SERGIO LUIZ MARQUES FILHO

UMA ABORDAGEM MULTITHREAD EM APLICAÇÕES PARALELAS UTILIZANDO MPI

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto A. Hexsel

CURITIBA

2005

Agradecimentos

A Jesus por ter me dado o milagre da vida e por todas as bençãos e capacitação depositadas sobre minha vida.

A minha esposa Bárbara pelo apoio, incentivo e suporte em todos os momentos.

A meus pais Sérgio e Lourdes pelo incentivo.

Ao amigo e professor Roberto por ter ido além das atribuições de um Orientador, atitude que foi imprescindível para a realização desta dissertação.

A Vanderlei, Luci e Tiago pela ajuda oferecida sem restrições.

A Edson e Neusa Santini, a Jerry e ao Grupo Jeová Rapha por toda ajuda, apoio e suporte em orações.

A Hongzhang Shan e Kevin D. Clark por gentilmente terem cedido os programas originais constantes desta dissertação.

Aos Professores e Funcionários da Universidade Federal do Paraná, por todo auxílio e orientação.

Aos colegas de Mestrado, sempre prontos a ajudar nos momentos de dificuldades.

A Enabler do Brasil na pessoa de Fernando Estrazulas, Júlio Osachuk, Paulo Caovila e Geraldo Cruz por toda a ajuda, compreensão e disponibilidade de infra-estrutura.

Resumo

Esta dissertação avalia a utilização de *threads* em conjunto com o padrão *Message Passing Interface* (MPI) para esconder a latência das operações de comunicação em máquinas com memória física e logicamente distribuída. Foram utilizados na avaliação três núcleos de aplicativos (*kernels*), que são *Fast Fourier Transform* (FFT), fatorização *LU* e ordenação Radix. Foram desenvolvidas versões com *threads* a partir das versões que utilizam o padrão MPI.

Para comparar as duas abordagens, foram executados experimentos para medir o tempo de execução dos programas com 2, 4, 8 e 16 computadores. Os experimentos foram executados para conjuntos de dados fixos e conjuntos com tamanhos crescentes com o número de processadores. Os resultados indicam que a utilização conjunta de *threads* e MPI pode reduzir o tempo de execução dos programas paralelos. Os ganhos médios obtidos foram 22% com o *kernel* Radix, e 16% com o *kernel* FFT. Os resultados para o *kernel LU* indicam que esta abordagem nem sempre compensa o esforço adicional de programação.

Abstract

This dissertation assesses the use of lightweight threads in parallel processes that communicate through the *Message Passing Interface* (MPI) to conceal the communication latency in systems with distributed memory. Three kernels were used to measure the performance of both approaches, namely *Fast Fourier Transform* (FFT), *LU* factorization and Radix sorting. New thread-based versions were developed from the original MPI-based code.

Experiments were run to measure execution time in systems with 2, 4, 8 and 16 computers, using both fixed-size and scaled data sets. The results show that using threads in MPI-based programs can indeed reduce the execution time of parallel programs. The threaded version of the Radix kernel achieves an average gain of 22% over non-threaded, while the gain was 16% for the FFT kernel. The results for the *LU* kernel suggest that the use of threads does not always justify the extra programing effort.

SUMÁRIO

1 INTRODUÇÃO 1
1.1 Objetivos 2
1.2 Motivação 2
1.3 Organização do Trabalho 3
2 TRABALHOS RELACIONADOS 4
3 DEFINIÇÕES PRELIMINARES 8
3.1 Paralelismo 8
3.2 Máquinas Paralelas 8
3.3 Programação em Memória Compartilhada 9
3.4 Programação por Troca de Mensagens 9
3.4.1 Características do Paradigma de Troca de Mensagens 10
3.4.2 *Parallel Virtual Machine* 11
3.4.3 *Message Passing Interface* 11
3.4.4 Primitivas de Comunicação MPI 15
4 PROCESSOS LEVES 17
4.1 Introdução 17
4.2 POSIX *Threads* 20
4.2.1 Funções Básicas para o Gerenciamento de *Threads* 20
4.2.2 Sincronismo entre *Threads* 22
4.3 MPI e *Threads* 24
4.4 Comportamento das *Threads* 25
5 PROGRAMAS DE TESTES 26
5.1 *Fast Fourier Transform* – FFT 27
5.1.1 Implementação *Multithread* do *Kernel* FFT 30
5.2 Fatorização *LU* 35
5.2.1 Implementação *Multithread* do *Kernel* *LU* 39

5.3 Ordenação *Radix* 41
5.3.1 Implementação *Multithread* do *Kernel Radix* 44

6 METODOLOGIA DE MEDIÇÃO DE DESEMPENHO 47
6.1 Métricas de Desempenho 47
6.2 Metodologia dos Experimentos 48
6.2.1 Implementação *Multithread* 49

7 RESULTADOS 51
7.1 Ambiente de Testes 51
7.2 *Fast Fourier Transform* – FFT 51
7.3 Fatorização *LU* 55
7.4 Ordenação Radix 60

8 CONCLUSÃO 64
8.1 Trabalhos Futuros 65

LISTA DE TABELAS

3.1 Operações de comunicação ponto-a-ponto MPI 15
3.2 Principais operações de comunicação coletiva MPI 16
4.1 Exemplo de utilização de variáveis condicionais. 24
5.1 Taxa de Comunicação para Computação dos *kernels* utilizados. 27
7.1 Tempos de execução em milissegundos (ms) e Aceleração para o *kernel* FFT para um conjunto de dados de 1048576 números complexos. 52
7.2 Aceleração e Eficiência obtidas para o conjunto de dados de 1048576 números complexos, em relação a 2 processadores. 53
7.3 Aceleração e Eficiência obtidas para o conjunto de dados de 1048576 números complexos. 53
7.4 Tempos de execução em ms para o *kernel* FFT com variação no tamanho do conjunto de dados. 53
7.5 Tempos de execução em ms e Aceleração para o *kernel LU* para um conjunto de dados de 2048×2048 elementos. 56
7.6 Aceleração e Eficiência obtidas para o conjunto de dados de 2048×2048, em relação a 2 processadores. 56
7.7 Aceleração e Eficiência obtidas para o conjunto de dados de 2048×2048. . 56
7.8 Tempos de execução em ms para o *kernel LU* com variação no número de processadores e no tamanho do conjunto de dados. 57
7.9 Tempos de execução em ms para o *kernel LU* para 8 processadores com variação no tamanho dos blocos. 58
7.10 Tempos de execução em ms e Aceleração para o *kernel* Radix para um conjunto de dados de 16384 chaves. 61
7.11 Aceleração e Eficiência obtidas para o conjunto de dados de 16384 em relação a 2 processadores. 61
7.12 Aceleração e Eficiência obtidas para o conjunto de dados de 16384 chaves. . 61
7.13 Tempos de execução em ms para o *kernel* Radix com variação no número de processadores e no tamanho do conjunto de dados. 62

LISTA DE FIGURAS

3.1 Operações de comunicação coletiva. 15

4.1 Estados de uma *thread*. 18

4.2 Componentes do espaço de endereçamento de um processo. 19

4.3 Ilustração do processamento de uma função (a) e uma *thread* (b). 20

5.1 Distribuição do conjunto de dados para os processadores do *kernel* FFT. . 28

5.2 Fase de transposição da matriz. 28

5.3 Fase de transposição da matriz. 29

5.4 Linhas sendo transpostas individualmente. 31

5.5 Transposição de um bloco realizada no comando `MPI_Sendrecv`. 34

5.6 Distribuição do conjunto de dados para os processadores do *kernel* *LU*. . . 37

5.7 Distribuição dos blocos usando uma matriz de quatro dimensões. 38

5.8 Algoritmo do *kernel* Radix. 42

5.9 Fase de permutação das chaves. 44

6.1 Operações do método *monitor* adicionadas aos programas de teste. 49

7.1 Resultados obtidos no *kernel* FFT para um conjunto de dados de 1048576 números complexos. 52

7.2 Resultados obtidos no *kernel* FFT com variação no tamanho do conjunto de dados. 54

7.3 Resultados do *kernel* *LU* para um conjunto de dados de 2048×2048 elementos. 55

7.4 Resultados obtidos no *kernel* *LU* com variação no número de processadores e no tamanho do conjunto de dados. 57

7.5 Variação no tamanho do bloco no *kernel* *LU*. 59

7.6 Resultados obtidos no *kernel* Radix para um conjunto de dados de 16384 chaves. 60

7.7 Resultados obtidos no *kernel* Radix com variação no número de processadores e no tamanho do conjunto de dados. 62

CAPÍTULO 1

INTRODUÇÃO

A programação paralela tem sido amplamente utilizada na resolução de problemas que demandam grande poder computacional. Duas arquiteturas têm se mostrado dominantes na programação paralela, aquela que utiliza máquinas com memória central e esta memória é compartilhada entre vários processadores, e a arquitetura que utiliza memória física e logicamente distribuída pelos nós de processamento.

A arquitetura abordada nesta dissertação é a que utiliza memória distribuída entre os nós de processamento, e o paradigma de programação com troca de mensagens, pois este paradigma tem-se tornado cada vez mais comum. Ao agregar-se vários computadores com memória distribuída como um recurso computacional único, este é chamado de *cluster* e tem obtido destaque tanto no meio acadêmico quanto no meio empresarial, pois oferece alto desempenho e grande disponibilidade, com custos reduzidos.

À medida em que os microprocessadores tornam-se mais rápidos e aumenta a distância entre as velocidades da computação e da comunicação, a latência da rede de interconexão torna-se o fator dominante do tempo de execução de programas paralelos. A latência pode ser escondida caso seja sobreposta por processamento útil dentro do programa. As *threads* (processos leves) possuem a característica de migrarem o processamento para uma outra *thread* quando a *thread* que está sendo processada é bloqueada por algum motivo. Esta característica pode ser explorada caso a parte que realiza a comunicação de um programa seja executada em uma *thread*, e a parte que realiza a computação em outra *thread*. Assim, caso uma *thread* esteja bloqueada aguardando em uma primitiva de comunicação, será realizada uma troca de contexto entre as *threads* e o processamento pode continuar em outra *thread*, alcançando assim a sobreposição da comunicação com computação.

Resta-nos então integrar a programação *multithread* com a programação paralela por troca de mensagens, que nos programas empregados nesta dissertação é realizada através do pacote de *threads* POSIX, que dispõe de primitivas para a programação *multithread*, e do padrão para programação por troca de mensagens *Message Passing Interface* (MPI). MPI é um conjunto de padrões para a programação por troca de mensagens e foi desenvol-

vido por um fórum que envolveu representantes de diversas áreas, incluindo representantes das áreas acadêmica, empresarial e governamental.

Neste trabalho avalia-se a utilização de *threads* em conjunto com o padrão MPI na sobreposição da comunicação com a computação. Para isto utiliza-se três pequenos programas de testes (*kernels*) que são o *Fast Fourier Transform* (FFT), fatorização *LU* e ordenação Radix com o intuito de verificar se há ganho de desempenho ao utilizar-se *threads* em conjunto com o MPI.

Parte-se inicialmente de três versões paralelas dos *kernels* e desenvolve-se as versões *multithread* para os referidos *kernels*, e são tomadas medidas para analizar o desempenho destes programas.

1.1 Objetivos

O objetivo desta dissertação é avaliar o desempenho de versões *multithread* de programas paralelos utilizando uma biblioteca de troca de mensagens MPI.

Uma fração do tempo de execução de um programa paralelo é devida à latência na interconexão de rede. O tempo dispendido em comunicação pode ser sobreposto com computação para tentar diminuir o tempo total de execução de um programa paralelo, e o objetivo deste trabalho é avaliar a utilização de *threads* em conjunto com uma biblioteca MPI para investigar as condições para a sobreposição, e seus efeitos na diminuição do tempo de execução de uma aplicação.

Para isto partiu-se de versões paralelas de três programas de testes que foram modificadas de maneira a criar versões *multithread* destes programas, e as versões paralelas originais foram então comparadas com as versões *multithread*.

1.2 Motivação

A utilização de *clusters* de computadores vem se tornando cada vez mais comum, tanto na área acadêmica quanto na área empresarial. Programas paralelos tem sofrido com a latência na interconexão de rede que é responsável por parte do tempo de execução de um programa paralelo.

Investigar mecanismos para amenizar este tempo gasto com a comunicação entre os processadores tem sido o foco de muitos estudos no meio acadêmico [AJK+91, TCW92,

RADT97].

Estudos que investigam a utilização de *threads* em conjunto com o paradigma da troca de mensagens, em especial com o padrão MPI, não têm sido realizados, e pouco se sabe sobre o comportamento de programas paralelos que utilizam MPI em conjunto com *threads*.

Assim, este trabalho visa avaliar a utilização das *threads* juntamente com o padrão MPI na sobreposição da latência da rede de interconexão existente na troca de mensagens entre os processadores.

1.3 Organização do Trabalho

Este trabalho encontra-se organizado da seguinte maneira.

O capítulo 2 discute trabalhos correlatos ao aqui exposto. O capítulo 3 é destinado a definições preliminares, no qual são abordados assuntos como paralelismo, programação por troca de mensagens, e informações a respeito do padrão MPI. O capítulo 4 contém definições a respeito de *threads*, funcionamento dos atributos e da sincronização entre as *threads*.

No capítulo 5 vê-se o funcionamento básico dos três kernels constantes desta dissertação, *Fast Fourier Transform*, fatorização *LU* e ordenação Radix. E aborda a implementação *multithread* dos três *kernels*, bem como as modificações realizadas nestes programas para suportar a utilização de *threads*.

O capítulo 6 apresenta as medidas de desempenho utilizadas nesta dissertação e a metodologia empregada para a implementação *multithread* dos *kernels*.

O capítulo 7 discute os resultados obtidos com as versões paralela original e *multithread*. Finalmente o capítulo 8 discute os resultados verificados e trabalhos futuros que podem complementar e estender este estudo.

CAPÍTULO 2

TRABALHOS RELACIONADOS

O trabalho descrito em [Hél03] contém uma modelagem e predição de desempenho das primitivas da biblioteca MPI. No trabalho são estudados conceitos do modelo de programação paralela por troca de mensagens, bem como são discutidas técnicas para a modelagem e predição de desempenho, e detalhes da implementação das primitivas de comunicação do MPI. São modeladas as primitivas da biblioteca MPI e a predição de desempenho é validada pela medição dos tempos de comunicação destas primitivas. A medição do tempo de comunicação é tomada na execução de programas simples que apenas enviam e recebem mensagens. Para obter o tempo de comunicação das primitivas é tomada a média dos tempos de comunicação de um total de 30 a 50 experimentos realizados. Os experimentos com os maiores tempos de comunicação são descartados. No trabalho utiliza-se o método *monitor* para a tomada dos tempos de comunicação das primitivas MPI. As primitivas *scatter* e *gather* do MPI não foram avaliadas.

A modelagem e predição de desempenho são realizadas tanto para as primitivas de comunicação ponto-a-ponto como para as primitivas de comunicação coletiva e é mostrado que o tempo de comunicação para as primitivas de comunicação ponto-a-ponto tem crescimento linear em função do tamanho das mensagens, enquanto que o tempo de comunicação para as primitivas de comunicação coletiva tem crescimento linear em função do tamanho das mensagens e tem crescimento logarítmico em função do número de processadores.

A dissertação [Hél03] foi importante no desenvolvimento deste trabalho pelo fato de utilizar a mesma biblioteca para programação por troca de mensagens, e utilizou-se este estudo das primitivas do MPI na implementação dos *kernels* FFT, *LU* e Radix. Também a metodologia de avaliação do desempenho dos programas constantes do trabalho descrito em [Hél03] foi utilizada como base para a metodologia de avaliação do desempenho dos *kernels* FFT, *LU* e Radix.

O conjunto de aplicativos SPLASH-2 é descrito em [SME+95] com dois objetivos. O primeiro é a caracterização quantitativa dos programas visando verificar as propriedades

fundamentais e as interações com a arquitetura dos computadores. As propriedades dos programas estudadas neste trabalho são: (i) balanceamento de carga, (ii) razão entre a comunicação e a computação e (iii) o tráfego, bem como (iv) assuntos relacionados com a localidade espacial, e (v) como essas propriedades são escaláveis quando o tamanho do problema aumenta e quando o número de processadores é incrementado. O segundo objetivo é auxiliar pessoas que queiram utilizar os programas nele descritos em avaliações de arquiteturas.

Os *kernels* que foram utilizados nesta dissertação constavam de SPLASH-2 e os resultados encontrados em SPLASH-2 foram importantes para a definição do conjunto de dados definidos para os testes realizados com os *kernels* FFT, *LU* e Radix. Também no trabalho SPLASH-2 estavam descritos os padrões das taxas de comunicação para computação dos três *kernels* que estão descritos no capítulo 3.

Os artigos [Ste96, SJJ94] são trabalhos correlatos que respectivamente abordam as vantagens da transferência em blocos e as vantagens da troca de mensagens em um ambiente com coerência na memória *cache* em multiprocessadores com *caches* coerentes.

Os trabalhos [Ste96, SJJ94] utilizam diversos *kernels* para avaliar o desempenho da transferência em blocos no conjunto de dados dos programas.

Apesar de tratar-se de uma arquitetura diferente da arquitetura utilizada nesta dissertação, estes trabalhos anteriormente citados utilizam os mesmos *kernels* para a realização dos experimentos. Foi nestes trabalhos que encontrou-se informações de forma detalhada do funcionamento dos três *kernels*. Isto auxiliou no entendimento dos padrões de comunicação e computação dos algoritmos que foi importante para a implementação *multithread* desses *kernels*. Também nestes trabalhos encontrou-se informações dos locais do algoritmo onde podia-se explorar a sobreposição da comunicação com computação. No trabalho [SJJ94] estava descrito o conceito da transmissão de mensagens iniciada pelo receptor e da transmissão de mensagens iniciada pelo emissor, que foi utilizado nesta dissertação.

Os trabalhos [Ste96, SJJ94] indicam que quanto menor o tamanho das mensagens (blocos), a vantagem da utilização das transferencias em blocos também diminui, e que as transferências em blocos são mais eficientes para um número moderado de processadores e para tamanhos grandes do conjunto de dados. Os trabalhos também mostram que a utilização de memórias *cache* primárias de tamanho maior podem diminuir a vantagem

da transferência em blocos na arquitetura *Cache-Coherente*.

