se mais fácil quando existe um algoritmo PRAM de custo ótimo para o problema. Algoritmos paralelos de ordenação têm se mostrado um desafio

aos desenvolvedores. Alguns, como o chamado Bitonic Merge-sort, possuem custo O (n log2 n), maior que o custo do melhor algoritmo seqüencial, o que o torna impraticável. Por esta razão, o algoritmo Quicksort, melhor algoritmo de ordenação de propósito geral, foi tomado como base para a implementação do algoritmo Hiperquicksort, este sim um eficiente algoritmo paralelo de ordenação.

O funcionamento do Hiperquicksort é como se segue: Dada uma lista de valores inicialmente distribuídos entre os processadores de uma rede hipercubo, pode-se definir a lista como ordenada quando (1) todas as listas de valores dos processadores estejam ordenadas, e (2) o valor do último elemento na lista de *Pi* (processador i) é menor ou igual ao primeiro elemento na lista de *P i+1,* para $0 \leq i \leq p-2$.

A fim de se desenvolver um algoritmo paralelo eficiente, cada processador deve ordenar sua lista de valores utilizando um algoritmo seqüencial eficiente. Deve-se então, utilizar um algoritmo eficiente de comunicação para a geração da solução final partindo das soluções parciais.

Na primeira fase do Hiperquicksort, cada processador usa o Quicksort para ordenar sua lista local de valores. Neste ponto, cada processador tem uma lista ordenada de valores, satisfazendo a condição (1) vista anteriormente, mas não a condição (2). O Hiperquicksort é um algoritmo recursivo, baseado, como o Quicksort, na estratégia de divisão e conquista e faz uso desta estratégia para preencher a condição (2). A segunda fase do algoritmo se encarrega justamente de cumprir essa condição. Durante cada passo desta fase do algoritmo, o hipercubo é dividido em dois subcubos. Cada processador envia valores para o seu parceiro no outro subcubo e depois realiza uma agregação entre os valores que o processador recebeu de seu parceiro e os valores que ele manteve em sua lista. Esta operação é denominada "*spliti-and-merge*" Conforme [3], o efeito desta operação de "*spliti-and-merge*" é a divisão de um hipercubo de valores ordena-

dos em dois hipercubos, tal que, cada processador tenha uma lista ordenada de valores e o maior valor do hipercubo inferior seja menor ou igual que o menor valor do hipercubo superior. Após *d* operações de "*spliti-and-merge*", o hipercubo original de *d* dimensões estará dividido em 2^d hipercubos de um único processador (dimensão zero) e a condição (2) estará satisfeita. Relembrando que cada processador possui uma lista ordenada de elementos as operações de "split-and-merge" se realizam como se segue: Um processador específico do hipercubo de *d* dimensões calcula um elemento pivô (como no Quicksort seqüencial) e realiza uma operação de *broadcast* deste pivô para os demais 2^{d-1} processadores do hipercubo. Cada processador usa este pivô para dividir sua lista local em duas sub-listas, Low-List e High-List, de modo que na Low_List estarão todos os valores menores ou iguais ao pivô, ao passo que na High-List estarão todos os valores maiores que o pivô. Cada processador *Pi* na metade inferior do hipercubo envia sua High-List – valores maiores que o pivô – ao seu parceiro na metade superior do hipercubo, o processador *Pi* * 2^{d-1} (onde * denota uma operação binária de ou exclusivo). Cada processador *Pi* na metade superior do hipercubo envia sua Low-List – valores menores ou iguais ao pivô – para seu parceiro na metade inferior do hipercubo, o processador *Pi* * 2^{d-1}. Neste ponto, todos os valores menores ou iguais ao pivô estão na metade inferior do hipercubo enquanto os valores maiores que pivô estão na metade superior do hipercubo (de dimensão *d-1).* A Figura 3.5 ilustra a comunicação entre os processadores para um hipercubo de três dimensões. A Figura 3.6 demonstra o funcionamento passo-a-passo do algoritmo para um hipercubo de duas dimensões.

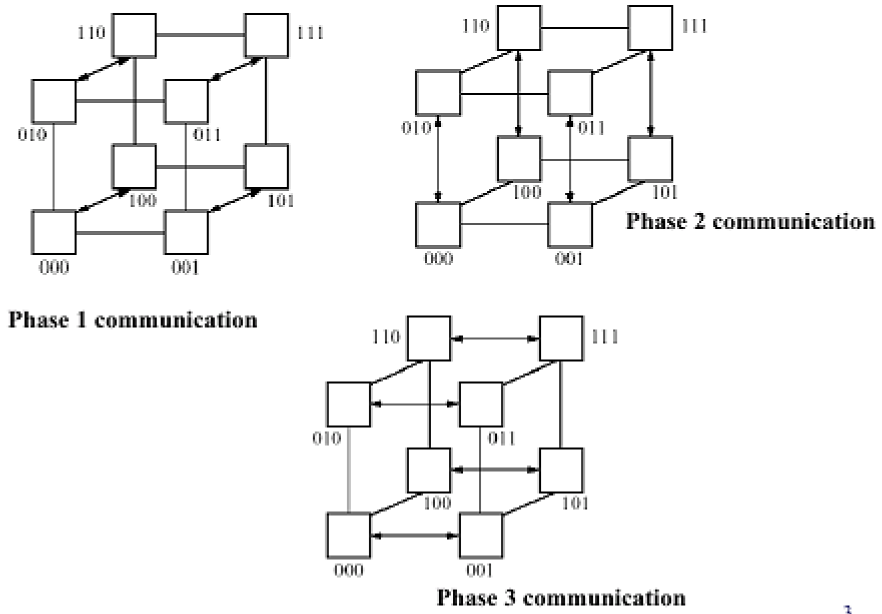

Figura 3.5 - Comunicação entre os processadores em um hipercubo de três dimensões

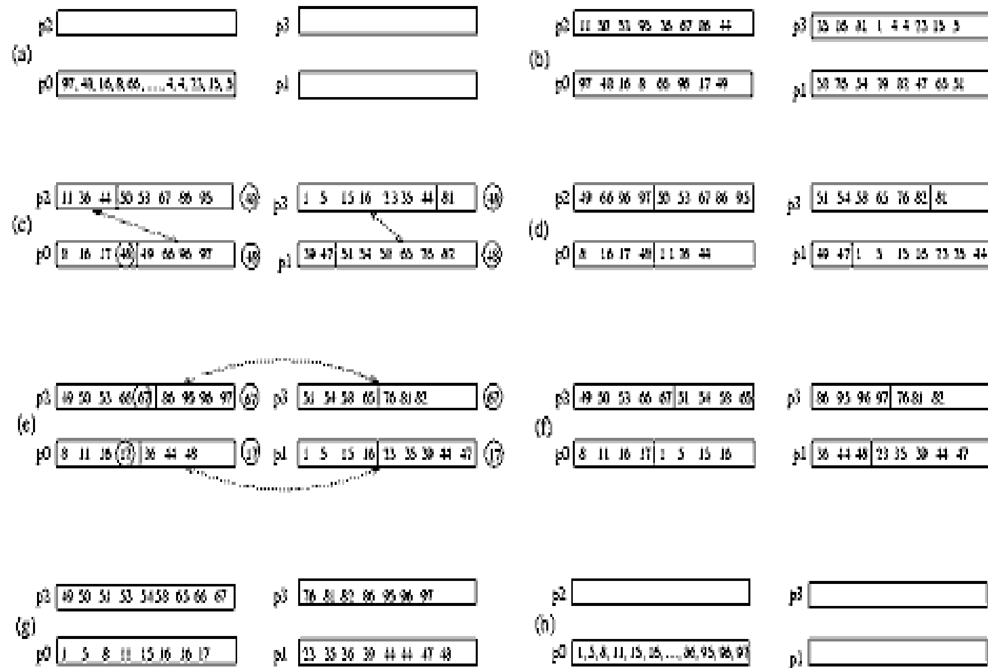

Figura 3.6 – Exemplo da execução do algoritmo em um hipercubo de duas dimensões.

3.4.1 Um algoritmo PRAM para o Hiperquicksort

Segundo [3], o primeiro passo para se implementar um algoritmo paralelo eficiente é ter à mão um algoritmo PRAM do mesmo. O algoritmo PRAM para a implementação realizada neste trabalho é como se segue:

```
                  HYPERQUICKSORT (HYPERCUBE MULTICOMPUTER)

Global  n                  {Número inicial de elementos por processador}
        d                  {Dimensão do Hypercubo}
        i                  {Número da dimensão do atual hypercubo}
Local   processor_Id       {Indentificador único do processador}
        partner            {Processo partner para troca de valores}
        root               {Processo root do hypercubo corrente}
        pivô               {Pivo calculado pelo root do hypercubo}

begin
  For all Pj do
    Ordene n valores usando o algoritmo quicksort sequencial
     if (d > 0) then
       For i = d decrementando até 1 do
         if (processor_id = root) then
           pivô = elemento médio da lista ordenada pertencente ao processador root
         end_if
         Processador root  distribui pivô no hypercubo atual
         Usa spliter para calcular High_List e Low_List
         Calcula partner
         if (processor_id > partner) then
           Envia Low_List para partner
           Recebe High_List do partner
          else {processor_id < partner}
            Envia High_List para partner
            Recebe Low_List do partner
          end_if
          Faça um merge das duas listas e forma uma nova lista ordenada
        end_for
      end_if
    end_for
end
```