O trabalho [JD98] caracteriza os padrões de comunicação em aplicações científicas paralelas utilizando a programação por troca de mensagens, e avalia os componentes temporal, espacial, e o volume da comunicação. Os efeitos da variação do tamanho do problema também são avaliados. O trabalho [JD98] utiliza diversos *kernels* para a avaliação dos padrões de comunicação em aplicações científicas paralelas, e utiliza o mesmo método denominado *monitor* para a medição do tempo de processamento e do tempo de comunicação nos programas empregados. [JD98] apresenta a conclusão de que a freqüência da comunicação tende ao crescimento à medida em que o tamanho médio das mensagens decresce, e que quando o tamanho do problema cresce, a freqüência da comunicação pode crescer ou permanecer a mesma, mas o tamanho médio de cada mensagem cresce, uma vez que há mais dados a transmitir. Assim, problemas maiores produzem mensagens maiores ao invés de produzirem mais mensagens. Estes resultados foram importantes ao avaliar os resultados dos experimentos realizados nesta dissertação, uma vez que parâmetros como o tamanho do problema e o número e tamanho das mensagens são aspectos importantes em nosso trabalho.

Dois artigos analisam métodos para sobreposição da latência no acesso à memória. Em [AJK+91] são verificados quatro métodos para esconder a latência no acesso à memória, os quais são (i) memória com *cache* coerente, (ii) modelos relaxados de consistência de memória (*Relaxed memory consistency models*), (iii) busca antecipada controlada por software (*Prefetching*) e (iv) múltiplos contextos (*multiple-context* ou *threads*). Em [TCW92] são abordados os métodos de busca antecipada e em menor escala múltiplos contextos. Ambos utilizam *kernels* para a avaliação da sobreposição da latência no acesso à memória.

Estes trabalhos [AJK+91, TCW92] mostraram que memória com *cache* coerente e modelos relaxados de consistência de memória melhoram o desempenho de programas paralelos de maneira uniforme, e que o ganho de desempenho com busca antecipada e múltiplos contextos são consideráveis, mas são dependentes da aplicação. Também chegou-se à conclusão de que com combinações destas técnicas geralmente obtém-se um melhor desempenho do que com cada uma das técnicas individualmente, e que utilizando combinações destas técnicas pode-se chegar a ganhos de desempenho da ordem de 4 até 7 vezes.

Estes trabalhos foram importantes no desenvolvimento desta dissertação pois serviram

de base para a verificação de que a comunicação existente em um programa paralelo pode ser sobreposta com computação através da utilização de *threads*.

O trabalho [AJK^{+}91] utiliza o *kernel LU* e obteve o resultado de que quando o custo da troca de contexto é elevado, o desempenho do *kernel* é pior do que a versão sem troca de contexto utilizada para comparação. O mesmo comportamento foi verificado nesta dissertação.

Os programas de testes utilizados nesta dissertação foram avaliados anteriormente em [Dav90, Ble91, SJJ94]. Destes artigos foram retiradas informações importantes do funcionamento dos algoritmos dos programas de testes constantes desta dissertação.

CAPÍTULO 3

DEFINIÇÕES PRELIMINARES

Este capítulo apresenta os termos e conceitos básicos sobre paralelismo que são utilizados ao trabalhar-se com a programação por troca de mensagens.

3.1 Paralelismo

Um computador paralelo é composto por um conjunto de processadores capazes de interagir, cooperando para resolver um problema computacional. Para execução num computador paralelo uma tarefa é decomposta em partes menores de maneira a permitir a execução concorrente, buscando a diminuição no tempo total de execução da tarefa completa ou a execução com um conjunto de dados maior.

3.2 Máquinas Paralelas

A idéia inicial para a construção de máquinas capazes de executar programas paralelos foi conectar um grande conjunto de unidades de processamento a uma memória comum, dando origem aos Multiprocessadores, que são computadores que possuem mais de um processador e cooperam através do compartilhamento de dados em memória. Esta abordagem apresenta problemas de congestionamento no acesso à memória na medida em que o número de processadores cresce.

A memória das máquinas paralelas pode ser fisicamente compartilhada pelos processadores ou pode ser fisicamente distribuída entre vários nós de processamento interligados por uma rede de comunicação. Do ponto de vista lógico, a memória pode ser compartilhada e algum mecanismo garante que as escritas efetuadas por um processador sejam visíveis aos demais. Em máquinas com memória logicamente distribuída, o programador é responsável por disseminar aos demais processadores os valores recentemente computados pelos nós.

Para a memória fisicamente distribuída existem duas arquiteturas alternativas que diferem no método como os dados são compartilhados entre os processadores. Na pri-

meira, as memórias fisicamente distribuídas são endereçadas como um único espaço de endereçamento logicamente compartilhado e recebem o nome de *memória distribuída compartilhada* (DSM). Na segunda o espaço de endereçamento consiste de múltiplos espaços de endereçamento que são logicamente disjuntos e recebem o nome de *multicomputadores*.

Um multiprocessador simétrico (SMP) é um computador composto de múltiplos processadores, de modo que cada processador tem a mesma habilidade e aproximadamente o mesmo tempo de acesso a qualquer localização de memória [Gre95]. Um SMP é composto por um único sistema de entrada e saída e possui apenas uma memória global que é logicamente compartilhada entre os processadores.

Os *clusters* de computadores – também conhecidos como aglomerados ou agregados – são sistemas de computação paralela compostos por computadores independentes que cooperam trabalhando conjuntamente como se fossem um recurso único. Os aglomerados são portanto máquinas paralelas com memória fisica e logicamente distribuída.

3.3 Programação em Memória Compartilhada

A programação em memória compartilhada é um paradigma de programação empregado no desenvolvimento de programas paralelos em máquinas com memória fisicamente compartilhada, mas este paradigma também pode ser utilizado em máquinas com memória distribuída desde que exista um mecanismo para a criação de um espaço de endereçamento global que emula a memória compartilhada. Neste paradigma os processadores compartilham uma única memória (física ou emulada) com o intuito de cooperar na realização de uma tarefa.

3.4 Programação por Troca de Mensagens

A troca de mensagens é um paradigma de programação bastante popular, e é usado em máquinas com memória logicamente distribuída. Para promover a comunicação entre os processadores neste paradigma faz-se uso de primitivas do tipo `send()` e `receive()`, utilizadas para o envio e recepção de mensagens, respectivamente.

3.4.1 Características do Paradigma de Troca de Mensagens

A implementação do paradigma de troca de mensagens pode variar bastante, desde implementações em soquetes (*sockets*), passando por *middleware* até implementações de bibliotecas de funções com suas *application programming interfaces* (APIs). A comunicação por troca de mensagens pode ser utilizada em ambientes de processamento paralelo heterogêneos, em que pode ser necessária a conversão de formatos de dados para a compatibilização entre diferentes processadores.

As primitivas de comunicação podem ser implementadas com ou sem bloqueio. No primeiro caso – também conhecido como comunicação síncrona – o processo que está invocando uma das primitivas permanece bloqueado até que o outro processo invoque a primitiva simétrica. No segundo caso – comunicação assíncrona – os processos prosseguem após a invocação de uma das primitivas.

No paradigma de troca de mensagens existem três diferentes formas de identificação dos processos emissor e receptor, que são (i) a identificação por *canais* na qual canais unidirecionais ligam o processo emissor ao processo receptor, (ii) a identificação que utiliza um número fornecido pelo sistema operacional que é denominado *process identification* (*pid*) do processo receptor e (iii) a identificação por rótulos ou *tags* no qual os rótulos são utilizados para identificar os processos.

A comunicação que envolve apenas dois parceiros em uma comunicação por troca de mensagens é conhecida como *comunicação ponto-a-ponto*, mas existem algoritmos paralelos baseados em troca de mensagens nos quais faz-se necessário a *comunicação global* entre os processos. A comunicação global envolve mais do que dois parceiros em uma troca de mensagens. Os tipos de comunicação global existes são *um-para-todos*, *todos-para-um* e *todos-para-todos*, os quais são descritos a seguir.

Na comunicação *um-para-todos*, um único processo envia dados para todos os outros processos envolvidos no programa paralelo em execução.

Na comunicação *todos-para-um*, todos os processos do programa paralelo em execução enviam dados para um único processo e a comunicação pode ser acompanhada de uma operação de redução global dos dados.

Comunicação *todos-para-todos* é quando todos os processos do programa comunicam-se com todos os outros processos simultaneamente.

Comunicação *personalizada* pode enquadrar-se em um dos três tipos de comunicação

citados anteriormente, com a diferença de que esta comunicação envolve um subconjunto dos processos do programa paralelo.

As duas implementações de troca de mensagens mais conhecidos são o *Parallel Virtual Machine* (PVM) [GV93] e o *Message Passing Interface* (MPI) [JSD96]. PVM é um ambiente que oferece uma máquina virtual que congrega os recursos de processamento dos nós de um agregado. O MPI é uma especificação de uma interface padrão para implementações do paradigma da troca de mensagens.

3.4.2 *Parallel Virtual Machine*

A biblioteca para troca de mensagens PVM pode ser empregada em um conjunto de computadores heterogêneos ligados em rede de maneira a formar uma única máquina virtual.

Duas partes compõem o PVM, sendo estas um processo servidor (*daemon*) chamado de *Pvmd* que é executado em cada máquina da arquitetura. A outra parte do PVM é uma biblioteca de funções que implementa a comunicação entre os *Pvmds*.

PVM é orientado para operar em arquiteturas de computadores heterogêneas e torna transparente as conversões de formato de dados. Para a transmissão de mensagens em PVM pode-se optar pela utilização de protocolo de transporte TCP [Mat97] ou UDP [Mat97].

3.4.3 *Message Passing Interface*

O padrão MPI foi proposto inicialmente pelo *Message Passing Interface Forum* (MPIF), que é um fórum internacional e aberto composto por representantes de laboratórios das áreas industrial, acadêmica e governamental. MPI possui características de vários sistemas de troca de mensagens existentes, e a introdução deste padrão possibilitou aos desenvolvedores de programas paralelos a criarem bibliotecas eficientes e portáveis entre diferentes plataformas. Desta forma MPI não é uma nova linguagem de programação, mas sim um padrão para bibliotecas de definições e funções que podem ser utilizadas em conjunto com linguagens como C e Fortran para alcançar o paralelismo em programas computacionais. Utilizar a biblioteca MPI irá tornar transparente ao utilizador muitos dos detalhes de programação paralela e, como conseqüência, tornar a programação paralela

mais acessível. As características principais do MPI são discutidas a seguir.

Grupo de Processos Um grupo de processos é uma coleção ordenada de processos na qual cada um dos processos possui um número inteiro único utilizado para identificação, que é denominado *rank* e é iniciado em 0 e varia até $P-1$. Em MPI é permitido a um processo pertencer a mais de um grupo, o que implica a possibilidade da existência de diferentes *ranks* para um mesmo processo. O padrão MPI especifica primitivas para a criação de Grupos de Processos. Os Grupos de Processos são utilizados quando existe a necessidade da comunicação entre um subconjunto dos processos, assim pode-se definir um Grupo de Processos para este subconjunto de processos e uma operação de comunicação global pode ser realizada apenas no subconjunto de processos pertencentes ao Grupo de Processos.

Contexto de Comunicação O conceito de Contexto de Comunicação permite particionar o espaço de comunicação de maneira que uma operação de envio de uma mensagem possa ser direcionada a um grupo específico de processos, e mensagens enviadas em um contexto de comunicação não podem ser recebidas em outro contexto. Quando uma comunicação por troca de mensagens ocorre, deve-se informar a que contexto de comunicação a mensagem pertence, e isto dá-se através de um parâmetro nas primitivas de comunicação.

Comunicador Um comunicador (*communicator*) é um objeto MPI que provê o escopo apropriado para a execução de uma operação de comunicação MPI, e define um conjunto de processos que podem ser contactados. Assim, comunicador é um identificador de uma coleção de processos que podem trocar mensagens, e um processo também pode pertencer a mais de um comunicador.

Em um programa MPI, uma operação de comunicação é executada dentro de um contexto que é especificado por um comunicador que indica os possíveis processos receptores da mensagem.

Tipos de Dados Para a comunicação de dados em MPI faz-se necessário informar o número de elementos transmitidos. Desta forma não é necessário informar o número de bytes da mensagem, o que torna a tarefa de transmissão de mensagens independente da

característica da máquina que está sendo utilizada pois não é necessário o conhecimento do tamanho dos elementos envolvidos na comunicação.

Mensagens Em MPI uma *mensagem* é composta pelo *conjunto de dados* que quer-se transmitir, mais algumas informações de controle da mensagem que é denominado *envelope* [JSD96].

O *envelope* de uma mensagem é composto por um identificador do *destinatário* da mensagem, um identificador do *remetente* da mensagem, o *tamanho* ou um símbolo indicador do final do conjunto de dados, uma *etiqueta* (*tag*) que é um número inteiro que pode variar de 0 a 32767 que é especificado pelo programador para identificar a mensagem, e um comunicador.

Desta forma:

$$envelope = destinatario + remetente + tamanho + etiqueta + communicator \tag{3.1}$$

E,

$$mensagem = envelope + dados \tag{3.2}$$

Iniciação e Finalização Antes de qualquer outra função MPI poder ser utilizada, deve-se efetuar uma chamada à função `MPI_Init` que configura o sistema para permitir a utilização das funções constantes da biblioteca MPI. Após um programa ter finalizado a utilização da biblioteca MPI a função `MPI_Finalize` deverá ser invocada, e esta função finaliza qualquer trabalho pendente desta biblioteca, como por exemplo a liberação de memória alocada pelo MPI.

Tipos de Comunicação Em MPI estão disponíveis dois tipos de comunicação, que são a *comunicação ponto-a-ponto* e a *comunicação coletiva.*

A *comunicação ponto-a-ponto* é a que acontece entre um emissor e um receptor apenas. Esta comunicação pode ser do tipo bloqueante e não-bloqueante. Em MPI é disponibilizado um conjunto de primitivas para a realização deste tipo de comunicação que diferem pela utilização de diferentes formas de sincronismo entre os processos envolvidos e pelo uso ou não de *buffers* na transmissão da mensagem.

Nas operações do tipo *ponto-a-ponto* os processos emissor e receptor necessitam invo-

car primitivas distintas, como por exemplo o `MPI_send` – utilizado para transmissão da mensagem – e `MPI_Recv` – utilizado para a recepção da mensagem.

A *comunicação coletiva* é a que acontece entre um grupo de processos. Neste caso os processos envolvidos utilizam a mesma primitiva para realizar a comunicação, sendo a diferenciação entre qual é o processo emissor (ou emissores) e quais são os receptores (ou receptor) é dada por parâmetros segundo a semântica da operação.

O MPI suporta as primitivas de comunicação coletiva *Broadcast*, *Scatter*, *Gather*, *Reduce* e *All-to-all*. Estas primitivas são descritas a seguir e na Figura 3.1 é mostrado o funcionamento destas primitivas.

A operação *broadcast* é uma comunicação do tipo um-para-todos na qual uma mensagem é enviada do emissor a todos os processos do grupo, inclusive ele mesmo.

As operações *scatter* e *gather* são comunicações do tipo um-para-todos e todos-para-um, respectivamente. Neste tipo de operação as mensagens transmitidas são diferentes. Na operação *scatter* o processo emissor divide a quantidade de elementos a ser transmitida pelo número de processos pertencentes ao grupo e então uma mensagem diferente é enviada a cada um dos participantes do grupo, inclusive o próprio emissor. Na operação *gather* cada processo envia elementos ao processo receptor e estes são armazenados obedecendo a ordem dos *ranks*. Nas operações *scatter* e *gather* os tipos de dados devem ser compatíveis e o número de elementos enviados é sempre igual ao número de elementos recebidos.

Na operação *reduce* a comunicação é do tipo todos-para-um e o processo receptor recebe elementos enviados pelos demais integrantes do grupo e realiza uma operação de redução sobre estes dados. Todos os processos envolvidos devem especificar a mesma operação de redução e estas operações podem ser operações de soma, operações de mínimo e máximo e operações lógicas do tipo *AND* e *OR*.

A operação *all-to-all* é uma comunicação todos-para-todos, na qual cada processo envia partes distintas de um conjunto de dados para os demais integrantes do grupo. Neste tipo de operação a quantidade de elementos componentes da mensagem é dividida pelo número de participantes da comunicação e as partes da mensagem são destinadas aos demais receptores segundo o *rank* dos receptores.

Barreira MPI também oferece uma operação de sincronização explícita entre os processos de um grupo que se dá através de primitivas do tipo *barrier*. Nesta primitiva

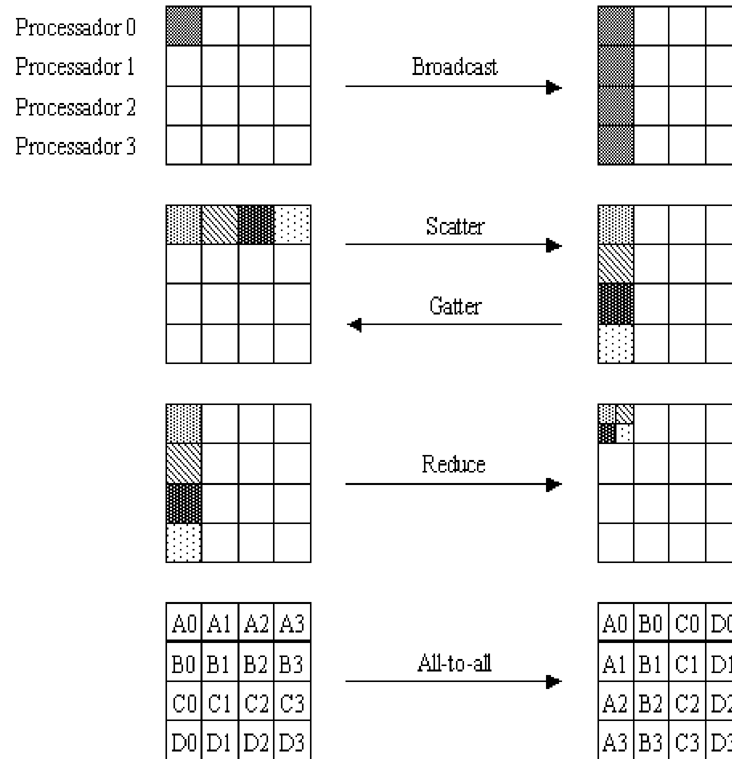


Figura 3.1: Operações de comunicação coletiva.

os processos ficam bloqueados até que todos os processos do grupo tenham alcançado a barreira.

3.4.4 Primitivas de Comunicação MPI

O padrão MPI disponibiliza um conjunto de primitivas para operações de comunicação que estão divididas em comunicação ponto-a-ponto e comunicação coletiva que são listadas nas tabelas 3.1 e 3.2.

Operação	Primitiva MPI
Send padrão	`MPI_Send(&buf, cnt, dtype, dest, tag, comm);`
Send buffered	`MPI_Bsend(&buf, cnt, dtype, dest, tag, comm);`
Send ready	`MPI_Rsend(&buf, cnt, dtype, dest, tag, comm);`
Send síncrono	`MPI_Ssend(&buf, cnt, dtype, dest, tag, comm);`
Send não bloqueante	`MPI_Isend(&buf, cnt, dtype, dest, tag, comm, &req);`
Receive bloqueante	`MPI_Recv(&buf, cnt, dtype, source, tag, comm, &status);`
Receive não bloqueante	`MPI_Irecv(&buf, cnt, dtype, source, tag, comm, &req);`

Tabela 3.1: Operações de comunicação ponto-a-ponto MPI

Operação	Primitiva MPI
Broadcast	`MPI_Bcast(&buf,cnt,dtype,root,comm);`
Scatter	`MPI_Scatter(&sndbuf,sndcnt,sndtype,&recvbuf,recvcnt,recvtyp,root,comm);`
Gather	`MPI_Gather(&sndbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtyp,root,comm);`
Reduce	`MPI_Reduce(&sndbuf,&recvbuf,cnt,dype,Op,root,comm);`
Barreira	`MPI_Barrier(comm);`
All-to-all	`MPI_Alltoall(&sndbuf,sndcnt,sndtype,&recvbuf,recvcnt,recvtype,comm);`

Tabela 3.2: Principais operações de comunicação coletiva MPI

CAPÍTULO 4

PROCESSOS LEVES

4.1 Introdução

Um **processo** [JM] é uma abstração de um programa em execução. O fluxo de instruções que está sendo executado por um processo também é conhecido como ***thread* de execução**. Este modelo pode ser estendido de maneira a permitir mais de uma *thread* em um único processo.

Threads [JM] permitem paralelismo de granularidade mais fina, pois enquanto a granularidade de um processo é um programa completo, a granularidade da *thread* pode ser de procedimentos dentro de um programa, que podem ser executados concorrentemente.

Os processos também são conhecidos como **processos pesados** no qual cada processo é composto por apenas uma *thread* de execução. Já as *threads* também são conhecidas por **processos leves**.

Os processos leves são implementados de maneira a que cada *thread* possua o seu próprio contador de programa (PC), sua própria pilha, mas os processos leves são executados sobre o mesmo espaço de endereçamento e podem compartilhar variáveis globais. Além disso, as *threads* também têm a capacidade de criar outras *threads*.

A comunicação entre as *threads* pode dar-se através de variáveis compartilhadas globais e a utilização destas variáveis pode ser controlada através de estruturas como monitores, semáforos, primitivas de exclusão mútua, variáveis condicionais e construções similares.

Existem dois modelos de controle de *threads*: (i) ***threads* a nível de usuário** e (ii) ***threads* a nível de sistema**, sendo que, no primeiro caso, as *threads* são executadas dentro de um processo e não são visíveis ao sistema operacional (SO), e no segundo caso o sistema operacional tem conhecimento de cada *thread* em execução e as *threads* competem entre si pelos recursos do sistema. Qualquer operação relativa a uma *thread* a nível de sistema é implementada como uma chamada de sistema (*system call*). Abordagens híbridas também são implementadas em alguns pacotes de *threads*.

O desenvolvimento de aplicações com múltiplas *threads* requer um ambiente com su-

porte a **compartilhamento de código** entre as *threads*, que também recebe o nome de *reentrância*.

Em muitos casos, aplicações *multithread* executam em computadores com menos processadores que os requeridos para direcionar cada *thread* a um processador dedicado. Por isto, em uma aplicação *multithread*, as *threads* devem compartilhar o tempo disponível do processador para a execução das *threads*. Um **escalonador** é parte do sistema operacional ou do pacote de *thread* e é responsável por alocar o tempo do processador para as *threads*.

Para a implementação da solução de compartilhamento do tempo do processador, as *threads* devem ser colocadas em processamento e ser posteriormente interrompidas para dar lugar à execução de outra *thread*. Para isto uma *thread* poderá assumir os estados: (i) **pronta** – indicando que está pronta para executar e é neste estado que as *threads* são criadas, (ii) **executando** – as *threads* assumem este estado toda vez que o escalonador aloca a *thread* a um processador, (iii) **bloqueada** – quando a *thread* deixa de ser processada por ter acabado o tempo destinado ao processamento da *thread* ou quando espera por alguma condição, e (iv) **finalizada** – quando a *thread* completa o seu processamento. Na Figura 4.1 são exemplificados os estados de uma *thread*.

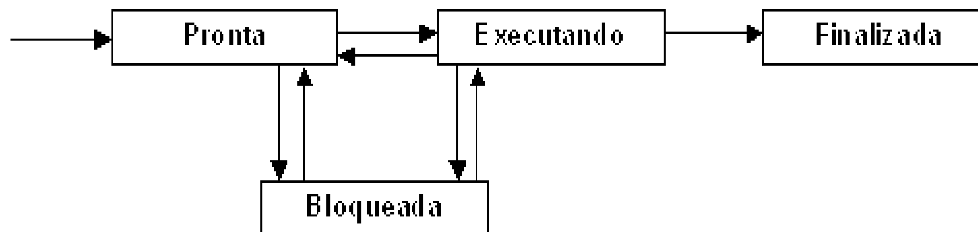


Figura 4.1: Estados de uma *thread*.

As *threads* existem dentro do espaço de endereçamento de um processo. Parte deste espaço de endereçamento é compartilhado por todas as *threads* pertencentes a um determinado processo, e a outra parte deste espaço de endereçamente é acessível unicamente pelas próprias *threads*.

Tomando-se como base um processo em ambiente Unix, os componentes do espaço de endereçamento que são compartilhados pelas *threads* são: (i) **código executável** que contém as instruções a serem executadas em linguagem de máquina, (ii) **dados estáticos** que são em geral constantes do programa, (iii) **dados dinâmicos** que é o espaço que não é alocado pelo programa quando o processo é iniciado. Os componentes do espaço de

endereçamento de um processo que não são compartilhados pelas *threads* são: (i) **pilha** (*stack*) que contém as variáveis de ambiente de execução, o *string* contendo a linha de comando e o endereço de retorno de procedimentos, e (ii) **registradores** que contém os registradores da Unidade Central de Processamento (CPU). Na Figura 4.2 estão mostrados os componentes do espaço de endereçamento de um processo típico do ambiente Unix.

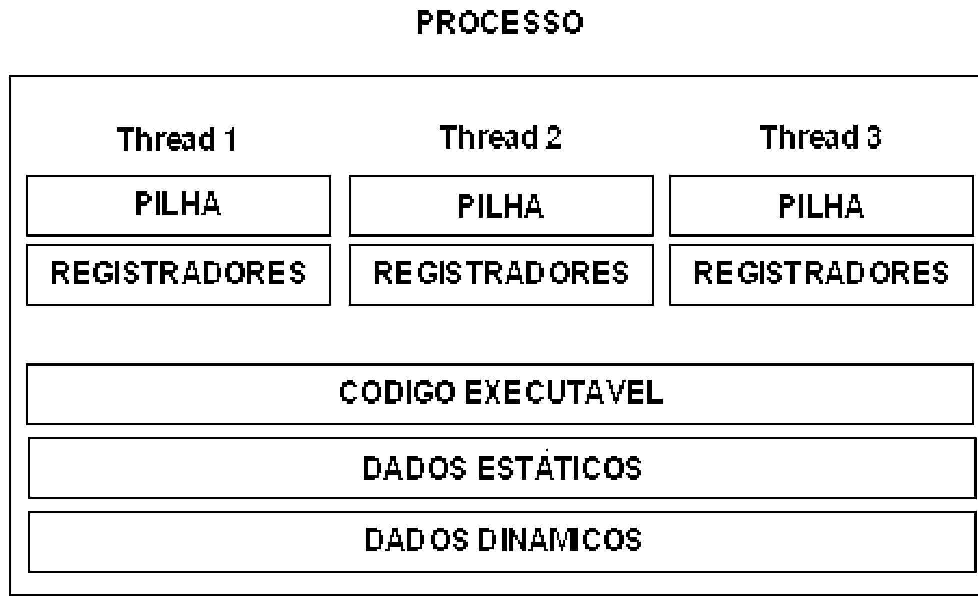


Figura 4.2: Componentes do espaço de endereçamento de um processo.

Na Figura 4.3 (a) é mostrado um esquema da execução de um programa que faz a chamada a uma função. Neste caso, a execução do programa que fez a chamada da função será desviada (`call`) para o código da função e retornará ao ponto de chamada no programa (`ret`) após a finalização do processamento da função. Caso a função fique bloqueada por uma eventual operação de Entrada/Saída todo o processo bloqueará. Já na Figura 4.3 (b) é mostrado o mesmo programa criando uma *thread* que executa a mesma função mostrada na Figura 4.3 (a). O programa criará um fluxo de instruções independente do programa principal e nunca retornará ao ponto de criação da *thread*. Caso a *thread* seja bloqueada por uma eventual operação de Entrada/Saída, somente a *thread* permanecerá bloqueada e o programa principal não estará bloqueado.

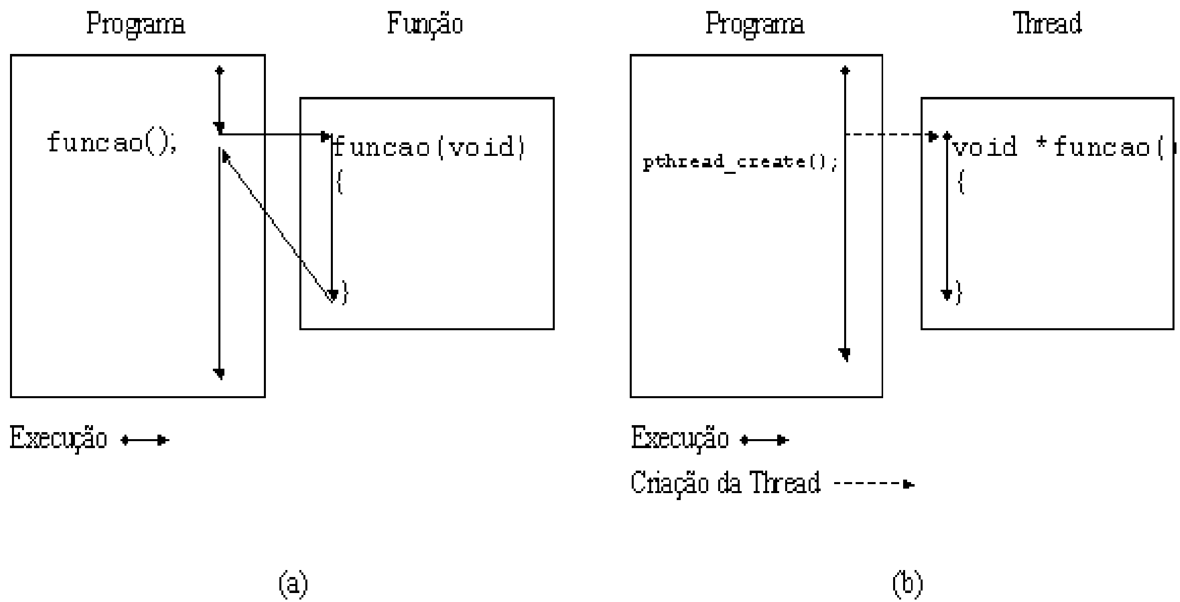


Figura 4.3: Ilustração do processamento de uma função (a) e uma *thread* (b).

4.2 POSIX *Threads*

POSIX significa *Portable Operating System Interface* (Interface de Sistema Operacional Portável) e é um padrão de programação desenvolvido pelo organização internacional Instituto dos Engenheiros Elétricos e Eletrônicos (IEEE). POSIX é um conjunto de padrões internacionais para interface com sistemas baseados em Unix.

Um pacote de *threads* tipicamente contém um sistema para o gerenciamento de *threads* de maneira transparente ao programador. Usualmente um pacote de *threads* inclui funções para a criação e destruição de *threads*, sincronismo entre *threads* através de exclusão mútua, variáveis condicionais e semáforos. O pacote de *threads* POSIX provê funções para a criação e destruição dinâmica de *threads*, desta maneira o número de *threads* não necessita ser informado antes do programa ser executado, podendo adaptar-se melhor às variações no tamanho do problema.

Nesta seção são descritos os comandos e características do pacote de *thread* POSIX que foi utilizado neste trabalho.

4.2.1 Funções Básicas para o Gerenciamento de *Threads*

Uma *thread* possui um identificador (*ID*), uma pilha, prioridade de execução e um endereço de início de execução. A estrutura interna de uma *thread* também contém informações

para o escalonamento da *thread*.

Em POSIX o comando `pthread_create` cria dinamicamente uma *thread* para executar uma função específica e a coloca em estado de **pronta**. `pthread_exit` causa a finalização da *thread* que chamou o comando, mas sem causar a finalização do processo por inteiro. `pthread_kill` envia um sinal a uma *thread* específica. `pthread_join` faz com que a *thread* espere pela finalização de uma outra *thread*. `pthread_self` retorna a identidade (*ID*) da *thread* que fez a chamada do comando.

O comando `pthread_yield` faz com que a *thread* que esteja executando ceda o processador a outra *thread* mesmo que o tempo destinado ao processamento da *thread* não tenha acabado.

Uma *thread* possui diversos atributos, os quais são tratados como **objetos**. Pelo fato de POSIX tratá-los como objetos, várias *threads* podem ser associadas ao mesmo objeto de atributos. Sendo assim, caso as propriedades de um objeto de atributos mudem, estas mudanças são refletidas em todas as *threads* associadas ao objeto. Também existem funções específicas para criar, configurar e destruir objetos de atributos, as quais são descritas abaixo.

O comando `pthread_attr_init` inicia um objeto de atributo de uma *thread* com os valores padrão. `pthread_attr_destroy` invalida os valores de um objeto de atributos. `pthread_attr_setstackaddr` e `pthread_attr_getstackaddr` respectivamente configura e examina o tamanho e a localização da pilha de uma *thread*. `pthread_attr_setdetachstat` e `pthread_attr_getdetachstat` respectivamente configura e examina a propriedade `detachstate`. Os valores possíveis para a propriedade `detachstate` são `PTHREAD_CREATE_JOINABLE` e `PTHREAD_CREATE_DETACHED`. `PTHREAD_CREATE_JOINABLE` indica que uma *thread* pode aguardar em um comando `pthread_join` para que a *thread* espere pela finalização de uma outra *thread*. `PTHREAD_CREATE_DETACHED` indica que uma *thread* deve chamar o comando `pthread_detach` quando da finalização da *thread* para liberar os recursos alocados. As *threads* são criadas como `PTHREAD_CREATE_JOINABLE` por padrão e a propriedade `detachstate` com o valor `PTHREAD_CREATE_JOINABLE` tem a liberação dos recursos alocados pelas *threads* realizada no comando `pthread_join`. `pthread_attr_setscope` e `pthread_attr_getscope` configura e examina a propriedade `contentionscope` que controla se a *thread* é criada a nível de usuário (`PTHREAD_SCOPE_PROCESS`) ou a nível de sistema (`PTHREAD_SCOPE_SYSTEM`).

pthread_attr_setinheritsched e pthread_attr_getinheritsched as funções configuram e examinam a propriedade inheritsched que controla se os parâmetros de escalonamento da *thread* serão herdados (PTHREAD_INHERIT_SCHED) ou não (PTHREAD_INHERIT_EXPLICITY) da *thread* criadora.

A informação sobre a política de escalonamento é armazenada em uma estrutura distinta do objeto de atributos. Os valores possíveis para a política de escalonamento são: (i) SCHED_FIFO que é a implementação do algoritmo *first-in-first-out* [JM] no qual a primeira *thread* na fila de *threads* prontas é a primeira a executar, (ii) SCHED_RR que é a implementação do algoritmo *round robin* [JM] no qual uma *thread* executa por um período de tempo, ao término deste período de tempo a *thread* é bloqueada e passa ao final da fila de *threads* prontas para executar, e (iii) SCHED_OTHER que é uma implementação dependente do pacote de *thread* e a implementação mais comumente encontrata é a do tipo *preemptiva*.

4.2.2 Sincronismo entre *Threads*

Quando as *threads* utilizam recursos compartilhados, deve haver sincronização nas iterações das *threads* para garantir que os resultados sejam consistentes. Existem dois tipos distintos de sincronismo: ***locking*** e ***waiting***, sendo que o primeiro refere-se tipicamente a eventos de bloqueio de curta duração e é destinado à exclusão mútua, já o segundo pode ter duração variável, referindo-se a bloqueios até que um determinado evento ocorra. POSIX chama de **mutex** e **variáveis condicionais** os dois tipos de mecanismos de sincronização.

Mutexes Um mutex ou *mutex lock* é o mais eficiente mecanismo de sincronismo entre *threads*, e os programas utilizam mutexes para preservar a semântica de seções críticas do programa e para obter acesso exclusivo a recursos. Em POSIX um mutex que for adquirido por uma determinada *thread* deve ser liberado pela mesma *thread* e uma vez adquirido o mutex, recomenda-se que este seja manipulado apenas por um período curto de tempo para evitar prejuízos no tempo de processamento das demais *threads*.

No fragmento de código abaixo é exemplificada a utilização de um mutex. Neste exemplo supõe-se que a variável a é compartilhada entre várias threads.

```
pthread_mutex_lock(&my_mutex);
```

```
a++;
pthread_mutex_unlock(&my_mutex);
```

Locking e *unlocking* são voluntários no sentido em que a exclusão mútua é alcançada apenas quando todas as *threads* corretamente adquiram o mutex antes de adentrar a uma região crítica.

Variáveis condicionais Uma variável condicional é um mecanismo que aguardará atomicamente por um evento em uma condição e é conveniente para bloquear a execução de uma *thread* até que uma combinação de eventos ocorra. As variáveis condicionais possuem operações atômicas para aguardar (`cond_wait`) e sinalizar (`cond_signal`).

As variáveis condicionais tem o seu nome derivado do fato de que são sempre utilizadas em conjunto com uma condição. Uma *thread* testará a condição e implicitamente executará `pthread_cond_wait` se o requisito for falso. Quando outra *thread* alterar as variáveis que devem fazer o requisito verdadeiro, esta sinalizará para as outras *threads* que estiverem bloqueadas na variável condicional, as quais retornarão ao processamento.

Para garantir que o teste da condição e o aguardo da *thread* sejam atômicos, a *thread* deve adquirir um mutex antes de testar a condição. A implementação POSIX garante que se uma *thread* bloquear-se em uma variável condicional o comando `pthread_cond_wait` bloqueia a *thread* e libera o mutex para que este possa ser adquirido por uma outra *thread* que atualizará a condição para liberar a *thread* bloqueada.

O comando `pthread_cond_signal` desbloqueia uma *thread* que esteja aguardando por uma condição. Se mais *threads* estiverem aguardando a mesma condição o comando `pthread_cond_broadcast` deverá ser chamado para desbloquear todas as *threads* que estiverem bloqueadas pela condição.

Na Tabela 4.1 estão mostrados dois fragmentos de código que indicam uma *thread* atualizando uma condição e sinalizando a outra *thread* que aguarda bloqueada pelo requisito.

thread 1	*thread 2*

`pthread_mutex_lock(&my_mutex);`
`procedure_done=1;`
`pthread_cond_signal(&items);`
`pthread_mutex_unlock(&my_mutex);`
	`pthread_mutex_lock(&my_mutex);`
	`while(!procedure_done)`
	` pthread_cond_wait(&items,&my_mutex);`
	`pthread_mutex_unlock(&my_mutex);`

Tabela 4.1: Exemplo de utilização de variáveis condicionais.