Figura 3.7 – Um algoritmo PRAM para o Hiperquicksort

O algoritmo PRAM que descreve o funcionamento do Hiperquicksort ilustrado na figura 3.7 pode ser resumido em quatro fases:

1. Cada processador ordena sua lista local utilizando para isso o quicksort seqüencial;

2. Um processador dedicado calcula o pivô e realiza uma operação de broadcast, ou seja, distribui o pivô a todos os processadores do hipercubo;
3. Os processadores pertencentes ao subcubo inferior enviam seus elementos maiores que o pivô (High List) para seus respectivos parceiros no hipercubo superior. Processadores situados no hipercubo superior enviam seus valores menores ou iguais ao pivô (Low List) para seus respectivos parceiros no hipercubo inferior;
4. Cada processador realiza uma operação de *merge* entre a lista que recebeu e a lista que manteve, a fim de obter uma lista ordenada;

É importante salientar que os passos dois, três e quatro estão localizados dentro de um *anel (loop)* e são repetidos *d* vezes, onde *d* representa o número de dimensões do hipercubo.

3.4.2 Hiperquicksort: Análise de Complexidade

Suponhamos que no início da ordenação, cada processador tenha uma lista com *n* valores. A complexidade de tempo esperada para o primeiro passo do algoritmo (o Quicksort) é de $O(n \log n)$ [3]. Assumindo que cada processador mantém $\frac{n}{2}$ elementos e envia ao parceiro outros $\frac{n}{2}$ elementos, o número esperado de comparações necessárias para agregar as duas listas em uma lista ordenada é $O(n)$. Se o algoritmo realiza operações de "split-and-merge" em hipercubos de $d, d-1, d-2, ..., 1$ dimensões, o número total de operações de "split-and-merge" esperadas é $O(dn)$, onde d denota o número de dimensões do hipercubo, e o número total de operações realizadas é $O(n(\log n + d))$.

Consideremos α a latência da mensagem e β o tempo necessário para se transferir um valor de um processador para outro processador adjacente. Em um cubo d-dimensional, o tempo de comunicação necessário para se transmitir o pivô para os demais processadores do cubo é $d\,(\alpha + \beta)$. Assumindo que cada processador envia metade de seus valores, o tempo necessário para enviar $\frac{2}{n}$ valores ordenados e receber $\frac{2}{n}$ valores ordenados do processador (parceiro) adjacente é $2\alpha + n\beta$. O tempo esperado de comunicação na fase de "split-and-merge" é:

$$\sum_{i=1}^{n} (\mathrm{i}\,(\alpha + \beta) + 2\alpha + \mathrm{n}\beta)\ .$$

Como na fase inicial do algoritmo (o Quicksort) não são necessárias comunicações entre processadores, este é o tempo esperado para a execução total do algoritmo.

CAPÍTULO 4

UMA IMPLEMENTAÇÂO EM MPI PARA O ALGORITMO HIPERQUICKSORT

Introdução

Neste capítulo, estaremos descrevendo a fase de implementação do algoritmo Hiperquicksort, utilizando como ferramenta, a biblioteca de troca de mensagens MPI (Message Passing Interface – Interface de Troca de Mensagens). Inicialmente apresentaremos um pouco da história da ferramenta, depois seus pré-requisitos e principais funções. Com base nestas informações passaremos para a descrição da implementação do algoritmo, seguida da análise de resultados dos testes realizados. A análise de resultados pode ser encarada como uma medida de desempenho tanto para o algoritmo Hiperquicksort quanto para o MPI, já que segundo [7], a ordenação é uma das melhores ferramentas para se analisar máquinas paralelas

4.1 O MPI

4.1.1 Histórico

A interface de troca de mensagens (MPI) é uma biblioteca de rotinas de envio e recebimento de mensagens. Quando o MPI é usado, os processos de um programa distribuído são escritos seqüencialmente, em C ou Fortran; eles se comunicam e sincronizam por meio de chamadas às funções MPI [1].