4.3 MPI e *Threads*

A biblioteca MPI pode trabalhar em conjunto com *threads* [JSD96]. O conceito *thread-safe* é quando múltiplas *threads* podem realizar chamadas a funções MPI ao mesmo tempo. O padrão para a implementação de distribuições MPI [For97] não requer um ambiente *thread-safe*, a garantia de um ambiente *thread-safe* é feita pelas distribuições MPI existentes. A distribuição MPI utilizada para a implementação dos *kernels* (MPICH) oferece um ambiente *thread-safe* para o desenvolvimento de aplicações MPI em conjunto com *threads*.

Quando trabalha-se com MPI e *threads*, deve-se substituir a função de inicialização do MPI que é denominada `MPI_Init` pela função `MPI_Init_threads` que tem como argumento de entrada o parâmetro `required`, que define qual será o suporte a *threads* requerido à biblioteca MPI, e o argumento de saída é o endereço da variável `provided` que será utilizada pela biblioteca MPI para dizer à aplicação qual suporte à *thread* foi concedido.

O suporte a *threads* disponível recai em uma das quatro categorias mostradas a seguir.

- `MPI_THREAD_SINGLE` que significa que nenhum suporte a *threads* é disponibilizado e a utilização desta categoria é equivalente à utilização do comando `MPI_Init`.
- `MPI_THREAD_FUNNELED` no qual apenas a *thread* principal pode fazer chamadas à rotinas MPI. Nesta categoria as outras *threads* podem executar outras tarefas enquanto a *thread* principal faz chamadas a funções MPI. Esta categoria pode ser utilizada quando a computação existente pode ser atribuída a várias *threads* que cooperarão na realização da tarefa, com a comunicação sendo realizada somente após as *threads* finalizarem o processamento.
- `MPI_THREAD_SERIALIZED` na qual múltiplas *threads* podem fazer chamadas a rotinas MPI mas apenas uma *thread* pode executar uma rotina MPI a cada momento.

- `MPI_THREAD_MULTIPLE` na qual as *threads* podem fazer chamadas a rotinas MPI sem qualquer restrição.

Em um programa MPI, todas as *threads* existentes compartilham o mesmo *comunicador* MPI. Assim uma mensagem destinada a uma *thread* estará acessível também às demais *threads* e é impossível endereçar uma *thread* especifica para receber uma determinada mensagem. Desta forma a *thread* que realizar uma chamada a uma função de recebimento da mensagem será a *thread* destinatária da referida mensagem. Também em um programa MPI quando uma *thread* é bloqueada por uma chamada a alguma função da biblioteca MPI, somente a *thread* que realizou a chamada permanecerá bloqueada até a finalização da requisição, as outras *threads* poderão continuar com o processamento.

4.4 Comportamento das *Threads*

Sabe-se que quando um processo realiza uma operação de Entrada/Saída (ES) bloqueante, o processo permanece bloqueado até que a entrada ou a saída torne-se disponível [KS96].

As *threads* podem ser utilizadas para sobrepor a Entrada/Saída com processamento, isto deve-se ao fato de que, como mostrado na figura 4.3, quando um processo faz a chamada a uma função, o *Contador do Programa* (PC) é desviado (`call`) para o início da função e só retornará ao ponto de chamada (`ret`) após todo o código da função ter sido executado. Quando uma *thread* é criada um novo fluxo de instruções é criado, que nunca retornará ao ponto de criação e o programa continua a ser executado, mas concorrentemente com a *thread*. Este comportamento das *threads* pode ser explorado de maneira a esconder a latência na comunicação existente em um programa paralelo.

A idéia básica de esconder a latência na comunicação com o uso de *threads* é desativar uma *thread* que esteja bloqueada aguardando a finalização de uma operação de comunicação e ativar uma outra *thread* que realize algum outro processamento. Quando a operação de comunicação tiver sido finalizada, o sistema pode continuar processando a *thread* original. Segundo [TCW92] existem algumas maneiras para esconder a latência, entre as quais estão (a) utilizando *threads* e (b) reorganizando o código em conjunto com um mecanismo de busca antecipada (*prefetch*). A reorganização de código é discutida em detalhes em [TCW92]. Nesta dissertação nos ateremos a utilização de *threads* para esconder a latência da comunicação entre os processadores.

CAPÍTULO 5

PROGRAMAS DE TESTES

Um programa de teste (*kernel*) é uma pequena parte de um programa real e escolhe-se a parte com a computação mais intensiva com o intuito de avaliar o desempenho de uma determinada máquina.

Uma aplicação completa é mais adequada à avaliação de desempenho do que um *kernel*, mas em inúmeros casos, o código da aplicação é complexo demais e não encontra-se disponível em versões paralelas.

Os *kernels* são adequados para a avaliação de desempenho, principalmente quando quer-se avaliar partes distintas da arquitetura [JD96]. O custo de implementar sistemas completos com o intúito de avaliar o desempenho é freqüentemente proibitivo, o que torna uma boa opção a escolha de *kernels* de aplicações, além do fato de os *kernels* constantes desta dissertação representarem uma grande gama de áreas de aplicações por incluírem os tipos de problemas que exigem computação intensiva mais comumente encontrados.

Para o desenvolvimento deste trabalho foram escolhidos três programas de testes: (i) *Fast Fourier Transform* (FFT), (ii) fatorização LU e (iii) ordenação Radix, os quais serão descritos adiante. A escolha dos referidos *kernels* deve-se ao fato de estes apresentarem taxas de comunicação para a computação bastante distintos, como é mostrado na tabela 5.1. Também os programas utilizados neste trabalho incluem uma fase de preparação de conjunto de dados bastante coerente com dados reais, e apresentam validações dos resultados após o processamento dos programas de maneira a garantir a coerência dos resultados.

Na tabela 5.1 são mostradas as taxas de comunicação para computação [SME+95] dos *kernels*, e nesta tabela P representa o número de processadores e N representa o tamanho do conjunto de dados. Pode-se observar que para o *kernel* Radix a taxa de comunicação para computação é constante durante toda a aplicação, enquanto que a taxa de comunicação para computação é diretamente proporcional a P e inversamente proporcional a N para os *kernels* LU e FFT, mas alterações no tamanho do conjunto de dados afetam menos o *kernel* FFT.

Kernel	Taxa Comunicação/Computação
FFT	$(P-1)/(PlogN)$
LU	$\sqrt{P}/\sqrt{N}$
Radix	$(P-1)/P$

Tabela 5.1: Taxa de Comunicação para Computação dos *kernels* utilizados.

5.1 *Fast Fourier Transform* – FFT

O *kernel* FFT é uma versão complexa e unidimensional do algoritmo descrito em [Dav90]. Este *kernel* FFT é ponto chave em muitas aplicações, tais como processamento de sinais, reconhecimento de voz, processamento de imagens, análise sísmica de petróleo e dinâmica dos fluídos.

O conjunto de dados para o algoritmo consiste (i) na *matriz de entrada* de N pontos complexos que serão transformados – na qual N é sempre uma potência de 2 – (ii) em outros N pontos complexos que são chamados de *Raízes da Unidade* que representam as N raízes complexas do número real 1, (iii) numa terceira *estrutura intermediária* de dados que também contém N pontos complexos e é utilizada como uma estrutura intermediária para manter valores calculados pelo algoritmo.

Tanto a matriz de entrada quanto a estrutura intermediária sofrem operações de leitura e gravação durante a execução do programa, enquanto que a matriz de raízes da unidade sofre apenas operações de leitura e esta estrutura é preenchida no início do programa. As estruturas são organizadas como matrizes de $\sqrt{N} \times \sqrt{N}$ elementos. O conjunto de dados é distribuído entre os processadores em blocos contíguos de linhas. A distribuição do conjunto de dados para os processadores é mostrada na Figura 5.1.

No primeiro passo do algoritmo a matriz de entrada é transposta na estrutura intermediária. No segundo passo do algoritmo são realizadas FFTs unidimensionais em cada linha da matriz intermediária com os resultados sendo armazenados novamente na matriz intermediária. No terceiro passo do algoritmo os elementos correspondentes da matriz raízes da unidade são aplicados a cada elemento da matriz intermediária. No quarto passo a matriz intermediária é transposta na matriz de entrada. No quinto passo são realizadas FFTs unidimensionais em cada linha da matriz de entrada com os resultados sendo armazenados na própria matriz de entrada. No último passo do algoritmo a matriz de entrada é transposta na matriz intermediária.

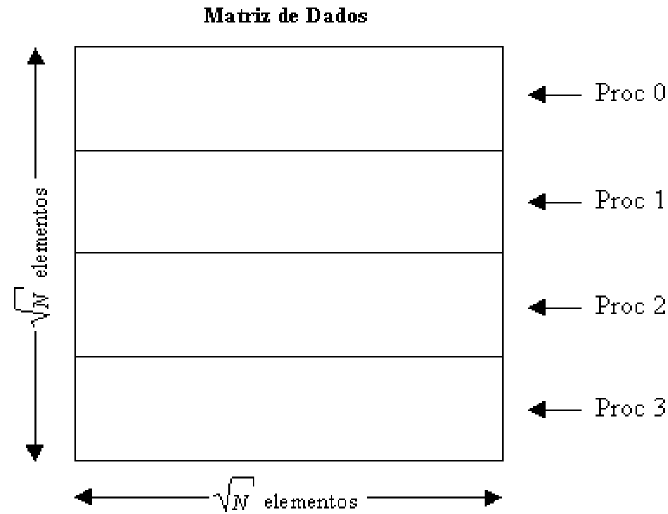


Figura 5.1: Distribuição do conjunto de dados para os processadores do *kernel* FFT.

A comunicação entre os processadores ocorre apenas durante as fases em que há a transposição da matriz. Existe um bloco de dados que sempre é transposto localmente – o bloco de dados pertencente ao próprio processador – e os outros $P-1$ blocos são sempre comunicados para os outros processadores. Durante a fase de transposição da matriz todos os processadores comunicam-se com todos os outros processadores. Um esquema da fase de transposição da matriz é mostrado na Figura 5.2.

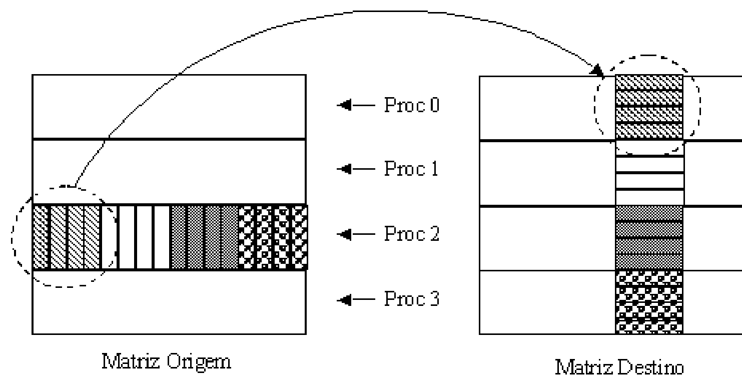
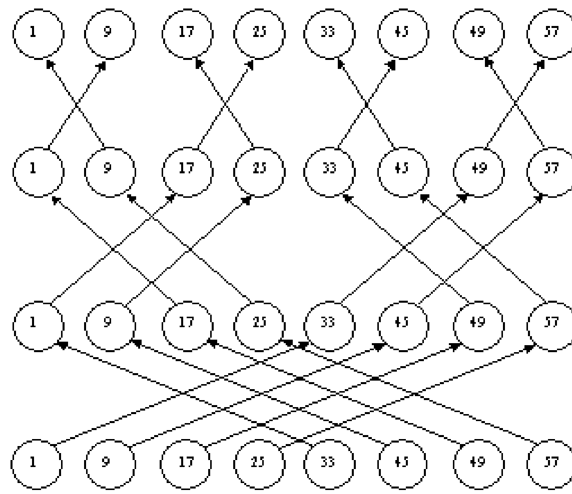


Figura 5.2: Fase de transposição da matriz.

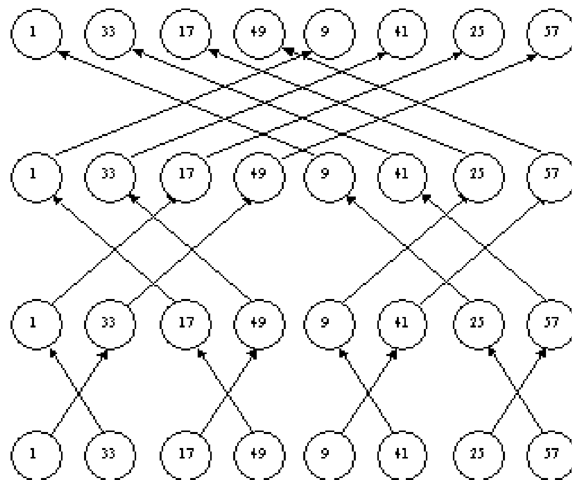
Durante o terceiro passo do algoritmo, os elementos correspondentes da matriz raízes da unidade são multiplicados pelos seus pares na matriz de entrada, e durante o segundo e o quinto passo do algoritmo, FFTs unidimensionais também fazem referência à matriz raízes da unidade.

Com o intúito de melhorar o desempenho do algoritmo, os passos 2 e 3 do algoritmo

foram agrupados, uma vez que ambos fazem referência à matriz raízes da unidade aproveitando os dados que estão na memória *cache* do processador. Esta alteração foi inicialmente descrita em [Dav90]. Outra alteração foi realizada no algoritmo para melhorar o desempenho: durante as fases 3 e 5 do algoritimo os elementos da matriz são multiplicados de forma que graficamente os elementos ficariam representada pela estrutura conhecida como *butterfly* – mostrada na Figura 5.3a. A ordem dos elementos nesta estrutura foi trocada (Figura 5.3b) para aproveitar a localidade espacial dos dados na memória do computador. Esta alteração foi descrita em [SJJ94]. Na Figura 5.3 está representada a aplicação do algoritmo *butterfly* em uma linha de uma matriz transposta de 8×8 elementos.



(a)



(b)

Figura 5.3: Fase de transposição da matriz.

5.1.1 Implementação *Multithread* do *Kernel* FFT

Como visto anteriormente, o *kernel* FFT utiliza o algoritmo denominado *Six-Step* FFT, o qual emprega seis passos para completar o algoritmo. O algoritmo requer três transposições da matriz do conjunto de dados, as quais representam os passos 1,4 e 6 do algoritmo. Esta transposição é realizada no algoritmo através da função denomina `Transpose`. O algoritmo requer duas aplicações de FFTs unidimensionais no conjunto de dados que representam as fases 2 e 5 do algoritmo, e esta função foi denominada `FFT1DOnce`. É realizada uma aplicação da matriz Raízes da Unidade na matriz de dados, esta é a fase 3 do algoritmo, e esta função foi denominada `TwiddleOneCol`. As funções que realizam os seis passos requeridos pelo algoritmo são mostradas abaixo na seqüência que estão no programa.

```
1 Transpose
2 FFT1DOnce
3 TwiddleOneCol
4 Transpose
5 FFT1DOnce
6 Transpose
```

As fases 2 e 3 do algoritmo foram agrupadas para aproveitar a localidade temporal do conjunto de dados, uma vez que ambas as fases fazem uso da estrutura Raízes da Unidade. Esta melhoria no programa tem o intuito de beneficiar o desempenho do programa e já constava da versão paralela original utilizada neste trabalho.

Na versão original, a fase de transposição da matriz era primeiramente completada para só então dar prosseguimento as outras fases do algoritmo. Sabe-se que uma maneira de melhorar o desempenho de um programa paralelo é sobrepor o tempo gasto no envio de uma mensagem (comunicação), por uma tarefa realizada no processador (computação) [JD96]. A versão original do programa não explorava este fato.

Como exposto anteriormente os passos 2 e 3 do algoritmo foram combinados em um único passo de computação, e o passo 5 também é um passo computacional. Os passos 1,4 e 6 do algoritmo são passos que exigem comunicação. O desempenho do programa pode ser melhorado caso o programa seja reestruturado de maneira a sobrepor comunicação com computação nos passos do algoritmo. O processador realiza FFTs unidimensionais

e aplica a matriz Raízes da Unidade a todas as linhas da matriz que estão de posse de um processador. Para sobrepor a comunicação com computação, as linhas que estão de posse de um processador podem ser transpostas individualmente. Após a transposição, realiza-se FFTs unidimensionais e aplica-se a matriz Raízes da Unidade na linha que fora transposta, e pode-se transpor outra linha da matriz enquanto realiza-se a tarefa de FFTs unidimensionais e aplicação da matriz Raízes da Unidade. Na figura 5.4 é demonstrada a transposição individual das linhas da matriz. A computação pode ser realizada concorrentemente às transposições das linhas da matriz.

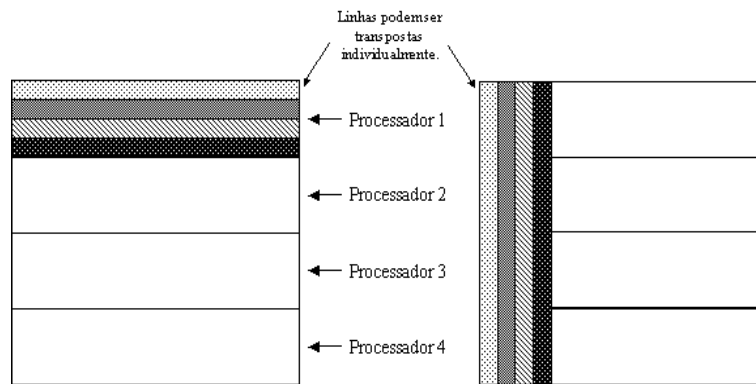


Figura 5.4: Linhas sendo transpostas individualmente.

Após a alteração realizada no programa, os seis passos do algoritmo ficaram como mostrado no fragmento de código abaixo.

```
for (j=0; j<rowsperproc; j++) {
   Transpose(j, n1, x, scratch);
   FFT1DOnce(direction, m1, n1, umain, &scratch[2*j*n1]);
   TwiddleOneCol(direction, n1, N, j, umain2, &scratch[2*j*n1]);
}

for (j=0; j<rowsperproc; j++) {
   Transpose(j, n1, scratch, x);
   FFT1DOnce(direction, m1, n1, umain, &x[2*j*n1]);
}

for (j=0; j<rowsperproc; j++) {
```

```
 Transpose(j, n1, x, scratch);
}
```

Após a alteração realizada tanto a transposição da matriz quanto a computação nas FFTs unidimencionais e aplicação das Raízes da Unidade são realizadas linha a linha. Utilizou-se comunicação não bloqueante para a implementação desta funcionalidade e existem pontos de sincronização no interior das funções de forma a garantir a coerência dos dados. Nenhuma sobreposição de comunicação com computação pode ser inserida na última transposição do algoritmo.

Existe uma barreira no início do algoritmo para garantir que todos os processadores estejam sincronizados e outra barreira antes de realizar-se o teste final para conferência dos resultados. O teste consiste em aplicar o próprio algoritmo mas no sentido inverso, e este sentido é representado pela variável denominada `direction` que pode conter os valores 1 ou -1 e é utilizado nas funções `FFT1DOnce` e `TwiddleOneCol`.

Durante a transposição da matriz, blocos de elementos de tamanho $\frac{\sqrt{N}}{P}$ são enviados a $P-1$ processadores. Mas sempre existirá um bloco de tamanho $\frac{\sqrt{N}}{P}$ que será transposto localmente.

Uma alteração foi realizada na função `Transpose` a qual passa-se a descrever.

Em MPI existem duas funções para o envio e recebimento simultâneos de mensagens. São as funções `MPI_Sendrecv_replace` e `MPI_Sendrecv`. Embora estas duas funções realizem a mesma tarefa, existe uma diferença entre elas: `MPI_Sendrecv` utiliza *buffers* diferentes para enviar e receber a mensagem. Já `MPI_Sendrecv_replace` utiliza o mesmo *buffer* para enviar e receber a mensagem [Pet97].

O programa original utiliza a função `MPI_Sendrecv_replace` da biblioteca MPI para enviar a linha a ser transposta para o destinatário. Ao receber a linha, um passo adicional era necessário no processador receptor, pois a linha previamente enviada ainda necessitava ser transposta, e isto era feito localmente no processador. Assim, o processador receptor recebia a mensagem com a linha e percorria-a por completo transpondo os elementos individualmente na matriz através de comandos do tipo `load` e `store` na memória local do processador receptor.

Foi realizada uma alteração no programa de maneira que a função `MPI_Sendrecv_replace` foi substituída pela função `MPI_Sendrecv` e o passo adicional descrito no parágrafo anterior foi suprimido, pois a transposição da matriz é realizada diretamente através

de uma característica explorada do comando `MPI_Sendrecv` que passa-se a mostrar.

A biblioteca MPI permite a criação de novos tipos de dados, além dos tipos de dados básicos, como: `int`, `double`, `char`, entre outros. A função `MPI_Type_contiguous` é utilizada para definir um tipo de dado composto por vários elementos consecutivos na memória do computador. A função `MPI_Type_vector` define um tipo de dado no qual os elementos não são consecutivos mas estão igualmente espaçados na memória do computador [Pet97].

Na alteração realizada, para o envio da linha da matriz a ser transposta, foi definido um novo tipo de dado do MPI denominado `rowtype` que mapeia $\frac{\sqrt{N}}{P}$ elementos de uma **linha** na matriz armazenada na memória local. O comando responsável pela criação do tipo `rowtype` é mostrado abaixo. O tipo de dado é criado com $2\frac{\sqrt{N}}{P}$ pelo fato de tratar-se de uma matriz de números complexos.

```
MPI_Type_contiguous(2*rootN/P,   /* contador */
                    MPI_DOUBLE,  /* tipo basico */
                    &rowtype     /* endereço do novo tipo */
                    );
MPI_Type_commit(&rowtype);
```

Também foi criado um novo tipo de dado MPI denominado `coltype` que mapeia uma **coluna** na matriz armazenada na memória local. O comando responsável pela criação do tipo `coltype` é mostrado abaixo. O novo tipo de dado é criado com um total de $\frac{\sqrt{N}}{P}$ números compostos por 2 elementos cada e com deslocamento de $2\sqrt{N}$ na memória do computador. O comando responsável pela criação do novo tipo de dado é mostrado abaixo.

```
MPI_Type_vector(rootN/P,   /* contador */
                2,         /* tamanho do bloco */
                2*rootN,   /* deslocamento */
                MPI_DOUBLE,
                &coltype);
MPI_Type_commit(&coltype);
```

Desta maneira um bloco de tamanho $2\frac{\sqrt{N}}{P}$ da linha da matriz a ser transposta que é destinado a um processador é **enviado** com o tipo de dado **linha** (`rowtype`) mas é

recebido como o tipo de dado **coluna** (`coltype`). Desta maneira a transposição da matriz é feita diretamente no comando `MPI_Sendrecv` sem a necessidade de processamento local no processador que recebeu a mensagem. Assim eliminou-se o passo adicional constante do programa original. O funcionamento descrito acima é mostrado na figura 5.5.

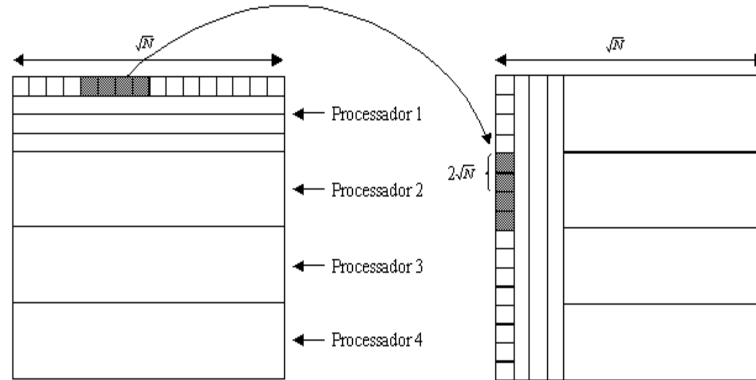


Figura 5.5: Transposição de um bloco realizada no comando `MPI_Sendrecv`.

Para toda linha transposta existe uma parte da linha de $\frac{\sqrt{N}}{P}$ elementos que será transposta localmente. Esta é a parte da linha que é destinada ao próprio processador. Esta parte da matriz que é transposta localmente continuou a existir no programa.

A comparação do desempenho das versões *multithread* e paralela foi realizada em programas onde ambos tinham a modificação descrita acima e os resultados serão mostrados em um capítulo a parte.

Para a implementação *multithread* do *kernel* FFT o processamento foi dividido em duas funções que realizam tarefas distintas. A primeira função é denominada `FFT1D_thread`, é a *thread* de **computação** e tem por tarefa realizar as atividades que empregam processamento – cálculos – no algoritmo, e a segunda função, que é denominada `TRANSPOSE_thread`, é a *thread* de **comunicação** que realiza atividades que envolvem a troca de mensagens entre os processadores.

Desta forma, a função `FFT1D_thread` foi designada para realizar as fases 2,3 e 5 do algoritmo, e a função `TRANSPOSE_thread` foi designada a realizar as fases 1, 4 e 6 do algoritmo.

Como as *threads* são fluxos de instruções que nunca retornarão ao ponto de criação da *thread* pelo programa principal [AJK$^+$91], todas as *threads* foram criadas com o atributo `detachstate` designado a `PTHREAD_CREATE_JOINABLE`, para que, após as chamadas de criação das *threads*, fosse adicionado um comando `pthread_join` de maneira a que o

programa principal não finalizasse até que as duas `threads` tivessem terminado o processamento.

Após finalizar a transposição de uma linha, a `thread` de comunicação irá sinalizar o término da transposição da linha, e também irá mudar o estado de uma variável de controle (*flag*) indicando que a linha corrente já está apta a ser calculada pela função `FFT1DOnce`. A variável de controle foi necessária porque uma *thread* pode ainda não ter atingido o ponto de sincronização e a outra *thread* já ter sinalizado. Então a *thread* de computação irá verificar se a variável de controle está assinalada, caso contrário ira chamar `pthread_cond_wait` de maneira a aguardar pelo sinal que indica a finalização da transposição.

Não houve a necessidade de sincronização entre o início das duas *threads* – barreiras nas *threads* – pois a sincronização descrita acima já garante que a *thread* de computação só irá prosseguir caso a *thread* de comunicação tenha realizado a transposição de ao menos uma linha.

Uma melhoria que pode ser realizada é que, como a transposição de uma linha da matriz é feita em blocos de $\frac{\sqrt{N}}{P}$ elementos, seja permitido que a função `FFT1DOnce` prossiga nos blocos que já tiverem sido transpostos.

5.2 Fatorização LU

LU é um *kernel* de uma fatorização LU de uma matriz densa, e representa muitos dos problemas de álgebra linear como fatorização QR e fatorização Cholesky.

Este *kernel* fatora uma matriz densa A no produto de uma matriz triangular inferior L e uma matriz triangular superior U. Para uma matriz de tamanho $n \times n$ o tempo de execução é $O(n^3)$ e o paralelismo é proporcional a n^2 [JD96]. O algoritmo utilizado é o descrito em [SJJ94]. A matriz A de tamanho $N \times N$ a ser fatorada, é decomposta em blocos de tamanho $B \times B$, com computação e comunicação sendo realizadas nestes blocos.

Durante cada iteração do algoritmo, os blocos que serão computados são classificados de três formas. Na iteração n existe apenas um *bloco diagonal* que está localizado na diagonal principal (linha n e coluna n). Todos o blocos que estão localizados na mesma linha, mas à direita do bloco diagonal e na mesma coluna, mas abaixo do bloco diagonal são chamados de *blocos de perímetro* e todos os blocos localizados na região delimitada

pelos blocos de perímetro são chamados *blocos interiores.*

Cada iteração do algoritmo consiste de três passos. No primeiro passo da iteração n o processador que possui o bloco diagonal na linha n e coluna n faz a fatoração do bloco diagonal. No próximo passo do algoritmo, todos os processadores que possuem blocos de perímetro fazem uso do bloco diagonal que fora previamente atualizado de maneira a atualizarem os blocos de perímetro. No terceiro passo do algoritmo todos os processadores que possuem blocos interiores fazem uso dos valores atualizados dos blocos de perímetro na mesma linha e coluna da matriz para atualizarem os blocos interiores. O terceiro passo do algoritmo é o que consome mais tempo pois é o que envolve o maior número de elementos a serem atualizados.

Neste algoritmo, a comunicação ocorre quando os blocos são transmitidos entre os processadores, de maneira que no primeiro passo do algoritmo não existe comunicação, pois o processador que possui o bloco diagonal apenas fatora e atualiza o bloco diagonal. No segundo passo do algoritmo existe a comunicação quando os processadores que possuem os blocos de perímetro fazem uso do valor atualizado do bloco diagnal com o intúito de atualizarem os blocos de perímetro. No terceiro passo do algoritmo também ocorre a comunicação quando os processadores que possuem blocos interiores fazem uso dos valores atualizados dos blocos de perímetro para atualizarem os blocos interiores. A distribuição do conjunto de dados para os processadores é mostrada na Figura 5.6.

Na programação paralela, com relação ao início da comunicação, existem duas maneiras de classificar a comunicação entre os processadores. Na primeira, que é conhecida como comunicação *iniciada pelo receptor*, geralmente utiliza operações de `load` para obter os dados necessários ao processador, e a segunda classificação é conhecida como comunicação *iniciada pelo transmissor*, na qual o processador que detém a informação envia-a ao receptor. O *kernel LU* utiliza a comunicação *iniciada pelo transmissor*, uma vez que após um processador fatorar um determinado bloco, o processador então procede a transmissão do bloco aos processadores que estarão aguardando por esta informação, e somente após a finalização da transmissão estes processadores iniciarão a atualização dos outros blocos.

Desta maneira um processador que detém um bloco diagonal irá proceder a fatoração do bloco diagonal no primeiro passo de cada iteração do algoritmo, e iniciará a transmissão para cada processador que seja proprietário de blocos de perímetro. Após, cada processador que detém blocos de perímetro aguardará pela finalização da transmissão do bloco

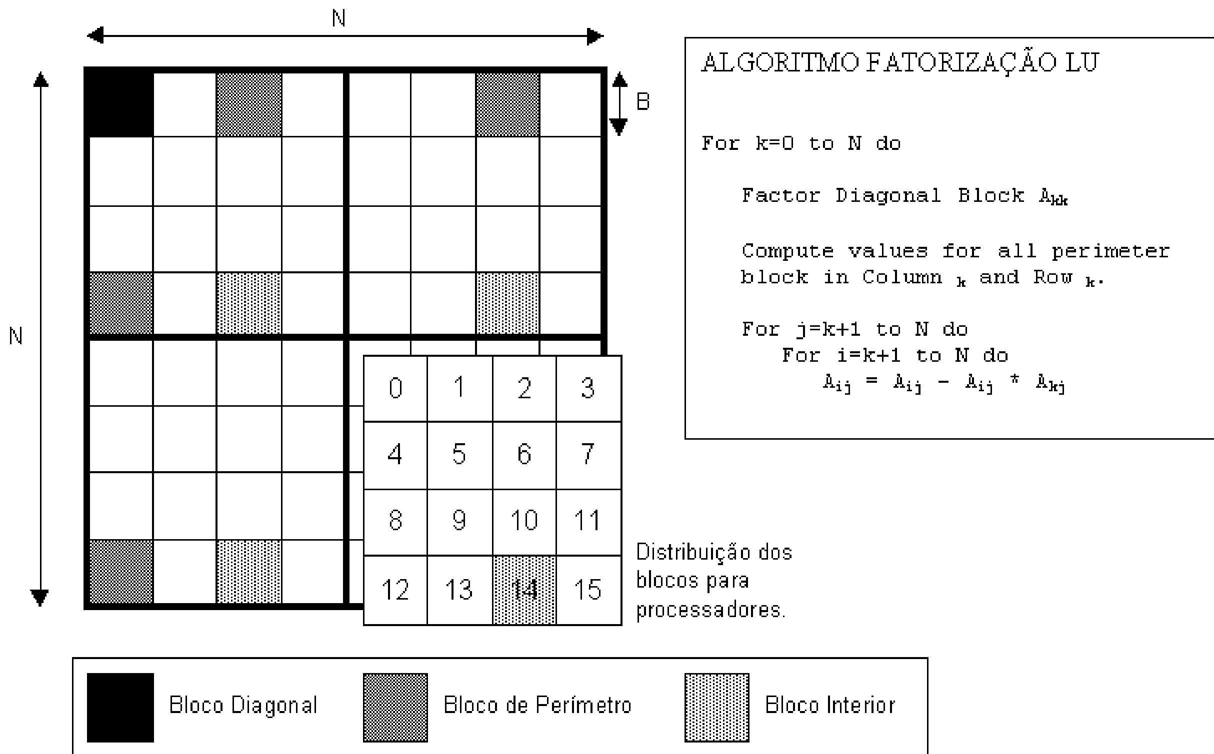


Figura 5.6: Distribuição do conjunto de dados para os processadores do *kernel LU*.

diagonal e só então procederá com a atualização dos blocos de perímetro com os valores do bloco diagonal transmitido. Após a atualização dos blocos de perímetro, o processador proprietário deste bloco iniciará uma nova transmisssão para todos os processadores proprietários de blocos interiores na mesma linha ou coluna do bloco de perímetro.

Para cada bloco interior que um processador detém, o processador deve aguardar pela finalização da transmissão de dois outros blocos. Uma dessas transmissões é proveniente do processador que é proprietário do bloco de perímetro na mesma linha do bloco interior, e a outra transmissão é proveniente do processador que é proprietário do bloco de perímetro na mesma coluna do bloco interior. Somente quando estas duas transmissões finalizarem é que o processador poderá iniciar a atualização do bloco interior. No caso de um processador ser proprietário tanto do bloco interior quanto do bloco de perímetro correspondente à linha ou à coluna do bloco interior, então o processador não necessita aguardar pela transmissão pois o bloco de perímetro poderá ser acessado diretamente da memória local do processador.

Quando existe a necessidade da transferência de dados entre os processadores, um bloco inteiro é transmitido, quer seja um bloco diagonal ou de perímetro. Por isto diz-se

que a unidade de transferência entre os processadores neste *kernel* é um *bloco*, e por este motivo o tamanho do bloco não é alterado à medida em que novos processadores são acrescentados para a resolução do problema.

O conjunto de dados do *kernel LU*, consiste de uma matriz de tamanho $N \times N$. O caminho natural para codificar este programa é a utilização de uma matriz bidimensional, mas em uma linguagem orientada a linhas – como a linguagem C na qual foram desenvolvidos os programas de teste – os elementos componentes do bloco não são armazenados em posições consecutivas de memória. Este fato traz perda de desempenho pois a unidade de transferência do *kernel* é um bloco e isto traz a necessidade de organizar os elementos do bloco em posições contíguas de memória antes de transmiti-los, adicionando custo à transmissão. Para contornar este problema, o conjunto de dados do *kernel* foi organizado em uma matriz quadridimensional, na qual as duas primeiras dimensões especificam o bloco da matriz e as outras duas dimensões identificam os elementos no bloco. A distribuição dos blocos na memória do processador é mostrada na Figura 5.7. Esta alteração foi inicialmente descrita em [Ste96].

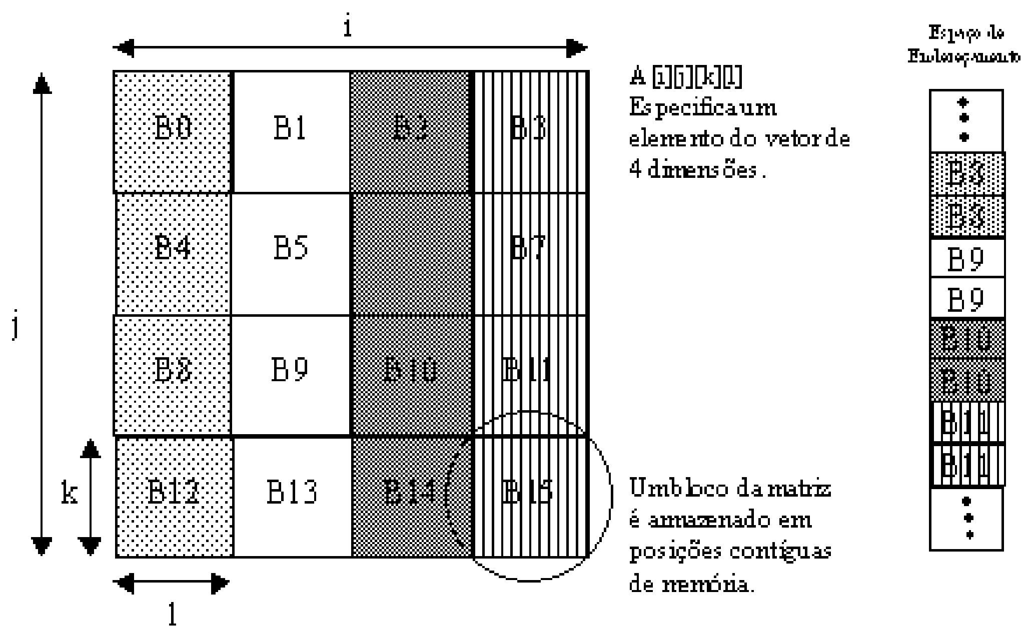


Figura 5.7: Distribuição dos blocos usando uma matriz de quatro dimensões.