A API (Aplication Programmer's Interface) do MPI, foi definida em meados da década de 90 por um grande conjunto de representantes de faculdades, do governo e das indústrias [www.lam-mpi.org]. A interface reflete as experiências passadas dos representantes com as bibliotecas de trocas de mensagens

anteriores (como o PVM - Parallel Virtual Machine). O objetivo do grupo era desenvolver uma biblioteca única que poderia ser facilmente implementada em uma diversidade de máquinas multi-processadas. Segundo [1], o MPI vem se tornando cada vez mais um padrão, e diversas implementações estão disponíveis.

Os programas implementados em MPI seguem o estilo de programação distribuída SIMD – Single Instruction Multiple Data. Mais precisamente, cada processador executa uma cópia do mesmo programa – em uma aplicação escrita em C, cada processador executa a função *main*. Cada instância do programa pode realizar diferentes ações. As instâncias interagem entre si chamando as rotinas da biblioteca MPI. As funções MPI suportam comunicação processo a processo e comunicação por grupo, interagindo com o meio.

Abaixo, nós apresentaremos um programa simples, mas completo, tendo em vista que o mesmo ilustra a troca de mensagens processo a processo. Posteriormente serão descritas as funções de comunicação em grupo.

Figura 4.1: Programa MPI para a troca de valores entre dois processos.

```
#include <mpi.h>
main(int argc, char *argv[]) {
int my_rank, other_rank, size;
int length = 1, tag = 1;
int my_value, other_value;
MPI_Status status;

* inicializa o MPI e define o ranking do processo (my_rank) *
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_Comm_World, &size);
MPI_Comm_rank (MPI_Comm_World, &my_rank);

If (my_rank = = 0) {
        other_rank = 1;  my_value = 100;
}else {
        other_rank = 0; my_value = 200;
}
MPI_Send (&my_value, length, MPI_INT, other_rank,
                tag, MPI_COMM_WORLD);
MPI_Recv (&other_value, length, MPI_INT, MPI_ANY_SOURCE,
                tag, MPI_COMM_WORLD, &status);
printf ("processo %d recebeu %d  de  %d,  my_rank, ohter_value,
          other_rank");
MPI_Finalize ( ); }
```

As funções utilizadas no procedimento da Figura 4.1 são descritas no próximo tópico do presente capítulo

4.1.2 Funções Básicas

O MPI possui dezenas de funções [www.lam-mpi.org], sendo que dentre estas seis são consideradas de uso imprescindível na maioria dos programas.

Abaixo estaremos apresentando tais funções, e explicando a funcionalidade de cada uma delas.

MPI_Init.- Inicializa a biblioteca MPI e recebe uma cópia dos argumentos passados ao programa.
MPI_Comm_size – Determina o número de processos disparados.
MPI_Comm_rank – Determina o rank do processo atual.
MPI_Send – Envia uma mensagem a outro processo.
MPI_Recv – Recebe uma mensagem de outro processo.
MPI_Finalize – Termina a execução do programa MPI.

4.1.3 Pré Requisitos

Segundo [www.lam-mpi.org] o software necessário para a instalação do MPI é como se segue:

- Sistema Operacional Linux
- Compiladores GCC e F77 (disponíveis na maioria das distribuições Linux)

4.2 – A Implementação

Conforme visto anteriormente, é mais fácil implementar um algoritmo paralelo quando já se tem para este algoritmo um algoritmo PRAM de custo ótimo. O algoritmo PRAM para o Hiperquicksort foi previamente descrito no Capítulo 3. Demonstraremos agora como foi realizada a implementação das principais funções do algoritmo na linguagem MPI.

Conforme visto anteriormente, no Capitulo 3, o algoritmo PRAM do Hiperquicksort pode ser resumido em quatro fases:

1. Cada processador ordena sua lista local utilizando para isso o quicksort seqüencial;
2. Um processador dedicado calcula o pivô e realiza uma operação de broadcast, ou seja, distribui o pivô a todos os processadores do hipercubo;
3. Os processadores pertencentes ao subcubo inferior enviam seus elementos maiores que o pivô (High List) para seus respectivos parceiros no hipercubo superior. Processadores situados no hipercubo superior enviam seus valores menores ou iguais ao pivô (Low List) para seus respectivos parceiros no hipercubo inferior;
4. Cada processador realiza uma operação de *merge* entre a lista que recebeu e a lista que manteve, a fim de obter uma lista ordenada.