Para executar as fases do algoritmo, o programa é dividido em quatro funções. A (i) função `lu0` é responsável pela fatoração do bloco diagonal na primeira fase do algoritmo.

Para o cálculo dos blocos de perímetro, a (ii) função `bdiv` é responsável pelo cálculo do bloco de perímetro que está à direita e na mesma linha do bloco diagonal, e a (iii) função `bmodd` e é responsável pelo cálculo do bloco de perímetro que está abaixo e na mesma coluna do bloco diagonal. Os blocos interiores são calculados pela (iv) função `bmod`.

5.2.1 Implementação *Multithread* do *Kernel* *LU*

O *kernel LU* é composto por 4 funções principais. A função `lu0` é responsável pela fatoração do bloco diagonal. As funções `bdiv` e `bmodd` são responsáveis pelo cálculo dos blocos de perímetro, e a função `bmod` realiza a atualização dos blocos interiores.

Este *kernel* tem três pontos de comunicação. O primeiro ponto de comunicação está localizado após a função `lu0`. Assim que o bloco diagonal é fatorado, este é enviado aos outros processadores que aguardam bloqueados pelo bloco diagonal. O segundo e o terceiro ponto de comunicação estão localizados após as funções `bdiv` e `bmodd`, nas quais os blocos de perímetro são calculados e são posteriormente comunicados aos outros processadores para que todos os proprietários de blocos interiores possam fatorá-los, e os processadores não fiquem bloqueados até que os blocos de perímetro cheguem. Nenhum ponto de comunicação existe após a atualização dos blocos interiores, mas existe um ponto de sincronização entre os processadores (barreira) antes da atualização destes blocos interiores para garantir que todos os processadores iniciem esta fase ao mesmo tempo. Como o número de blocos interiores é maior que o número de blocos diagonais e de perímetro, esta tornou-se a fase que demanda maior tempo de processamento neste algoritmo.

O *kernel LU* foi o programa que exigiu o maior número de pontos de sincronização entre as *threads*. Tal qual o *kernel* FFT, a sincronização entre as *threads* foi realizada através de variáveis condicionais da biblioteca POSIX, mas também foi necessário a utilização de variáveis de controle (*flags*) para evitar que uma *thread* sinalizasse antes que a outra *thread* tivesse atingido o ponto onde esta chama a função `pthread_cond_wait`. Sempre que era realizada uma sinalização com o comando `pthread_cond_sign`, o *flag* também recebia o valor `verdadeiro` que significava que a *thread* já sinalizara. Assim, se uma *thread* que está processando encontrar o *flag* com o valor `verdadeiro`, a *thread* prosseguirá. Caso o *flag* estiver com o valor `falso`, a *thread* aguardará bloqueada em um comando `pthread_cond_wait`.

Para a implementação *multithread* do *kernel LU* o processamento foi feito de ma-

neira semelhante à utilizada no *kernel* FFT, no qual o processamento foi dividido em duas funções que realizam tarefas de comunicação e computação. Estas funções são, respectivamente, `lu_comm` e `lu_compute`. Também semelhante ao *kernel* FFT as *threads* foram criadas com o atributo `detachstate` designado a `PTHREAD_CREATE_JOINABLE` para que o programa principal não finalizasse até que as duas *threads* tivessem terminado o processamento.

Segundo [SJJ94] o *kernel LU* tem melhor desempenho ao trabalhar com blocos de 8×8 ou 16×16 elementos. Isto deve-se ao fato de blocos com estes tamanhos ajustarem-se por completo na memória *cache* primária dos processadores, e o tempo de acesso a esta memória é menor que o tempo de acesso à memória principal. Este fato não favoreceu o desenvolvimento da versão *multithread* descrita nesta dissertação, pois um bloco de 64 (8×8) elementos era processado muito rapidamente, antes que finalizasse o *quantum* de tempo da *thread*. Assim a *thread* de comunicação não competia pelo processador com a *thread* de computação, e, conseqüentemente, não havia sobreposição da comunicação com computação. Por isto houve a necessidade de explicitamente forçar as *threads* a liberarem o processador através do comando `pthread_yield`.

Para isto algumas alterações foram realizadas no programa, as quais passa-se a descrever. O comando bloqueante `MPI_Send` foi substituído por uma comunicação não bloqueante seguida de um comando `pthread_yield`, como mostrado abaixo:

```
MPI_Isend(...,&requisicao);
pthread_yield();
MPI_Wait(&requisicao,&estado);
```

Também o comando bloqueante `MPI_Recv` foi substituído pelo conjunto de comandos abaixo:

```
MPI_Irecv(..., &requisicao);
pthread_yield();
MPI_Wait(&requisicao,&estado);
```

O comando `MPI_Recv` e o comando `MPI_Irecv` seguido por um `MPI_Wait` são equivalentes, mas após um envio ou recebimento não bloqueante utilizava-se o comando `pthread_yield` para forçar a *thread* a perder o seu *quantum* de tempo de processamento, para que a *thread* de computação realizasse a sua tarefa enquanto a mensagem

era enviada ou recebida. Também foi necessário acrescentar um comando `pthread_yield` no interior das funções `bdiv` e `bmodd` para que estas retornassem o processador à *thread* de comunicação para que a *thread* finalizasse este recebimento ou envio das mensagens citadas acima. Mas para blocos maiores (64×64) não houve a necessidade deste artifício, pois os blocos eram suficientemente grandes para haver competição entre as *threads*.

No interior do programa existem vários laços nos quais são calculados os endereços de memória dos blocos da matriz principal para serem fatorados pelas funções `lu0`, `bdiv`, `bmodd` e `bmod`. Estes laços tiveram de ser duplicados para constarem tanto da *thread* de computação quanto da *thread* de comunicação. Isto causou o aumento do número de instruções executadas pelos processadores. O impacto desta duplicação será discutido no capítulo dedicado aos resultados obtidos nos experimentos.

Neste *kernel* só foi possível aproveitar a característica das *threads* de sobreposição da comunicação com computação na fase em que os blocos de perímetro são calculados. Isto deve-se ao fato de que nenhuma comunicação existe no cálculo do bloco diagonal e a fase de atualização dos blocos de perímetro só pode ser iniciada após a chegada do bloco diagonal e a fase de atualização dos blocos interiores só tem início após a chegada de todos os blocos de perímetro.

Uma melhoria que pode ser realizada é reorganizar o programa de maneira a permitir que os blocos interiores que já receberam os blocos de perímetro correspondentes a linha e coluna, iniciem o cálculo desses blocos interiores, sem a necessidade de esperar por blocos de perímetro que não pertencem a sua linha e coluna.

Não foi necessário sincronização para que as *threads* iniciem o processamento ao mesmo tempo, pois a sincronização existente onde dá-se a comunicação já garante a coerência do resultado.

5.3 Ordenação *Radix*

Funções de ordenação são utilizados em uma grande quantidade de aplicações, incluindo sistemas de gerenciamento de banco de dados e aplicações aeroespaciais. O *kernel* Radix é um algoritmo de ordenação de números inteiros descrito em [Ble91]. O estudo deste *kernel* é interessante porque requer a movimentação de uma grande quantidade de dados entre os processadores durante cada fase da execução da ordenação.

A cada processador é designado o mesmo número n de chaves para serem ordenadas. Cada iteração do algoritmo é composta por três fases: Na primeira fase de cada iteração um processador passa sobre todas as chaves em seu poder e constrói um *histograma local* que representa a distribuição dos dígitos na porção de chaves em seu poder. Na segunda fase, todos os processadores combinam os seus histogramas locais em um *histograma global* que indica a soma da distribuição total de dígitos nas chaves. Na terceira fase os processadores fazem uso do histograma global para fazer a permutação das chaves que cada processador possui. Assim, as chaves são escritas uma a uma em um vetor e são enviadas ao processador destino, e o algoritmo continua em outras iterações até que todos os dígitos tenham sido verificados e a ordenação tenha sido finalizada. O algoritmo do *kernel* Radix é mostrado na Figura 5.8.

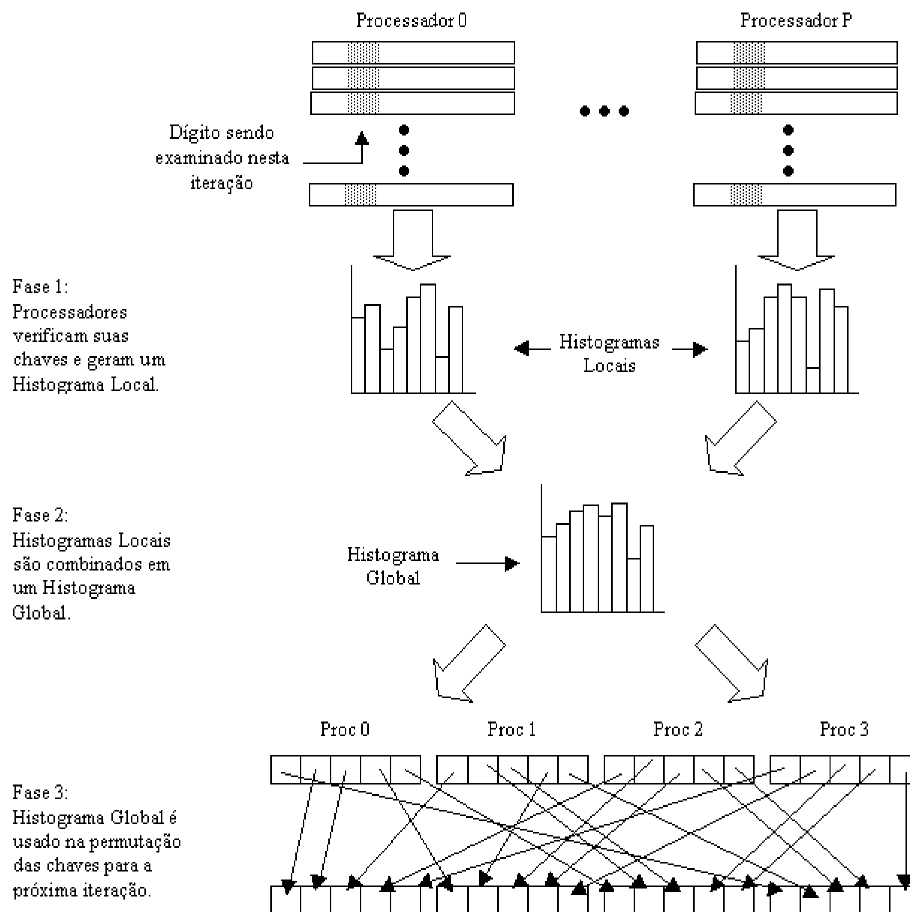


Figura 5.8: Algoritmo do *kernel* Radix.

Em contraste com outros algoritmos de ordenação, *radix* não utiliza apenas comparações para determinar a ordem das chaves. O algoritmo *radix* analiza as chaves como inteiros de b–bits. Números de ponto flutuante também podem ser ordenados por este

método por causa da representação de ponto flutuante [JD97].

O algoritmo base examina as chaves que serão ordenadas em blocos de r–bits a cada iteração, iniciando no bloco de r–bits menos significativos em cada chave. A cada iteração do laço principal do programa, o algoritmo ordena as chaves de acordo com o bloco de r–bits corrente na iteração. O algoritmo requer b/r passos.

No decorrer do algoritmo, primeiramente será determinada a posição (*rank*) de cada chave – a posição da chave na ordem de saída – e então as chaves serão trocadas nas respectivas posições determinadas pelo *rank*. Para a definição da posição de cada chave, o algoritmo executa a operação $[(chave >> deslocamento)\ \&\ mascara]$. A $chave$ é o número que está sendo analizado, $deslocamento$ é dado pelo produto de $iteracao$ e r. $iteracao$ é um número variando de 0 a b/r, e $mascara$ é dado por $(b-1)$.

O conjunto de dados do *kernel radix* consiste de chaves a serem ordenadas, as quais são armazenadas em vetores de inteiros, e um segundo vetor de inteiros que atua como uma estrutura intermediária à medida em que a ordenação progride. As chaves são repetidamente ordenadas do vetor de inteiros para a estrutura intermediária e vice-versa durante as iterações. O tamanho destas estruturas é igual ao número de chaves a processar. Também são necessárias estruturas para armazenar o histograma local e global. Cada processador mantém o próprio histograma local e uma cópia do histograma global também é mantida localmente.

Um fator importante que influência no desempenho do algoritmo *radix* é o valor de r utilizado na ordenação. O algoritmo executa uma iteração para cada bloco de r–bits nas chaves, o que indica que quanto maior o valor de r, menor será o número de iterações requeridas. Por outro lado, quanto maior o valor de r, maior será o espaço requerido para as estruturas que armazenam o histograma local e global [Ble91].

A comunicação neste algoritmo ocorre principalmente na terceira fase de cada iteração, quando as chaves são permutadas entre os processadores, e esta comunicação é do tipo *todos-para-todos*. A comunicação ocorre também na segunda fase do algoritmo, quando os histogramas locais são agrupados no histograma global, mas a comunicação que ocorre na segunda fase a comunicação é menos intensa que na terceira fase.

Durante a fase de permutação das chaves o programa agrupa as chaves destinadas a um mesmo processador em posições consecutivas de memória. Desta maneira pode-se enviar as chaves destinadas a um mesmo processador em uma única transferência. Este

comportamento do programa é monstrado na Figura 5.9.

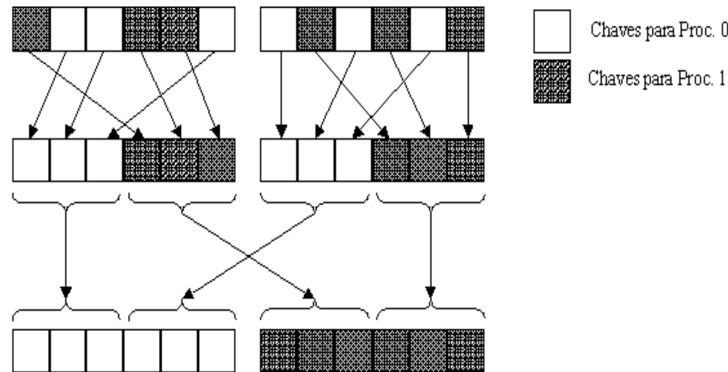


Figura 5.9: Fase de permutação das chaves.

5.3.1 Implementação *Multithread* do *Kernel Radix*

O *kernel Radix* é composto principalmente de três funções que realizam as três fases do algorimo já descritas nesta dissertação.

A função `histogram` é responsável por calcular o histograma local e nesta fase existe apenas processamento local.

A segunda função é denominada `all_scan_bucket` e realiza uma varredura (*scan*) nos histogramas locais de todos os processadores de maneira a agrupá-los em um único histograma global. Este agrupamento é realizado com operações de adição nos elementos dos histogramas locais e a compilação do histograma global é iniciada no processador 0 e é finalizada no processador $P-1$. Após a compilação do histograma global apenas o processador $P-1$ detém este histograma e envia-o para todos os outros processadores. Nesta fase conhece-se o número de mensagens que é $2(P-1)$ e o tamanho das mensagens que é $2^r * sizeof(int)$.

A terceira função do programa é denominada `all_coalesce` e nesta fase as chaves que estão de posse dos processadores são trocadas de acordo com o histograma global. Esta função realiza a ordenação do bloco de $r-bits$ corrente da iteração e envia as chaves aos processadores destino através de uma função denominada `handle_outgoing_coalesce_msgs`, e recebe as chaves provenientes dos outros processadores fazendo uso de uma função denominada `handle_incoming_coalesce_msgs`. Nestas funções é utilizada a comunicação do tipo não bloqueante e esta fase do algoritmo é a fase que apresenta o maior volume

de comunicação.

Neste algoritmo existe um ponto de sincronização entre os processadores que dá-se através de uma barreira localizada após o cálculo do histograma local. Isto é necessário pois antes de compilar o histograma global todos os processadores devem estar de posse dos histogramas locais devidamente calculados.

Para o desenvolvimento da versão *Multithread* deste *kernel* foi utilizada uma abordagem diferente da abordagem utilizada nos *kernels LU* e FFT com relação às tarefas designadas a cada *thread*.

Neste *kernel* foram criadas três *threads* denominadas `radix_comput`, `radix_comm_incoming` e `radix_comm_outgoing`. A *thread* `radix_comput` foi designada a realizar a criação do histograma local, criação do histograma global e o cálculo da distribuição das chaves para os processadores. É importante notar que esta *thread* também realiza comunicação na fase em que o histograma global é compilado, portanto neste *kernel* existe comunicação em todas as *threads*.

A comunicação existente na distribuição das chaves para os processadores foi distribuída em duas outras *threads*, cita-se `radix_comm_incoming` e `radix_comm_outgoing` que respectivamente ficaram responsáveis por receber as mensagens provenientes dos outros processadores e enviar as chaves aos demais processadores.

Optou-se por esta abordagem pelo fato de este *kernel* apresentar pouca computação (granularidade pequena) e porque o algoritmo original apresentava duas fases distintas que são o envio das chaves para os outros processadores e o recebimento das chaves provenientes dos demais processadores, e estas fases ocorriam em momentos distintos do programa. Buscou-se então sobrepor a comunicação que existe no envio das mensagens com a comunicação que ocorre no recebimento das mensagens.

Neste *kernel* foi necessário a inclusão de barreiras nas *threads* para garantir que todas as *threads* estejam sincronizadas a cada iteração do algoritmo, e o programa estava otimizado para enviar o maior número de chaves possíveis em um bloco de 1500 bytes, que é o tamanho máximo de um pacote IP em rede *Ethernet*. Também semelhante ao *kernel* FFT e *LU* as *threads* foram criadas com o atributo `detachstate` designado a `PTHREAD_CREATE_JOINABLE` para que o programa principal não finalizasse até que as *threads* tivessem terminado o processamento.

CAPÍTULO 6

METODOLOGIA DE MEDIÇÃO DE DESEMPENHO

6.1 Métricas de Desempenho

Existe uma grande diversidade de índices de desempenho que são utilizados para a aferição e avaliação do desempenho de sistemas paralelos. A seguir são discutidas algumas destas métricas que são (i) o tempo de execução, (ii) a aceleração, e (iii) a eficiência. Após a apresentação das métricas, é apresentada a metodologia que foi empregada nos experimentos.

Tempo de Execução. *Tempo total de execução* é o tempo que seria medido por um relógio externo que registra o tempo desde o início até o final da execução de um programa e é dado por $T_{(N,p)}$, onde N é o tamanho do problema e p é o número de processadores.

Aceleração. O fator de aceleração (*Speedup*) [Dan93] de um programa paralelo é dado por:

$$S_{(N,p)} = \frac{T_{(N,1)}}{T_{(N,p)}} \tag{6.1}$$

Onde $T_{(N,1)}$ é o tempo de execução do programa utilizando-se apenas 1 processador (tempo sequencial) e $T_{(N,p)}$ é o tempo de execução do mesmo programa em p processadores.

Eficiência. A eficiência [Dan93] de um programa paralelo é definida como a razão entre o fator de aceleração e o número de processadores.

$$E_{(N,p)} = \frac{S_{(N,p)}}{p} = \frac{T_{(N,1)}}{p.T_{(N,p)}} \tag{6.2}$$

Granularidade. A granularidade é o volume de processamento realizado pela tarefa, em relação ao volume de comunicação existente na tarefa, e uma tarefa é uma unidade computacional análoga a um processo em ambiente Unix.

6.2 Metodologia dos Experimentos

Para a tomada dos tempos de execução dos programas e posterior análise das métricas de desempenho, foram utilizadas duas versões de cada um dos *kernels* constantes desta dissertação. A primeira foi uma versão paralela na qual deu-se prioridade à comunicação não bloqueante para procurar sobrepor a comunicação com computação, mas sem a utilização de múltiplas *threads*. A segunda versão utilizada foi um programa utilizando múltiplas *threads*. Ambas as versões utilizavam a biblioteca MPI para comunicação entre os processadores.

Foram realizados vários testes, em um total de 20 a 30 execuções dos programas, e após a tomada dos tempos de execução, era calculada a média dos tempos de execução para minimizar eventuais distorções inseridas por outras aplicações e pelo sistema operacional que pudessem concorrer com os *kernels* pelo processador ou por influências do sistema operacional que pudessem distorcer o tempo real de execução dos *kernels*. Estas execuções eram realizadas em seqüência e sem intervalos entre elas de maneira a manter preenchida a memória *cache* do computador para medir o desempenho máximo de cada *kernel*.

Também dentre os tempos medidos os maiores valores eram descartados, pois conforme [Hél03] os menores tempos geralmente são os mais precisos pois sofrem menor influência do sistema operacional e de outras aplicações.

Toda vez que houve a necessidade de medir o tempo de execução de uma determinada parte dos programas, utilizou-se o método descrito em [Hél03] e que recebeu o nome de *monitor* em [JD98], que consiste em incluir rotinas de tomada de tempo antes e depois de uma comunicação entre os processadores ou antes e depois de uma determinada parte do programa que quer-se avaliar. A diferença entre o tempo final e o tempo inicial era calculada e o resultado era mostrado na tela e anotado para posterior cálculo da média destes tempos. As funções para tomada do tempo de execução utilizadas nos programas constantes desta dissertação são semelhantes as funções mostradas na figura 6.1.

Também para a tomada dos tempos de execução dos programas, buscou-se maximizar o tamanho do problema no qual procurou-se encontrar valores representativos dos programas e buscou-se valores que permitissem uma granularidade maior para permitir a exploração das características das *threads*.

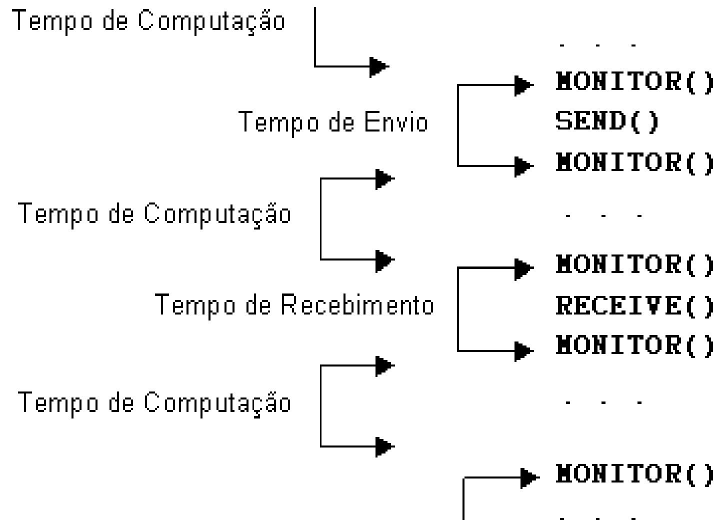


Figura 6.1: Operações do método *monitor* adicionadas aos programas de teste.

6.2.1 Implementação *Multithread*

Ao desenvolver-se as versões *multithread* dos *kernels* foram seguidos alguns passos, os quais passa-se a descrever.

Como partiu-se de versões paralelas dos *kernels* estudados, antes de iniciar a programação *multithread* foi necessário um entendimento prévio do problema. Assim, iniciou-se um estudo do problema solucionado por cada um dos *kernels* e um entendimento dos algoritmos utilizados.

Após, como utilizou-se a abordagem de separar o processamento e a comunicação entre os processadores em duas *threads* distintas, foi necessário um entendimento dos padrões de comunicação existentes nestes programas. A abordagem de usar duas *threads* uma responsável pela comunicação e outra responsável pela computação, foi necessária no desenvolvimento dos programas porque o número de *threads* permitida em um processo é limitado. Alguns pacotes de *threads* trabalham com um número máximo de 128 *threads*. Caso se desejasse dividir o processamento de partes do conjunto de dados a *threads* distintas em um mesmo processador, inclusive com a comunicação envolvida, o número de *threads* disponível poderia ser insuficiente para os conjuntos de dados maiores.

Só então foi possível iniciar a programação da versão *multithread*, e foi necessário sincronizar as *threads* de maneira a que a execução do programa com as duas *threads* fosse equivalente à execução do programa paralelo original. Assim que os cálculos de uma determinada fase do algoritmo terminavam, era necessário informar a outra *thread* que

estava bloqueada aguardando pela sinalização.

Como o bloqueio de uma *thread* não causa o bloqueio de uma outra *thread*, para toda barreira da biblioteca MPI existente nos programas, foi necessário incluir uma sincronização na outra *thread* para que esta não continuasse o processamento enquanto a *thread* inicial permanecia bloqueada pela barreira do MPI. Esta sincronização nas *threads* foi chamada de *Barreira nas threads*, enquanto que a barreira do MPI foi chamada de *Barreira nos processadores.*

Também buscou-se manter o tamanho das mensagens em um valor inferior a 1500 *bytes*, que é o tamanho máximo de um pacote em rede *Ethernet* [Mat97], para evitar o *overhead* gerado no envio de pacotes superiores a este valor. Mas isto só foi realizado desde que não houvesse a necessidade de reorganizar os dados em posições contíguas de memória.

CAPÍTULO 7

RESULTADOS

Neste capítulo é descrito o ambiente de testes no qual realizou-se os experimentos desta dissertação e mostra-se os resultados obtidos para os *kernels* FFT, *LU* e Radix.

7.1 Ambiente de Testes

Nos experimentos realizados para a avaliação do desempenho dos programas de testes foi utilizado o laboratório da Universidade Federal do Paraná cujos nodos possuem as características abaixo:

- 16 nós de processamento com um processador AMD Duron de 1200 MHz, 128 Mbytes de memória RAM, 128 Kb de memória *cache* (L1) e 64 Kb de memória *cache* (L2).
- Rede de interconexão: Fast-Ethernet.
- Sistema Operacional: Linux Kurumin 4.0, *Kernel* 2.4.25.
- Implementação do padrão MPI: MPICH 1.2, disponível em `ftp://ftp.mcs.anl.gov/pub/mpi/mpich.tar.gz`.

7.2 *Fast Fourier Transform* – FFT

O *kernel* FFT apresentou ganho de desempenho ao utilizar-se *threads* em conjunto com a biblioteca MPI, como mostrado no gráfico da figura 7.1 e na Tabela 7.1. Foram realizados testes (i) na versão paralela original (paralela), e (ii) em uma versão *multithread*. As duas versões utilizam a biblioteca MPI para comunicação entre os processadores. O gráfico 7.1 mostra no eixo vertical o tempo de execução do programa, e no eixo horizontal o número de processadores, e mostra o tempo de execução para o *kernel* FFT quando o número de processadores varia de 2 até 16 processadores. Neste experimento o conjunto de dados foi mantido fixo em 1048576 números complexos de dupla precisão (*double*). Foi escolhido o tamanho do conjunto de dados de 1048576 números complexos pois este foi o tamanho

máximo de memória que pode-se alocar nos computadores disponíveis. Conjunto maiores que este causavam erros de alocação de memória, principalmente para 2 processadores.

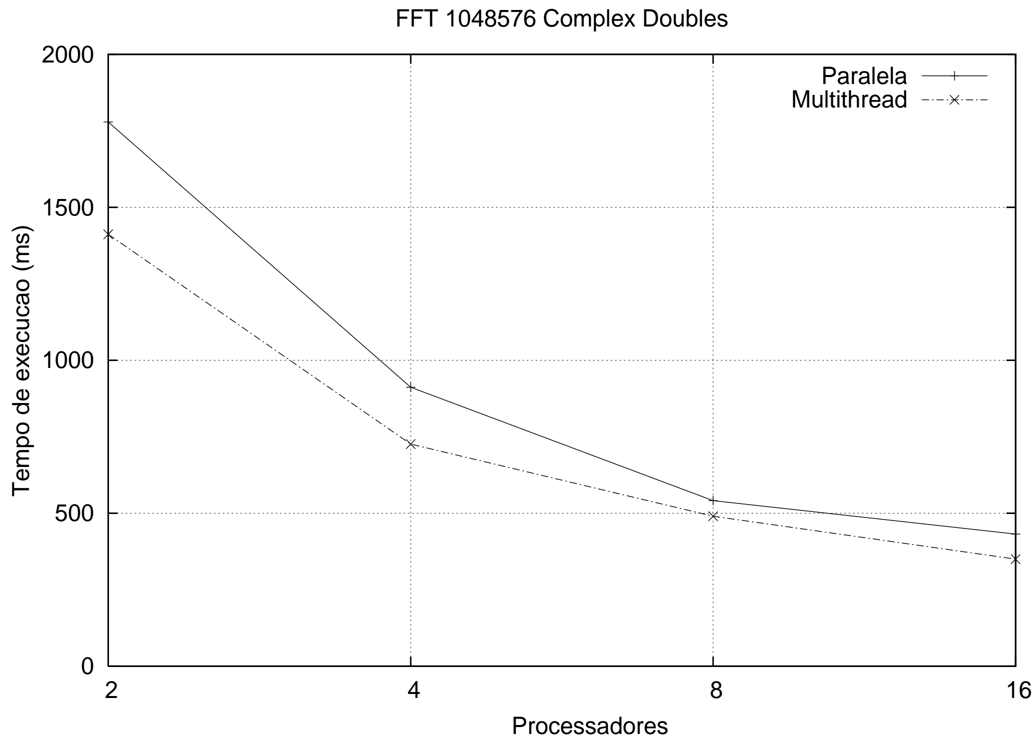


Figura 7.1: Resultados obtidos no *kernel* FFT para um conjunto de dados de 1048576 números complexos.

Versão	$P = 2$	$P = 4$	$P = 8$	$P = 16$
Paralela	1779	912	541	432
Multithread	1412	726	490	350
Percentual	20,6%	20,4%	9,5%	19%
Aceleração	1,25	1,25	1,10	1,23

Tabela 7.1: Tempos de execução em milissegundos (ms) e Aceleração para o *kernel* FFT para um conjunto de dados de 1048576 números complexos.

Na tabela 7.1 é mostrada a aceleração obtida com a versão *multithread* em relação à versão paralela, que mostra o ganho de desempenho obtido ao utilizar-se *threads* para explorar a sobreposição da comunicação com a computação.

Na tabela 7.2 é mostrada a aceleração e a eficiência obtida no *kernel* FFT. Nesta dissertação foram realizados experimentos com 2,4,8 e 16 processadores. Não foram realizados testes com um único processador (versão serial) pois os programas utilizados eram versões paralelas, e a versão serial não estava disponível. Na tabela 7.2 a aceleração foi

Aceleração	Paralela	*Multithread*	Eficiência	Paralela	*Multithread*
$S_{f_4} = T_2/T_4$	1,95	1,94	E_{f_4}	0,48	0,48
$S_{f_8} = T_2/T_8$	3,28	2,88	E_{f_8}	0,41	0,36
$S_{f_{16}} = T_2/T_{16}$	4,11	4,03	$E_{f_{16}}$	0,25	0,25

Tabela 7.2: Aceleração e Eficiência obtidas para o conjunto de dados de 1048576 números complexos, em relação a 2 processadores.

Aceleração	Paralela	*Multithread*	Eficiência	Paralela	*Multithread*
$S_{v_4} = T_2/T_4$	1,95	1,94	E_{v_4}	0,48	0,48
$S_{v_8} = T_4/T_8$	1,68	1,48	E_{v_8}	0,21	0,18
$S_{v_{16}} = T_8/T_{16}$	1,25	1,40	$E_{v_{16}}$	0,07	0,08

Tabela 7.3: Aceleração e Eficiência obtidas para o conjunto de dados de 1048576 números complexos.

calculada em relação a 2 processadores, e a eficiência foi denominada E_f pois também foi calculada em relação a 2 processadores. Na tabela 7.3 é mostrada a aceleração e a eficiência para o *kernel* FFT, mas nesta tabela a aceleração e a eficiência foram calculadas com o número de processadores variável, e a eficiência foi denominada E_v.

Versão	$P = 2$	$P = 4$	$P = 8$	$P = 16$
Paralela	23	66	230	321
Multithread	19	53	202	288
Percentual	17,4%	19,7%	12,2%	10,3%

Tabela 7.4: Tempos de execução em *ms* para o *kernel* FFT com variação no tamanho do conjunto de dados.

No gráfico da figura 7.2 e na Tabela 7.4 é mostrado o tempo de execução para o *kernel* FFT quanto o tamanho do conjunto de dados foi variado. Como o *kernel* FFT trabalha com o conjunto de dados organizado em uma matriz de $\sqrt{N} \times \sqrt{N}$ elementos, a variação do tamanho do conjunto de dados teve de ser quadruplicada conforme aumentava-se o número de processadores, sendo realizados testes com 16384 números complexos em 2 processadores, 65536 números complexos em 4 processadores, 262144 números complexos em 8 processadores, e 1048576 números complexos em 16 processadores.

O ganho obtido com a versão *multithread* em relação à versão original comprova que foi acertada a abordagem utilizada na implementação *multithread* do programa de transpor as linhas da matriz separadamente e efetuar os cálculos das FFTs unidimencionais à medida

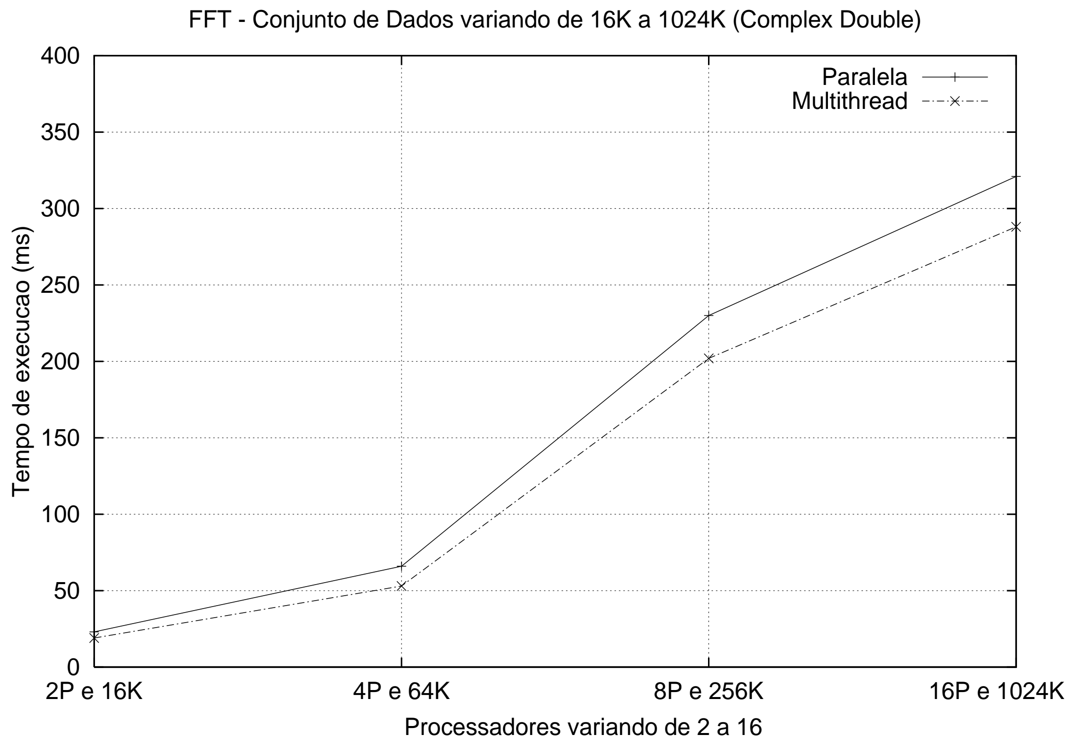


Figura 7.2: Resultados obtidos no *kernel* FFT com variação no tamanho do conjunto de dados.

em que as linhas eram transpostas.

Também contribuiu para a obtenção do ganho de desempenho mostrado nos gráficos 7.1 e 7.2, o método utilizado na transposição das linhas da matriz. Este método foi descrito em detalhe no capítulo 5 e consiste em efetuar a transposição das linhas da matriz na própria função da biblioteca MPI responsável pela comunicação entre os processadores nesta fase, pois este método dispensava a necessidade de processamento na reorganização dos dados recebidos. Isto contribuiu para que a *thread* de comunicação que realizava as transposições não concorresse em computação com a *thread* de computação que realizava as FFTs unidimensionais.

O *kernel* FFT apresentou um aumento significativo no tempo de execução para o conjunto de dados variável, como é mostrado no gráfico da figura 7.2. Isto é devido ao fato de que neste experimento o número de processadores aumentava em duas vezes, mas o tamanho do conjunto de dados aumentava em quatro vezes, o que causou um aumento desproporcional do conjunto de dados em relação ao número de processadores. O dobro de processadores não foi suficiente para manter constante o tempo de execução neste algoritmo, pois com o aumento do tamanho do conjunto de dados aumenta-se a

comunicação existente no programa [JD98]. Este aumento na comunicação e aumento no número de elementos da matriz calculados em cada processador causou a elevação do tempo de execução do programa. A versão *multithread* apresentou melhor desempenho que a versão paralela original, os ganhos de desempenho são de 20,6%, 20,4%, 9,5% e 19% para 2,4,8 e 16 processadores respectivamente.

7.3 Fatorização *LU*

O *kernel LU* não apresentou ganho de desempenho com a utilização de *threads* em conjunto com a biblioteca MPI, como pode-se verificar no gráfico da figura 7.3 e na Tabela 7.5, que mostram (i) a versão paralela original (paralela), e (ii) uma versão *multithread*.