Na implementação MPI do algoritmo, cada processador realiza uma chamada à função Qsort() para ordenar sua lista local. A função Qsort() é uma chamada ao método de ordenação seqüencial quicksort. O algoritmo deve esperar até que todos os processadores tenham ordenado sua lista local para iniciar a próxima fase do algoritmo. Em MPI, a função MPI_Barrier( ) realiza exatamente esta tarefa. A execução do programa é interrompida até que todos os processadores atinjam este ponto. É importante salientar que a função MPI_Barrier gera um atraso na execução do algoritmo e apenas deve ser utilizada quando for inevitável. A Figura 4.2 ilustra a chamada à função MPI_Barrier ( ), assim que o processador ordena sua lista local.

```
/* 2) SORT USING SEQUENTIAL QUICKSORT  */
tam = comprimento;
qsort(dados, tam, sizeof(int), intorder);
//printf("\nProcesso [*%d*] ordenou: ", my_node);
for (i = 0; i < comprimento; i++)
 {
  //printf(" %d ", dados[i]);
 }
 //printf("\n");
 MPI_Barrier(MPI_COMM_WORLD);
```

Figura 4.2 – Chamada à função MPI_Barrier( )

Neste ponto cada processador possui sua lista ordenada e o algoritmo pode prosseguir sua execução.

Para a segunda fase do algoritmo, o broadcast do elemento pivô, utilizamos a função MPI_Bcast. A Figura 4.3 ilustra como a função MPI_Bcast foi utilizada no escopo do algoritmo.

Para a implementação da terceira e quarta fases do algoritmo, foram utilizadas as funções MPI_Send e MPI_Recv, da biblioteca MPI, já citadas no presente capítulo, e a função *merge*, presente na linguagem C. A Figura 4.4 demonstra a utilização das mesmas na implementação do algoritmo.

4.3 O Ambiente de Execução

O algoritmo Hiperquicksort analisado foi executado e testado sucessivamente, em hipercubos com dimensão variando entre zero (algoritmo seqüenci-

al não puro) e três. No total, foram utilizados oito computadores em rede, com a seguinte configuração:

- Máquinas com Processadores Intel Celeron 300MHz, 128Mb de memória RAM.

4.4 Os Testes

O algoritmo foi testado a fim de se comprovar o *speedup* atingido pelo mesmo. O *speedup* não foi medido relativamente ao tempo de execução do algoritmo Quicksort seqüencial. A opção escolhida foi calcular a razão entre o tempo de processamento paralelo e o tempo de processamento em um hipercubo de dimensão zero, o chamado Quicksort seqüencial não puro [6]. Esta escolha se deve ao fato de que o algoritmo seqüencial é demasiadamente eficiente e somado ao "overhead" gerado pelas sucessivas trocas de mensagens entre processadores, provavelmente não conseguiríamos atingir *speedup* algum.

Desta forma, se computarmos o tempo para hipercubos de dimensão 0 (zero) e compararmos com os tempos de hipercubos de 1, 2 e 3 dimensões, teremos uma forma de avaliar o *speedup*. Os tempos resultantes de cada processamento foram calculados por uma média de 10 processamentos, desprezando-se o maior e menor tempo de execução, além dos tempos demasiadamente fora da média. As Tabelas 4.1, 4.2, 4.3 e 4.4 ilustram a comparação entre os tempos de processamento para hipercubos de dimensão 0, 1, 2 e 3 processando vetores de 4.069, 8.192, 16.384 e 32.768 elementos.

Tabela 4.1 – Vetor de 4.096 elementos: D – dimensão do hipercubo, NP– Número de processadores do hipercubo, T – tempo em segundos, SO – Speedup Ótimo, SM – Speedup Medido.

D	NP	T	SO	SM
0	1	0,57	1	1
1	2	0,54	2	1,05
2	4	0,56	4	1,01
3	8	0,82	8	0.69

Tabela 4.2 – Vetor de 8.192 elementos: D – dimensão do hipercubo, NP– Número de processadores do hipercubo, T – tempo em segundos, SO – Speedup Ótimo, SM – Speedup Medido.

D	NP	T	SO	SM
0	1	0.58	1	1
1	2	0.52	2	1.11
2	4	0.58	4	1
3	8	0.84	8	0.69

Tabela 4.3 – Vetor de 16.384 elementos: D – dimensão do hipercubo, NP– Número de processadores do hipercubo, T – tempo em segundos, SO – Speedup Ótimo, SM – Speedup Medido.

D	NP	T	SO	SM
0	1	0.63	1	1
1	2	0.54	2	1.16
2	4	0.59	4	1.06
3	8	0.89	8	0.71

Tabela 4.4 – Vetor de 32.768 elementos: D – dimensão do hipercubo, NP– Número de processadores do hipercubo, T – tempo em segundos, SO – Speedup Ótimo, SM – Speedup Medido.

D	NP	T	SO	SM
0	1	0.70	1	1
1	2	0.56	2	1.25
2	4	0.67	4	1.14
3	8	0.82	8	0.85

4.5 Análise de Resultados

O *speedup* atingido pelo algoritmo Hiperquicksort m''ostrou-se, na média, muito abaixo do speedup ótimo. Além disso, o aumento do número de processadores ocasionou um aumento no tempo de processamento. Esta queda no *speedup* ao aumentarmos o número de processadores mostra que o algoritmo não é escalavel.

A não escalabilidade do algoritmo pode ser explicada por diversos fatores, dentre os quais citamos:

- O tempo excessivo necessário às trocas de mensagens entre os processadores;
- A rede não dedicada exclusivamente ao processamento paralelo. O tráfego de informações na rede pode ocasionar em congestionamento das mensagens trocadas entre processadores.

Notou-se também, um sensível aumento no *speedup* ao aumentarmos o número de elementos a serem ordenados. O melhores *speedups* medidos foram para vetores com 32.768 elementos. Esta constatação pode ser explicada pelo fato de que em grandes massas de dados, o ganho de desempenho compensa o *overhead* gerado pelas trocas de mensagens.

CAPÍTULO 5
CONCLUSÕES

A ordenação interna é um poderoso recurso computacional, que, em conjunto com a teoria da programação paralela e distribuída conseguiu alcançar um alto nível de funcionalidade. Atualmente, os métodos de ordenação paralelos, foco de estudo por diversos anos, estão sendo utilizados por milhares de máquinas em todos os tipos de aplicação.

Deve-se, porém, realizar um estudo avaliativo e verificar a real possibilidade de paralelização de um método qualquer. Conforme visto no presente trabalho, implementações paralelas nem sempre geram resultados satisfatórios e o custo de implementação pode pesar mais que os benefícios por ela gerados.

Concluímos também que ao se utilizar muitos processadores em máquina paralela em ambientes distribuídos, o *speedup* pode diminuir. No geral, a performance do algoritmo MPI mostrou-se razoável.

Como trabalhos futuros sugere-se:

- Realizar uma implementação do algoritmo utilizando funções mais avançadas da biblioteca MPI, tais como, MPI_Scatter e MPI_ Gather, responsáveis pela realizacao de uma distribuição e coleta de dados mais eficientes, respectivamente;
- Otimizar o tráfego de trocas de mensagens entre os processadores;
- Realizar uma implementação do algoritmo em um computador multi processado.

Referências Bibliográficas

[01] GREGORY R. ANDREWS, Foundations of Multithread, Parallel and Distrubuted Programming

[02] http://www.lam-mpi.org, [Consulta: 04/2002].

[03] MICHAEL J. QUINN, Parallel Computing Theory and Practice

[MONTE-MOR] MONTE-MOR, J.A. Paralelização de um Método de Aprendizado Indutivo de Máquina Baseado na Teoria de Conjuntos Aproximados.

[04] ALMASI,G.S; GOTTLIEB A Highly Parallel Computing, 2. ed. The Benjamins Cumming Publishing Company, 1994.

[05] FRANCIS B. MACHADO; LUIZ PAULO MAIA, Arquitetura de sistemas operacionais.

[06] ALBUQUERQUE, J.; COELHO JR., J.N. Avaliando o Hiperquicksort Utilizando uma NOW. *INFOCOMP*, vol.1, p.26-31

[07] CERIN, CHRISTOPHE; Na Out-of-Core Sorting Algorithm for Clusters with Processors at Diferent Speed.

[08] AMDAHL, G. Validity of the Single Processor Aproach to Achiev Large Scale Computing Capabilities. In: AAAFIPS Conference Proccedings, *Thompson Books,* v.30, p.483-485.

[09] DUNCAN, R. A Survey of Parallel Computers Architectures. *IEEE Computer*, p.5-16, Fevereiro 1990

[10] ZIVIANI, N. Projeto e Análise de Algoritmos com Implementações em C e PASCAL.