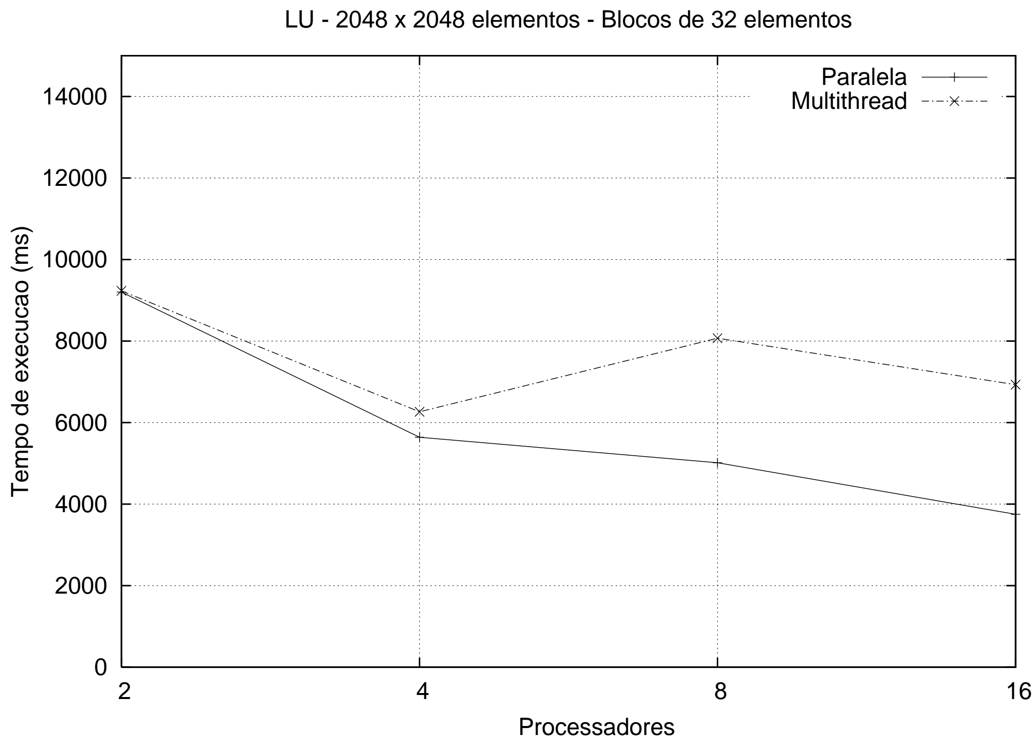


Figura 7.3: Resultados do *kernel LU* para um conjunto de dados de 2048×2048 elementos.

No gráfico da figura 7.3 e na Tabela 7.5 é mostrado o tempo de execução para o *kernel LU* em função do número de processadores, que foi variado de 2 até 16 processadores, enquanto que o conjunto de dados foi mantido fixo em 2048 elementos. Foi escolhido o tamanho do conjunto de dados de 2048 elementos pois este foi o tamanho máximo de memória que pode-se alocar nos computadores disponíveis. Matrizes maiores que 2048 elementos causavam erros de alocação de memória, principalmente para 2 processadores.

Versão	$P = 2$	$P = 4$	$P = 8$	$P = 16$
Paralela	9204	5639	5014	3751
Multithread	9233	6265	8070	6930
Percentual	0,31%	10%	37,86%	45,87%
Aceleração	0,99	0,90	0,62	0,54

Tabela 7.5: Tempos de execução em *ms* e Aceleração para o *kernel LU* para um conjunto de dados de 2048×2048 elementos.

Na tabela 7.5 é mostrada a aceleração obtida com a versão *multithread* em relação à versão paralela. Os valores abaixo de 1 mostram que neste *kernel* houve perda de desempenho ao utilizar-se *threads* em conjunto com MPI.

Aceleração	Paralela	*Multithread*	Eficiência	Paralela	*Multithread*
$S_{f_4} = T_2/T_4$	1,63	1,47	E_{f_4}	0,40	0,36
$S_{f_8} = T_2/T_8$	1,83	1,14	E_{f_8}	0,22	0,14
$S_{f_{16}} = T_2/T_{16}$	2,45	1,33	$E_{f_{16}}$	0,15	0,08

Tabela 7.6: Aceleração e Eficiência obtidas para o conjunto de dados de 2048×2048, em relação a 2 processadores.

Aceleração	Paralela	*Multithread*	Eficiência	Paralela	*Multithread*
$S_{v_4} = T_2/T_4$	1,63	1,47	E_{v_4}	0,40	0,36
$S_{v_8} = T_4/T_8$	1,12	0,77	E_{v_8}	0,14	0,09
$S_{v_{16}} = T_8/T_{16}$	1,33	1,16	$E_{v_{16}}$	0,08	0,07

Tabela 7.7: Aceleração e Eficiência obtidas para o conjunto de dados de 2048×2048.

Na tabela 7.6 é mostrada a aceleração e a eficiência obtida no *kernel LU*. A aceleração foi calculada em relação a 2 processadores, e a eficiência foi denominada E_f pois também foi calculada em relação a 2 processadores. Não foram realizados testes com um único processador (versão serial) pois o programa utilizado era uma versão paralela, e a versão serial não estava disponível. Na tabela 7.7 é mostrada a aceleração e a eficiência para o *kernel LU*, mas nesta tabela a aceleração e a eficiência foram calculadas com o número de processadores variável, e a eficiência foi denominada E_v.

No gráfico da figura 7.4 e na Tabela 7.8 é mostrado o tempo de execução para o *kernel LU*, e neste experimento tanto o número de processadores quanto o conjunto de dados foi variado. O conjunto de dados utilizado foi de 2048 elementos para 2 processadores, 2560 elementos para 4 processadores, 3232 elentos para 8 processadores e 4096 elementos para

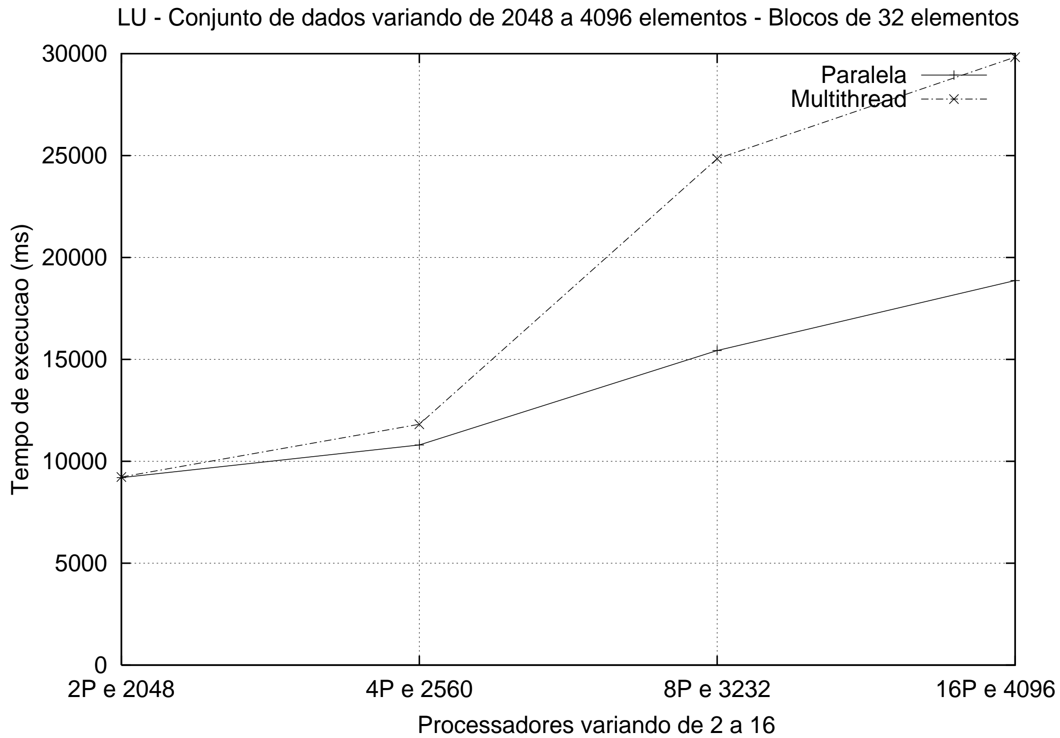


Figura 7.4: Resultados obtidos no *kernel LU* com variação no número de processadores e no tamanho do conjunto de dados.

Versão	$P = 2$	$P = 4$	$P = 8$	$P = 16$
Paralela	9202	10802	15435	18866
Multithread	9227	11813	24845	29828
Percentual	0,27%	8,55%	37,87%	36,75%

Tabela 7.8: Tempos de execução em *ms* para o *kernel LU* com variação no número de processadores e no tamanho do conjunto de dados.

16 processadores. Estes valores para o conjunto de dados do *kernel LU* foram escolhidos pelo fato de que este *kernel* apresenta um crescimento do tempo de execução proporcional a $O(n^3)$. Os valores do conjunto de dados foram arredondados de maneira a permitir a organização do conjunto de dados em blocos de 32×32 elementos.

Como discutido no capítulo 5, para blocos de tamanho inferiores a 32×32 elementos foi necessário explicitamente forçar a *thread* a trocar de contexto. Isto foi necessário pois caso fosse deixada livre a troca de contexto entre as *threads*, a *thread* de computação realizava cálculos em vários blocos sem a ocorrência da troca de contexto e, conseqüentemente não havia sobreposição de comunicação com computação. O desempenho apresentado nos gráficos das figuras 7.3 e 7.4 foi possível com a utilização da troca de contexto explícita

entre as *threads*.

No desenvolvimento do *kernel LU*, as *threads* foram divididas em uma *thread* responsável pela computação e outra *thread* responsável pela comunicação entre os processadores. A fase que demanda o maior tempo de processamento é a fase na qual os blocos interiores são calculados, e esta tarefa é realizada na função `bmod`, quando não existe comunicação entre os processadores. O fato desta fase ter sido implementada dentro da *thread* de computação contribuiu para a perda de desempenho apresentada por este *kernel*, pois levava a trocas de contexto desnecessárias entre a *thread* de computação e a *thread* de comunicação, uma vez que nesta fase do algoritmo dever-se-ia dar exclusividade à *thread* de computação e evitar a concorrência entre as *threads*. O pacote de *threads* utilizado não permitia dar exclusividade a uma *thread* uma vez iniciado o processamento das *threads*.

No programa paralelo original do *kernel LU*, existem laços que calculam os endereços dos blocos que estão sendo processados e o número de mensagens que serão enviadas pelo processador. Na implementação *multithread* deste *kernel*, como as *threads* eram divididas em *thread* de computação e *thread* de comunicação, houve a necessidade de duplicar os laços que calculam os endereços dos blocos correntes e o número de mensagens. Assim, a *thread* de comunicação teve de realizar processamento adicional para computar estes valores, o que também contribui para a perda de desempenho.

A perda de desempenho mostrada nos gráficos das figuras 7.3 e 7.4, deve-se ao fato de que a sobreposição da comunicação com a computação só era possível em uma das três fases do algoritmo. A primeira fase é mandatória para as demais fases, a segunda fase onde é possível a sobreposição da comunicação com a computação é pequena em relação as outras fases e na última fase não existe comunicação entre os processadores.

Versão	B=16	B=32	B=64	B=128
Paralela	6113	5055	6609	9967
Multithread	8637	8044	8890	10242
Percentual	29,2%	37,15%	25,65%	0,03%

Tabela 7.9: Tempos de execução em *ms* para o *kernel LU* para 8 processadores com variação no tamanho dos blocos.

Ao contrário do que é reportado em outros trabalhos que utilizam o *kernel LU* [JD96, SME+95, Ste96], nos quais o tamanho do bloco que apresentava o melhor desempenho era 8×8 ou 16×16 elementos, o tamanho de bloco que apresentou o melhor

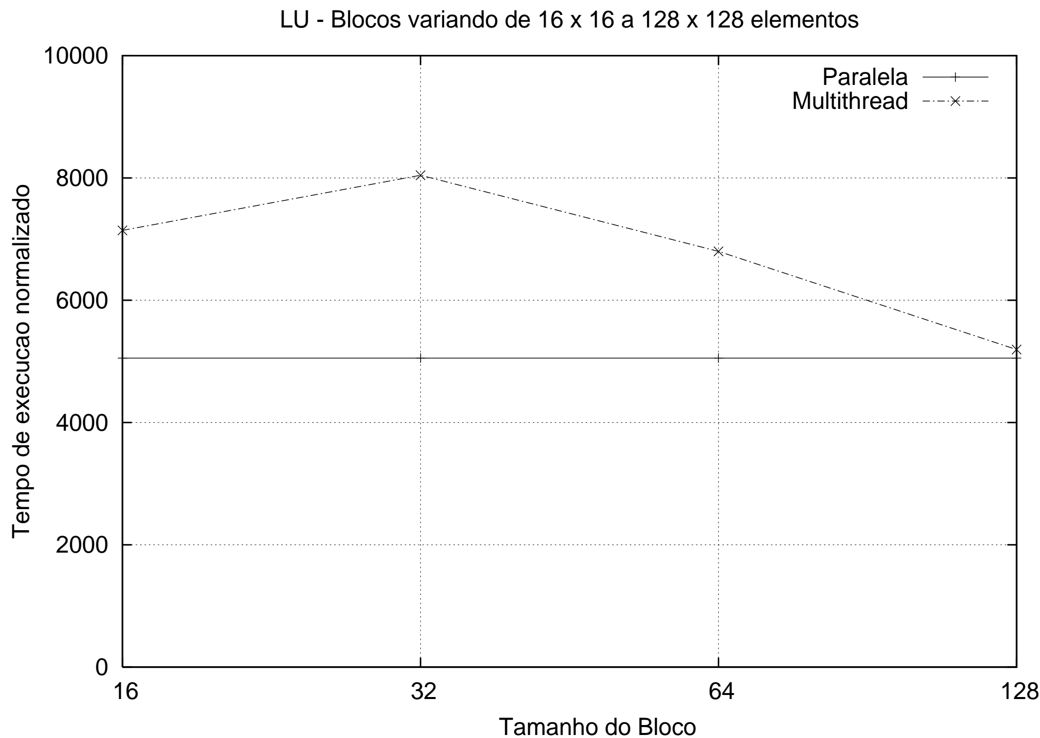


Figura 7.5: Variação no tamanho do bloco no *kernel LU*.

desempenho na versão utilizando a biblioteca MPI foi 32×32 elementos, portanto este foi o tamanho do bloco escolhido para realizar os experimentos. Também foram realizados testes com o tamano do bloco variando desde 16×16 elementos até 128×128 elementos, sendo que nestes experimentos foi observada uma tendência de que quanto maior o tamanho do bloco, que significa aumentar a granularidade do programa, a versão *multithread* apresentou perda menor de desempenho em relação à versão paralela original. Mesmo assim o desempenho apresentado para blocos maiores foi insuficiente para superar o desempenho da versão paralela original.

Na Tabela 7.9 são mostrados os valores obtidos para o *kernel LU* ao utilizar-se 8 processadores e tamanho do conjunto de dados de 2048×2048. O tamanho do bloco (B) foi variado de 16×16 elementos até 128×128 elementos, e no gráfico da Figura 7.5 os valores foram normalizados. Pode-se observar que o melhor desempenho obteve-se para o tamanho do bloco de 32×32 elementos, mas o tamanho de bloco em que a versão *multithread* mais se aproximou da versão paralela original foi para o tamanho do bloco de 128×128 elementos.

O trabalho [AJK+91] descreve resultados semelhantes para o *kernel LU*, que apre-

sentou perda de desempenho quando o número de ciclos do processador necessários para realizar a troca de contexto entre *threads* é igual ou superior a 16 ciclos. Apesar de os processadores utilizados no trabalho [AJK+91] realizarem trocas de contexto em hardware, os resultados obtidos foram semelhantes aos obtidos nesta dissertação para o *kernel LU*.

7.4 Ordenação Radix

O *kernel* Radix apresentou ganho de desempenho ao utilizar-se *threads* em conjunto com a biblioteca MPI, como mostrado no gráfico da figura 7.6 e na Tabela 7.10, com (i) a versão paralela original (paralela), e (ii) a versão *multithread*.

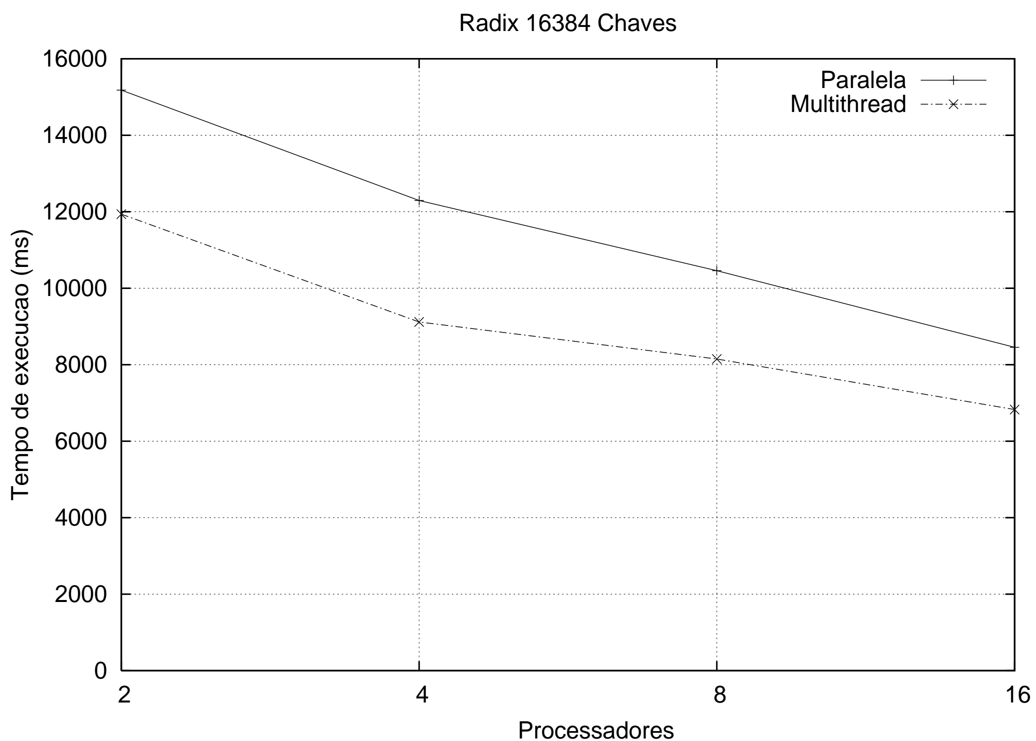


Figura 7.6: Resultados obtidos no *kernel* Radix para um conjunto de dados de 16384 chaves.

Na tabela 7.10 é mostrada a aceleração obtida com a versão *multithread* em relação à versão paralela, que mostra aceleração entre 1,23 e 1,34, ao utilizar-se *threads*, pois existe a sobreposição da comunicação com computação.

No gráfico da figura 7.6 e na Tabela 7.10 é mostrado o tempo de execução para o *kernel* Radix quando o número de processadores era variado de 2 até 16 processadores. Neste experimento o conjunto de dados foi mantido fixo em um valor de 16384 chaves de

Versão	$P=2$	$P=4$	$P=8$	$P=16$
Paralela	15182	12294	10459	8456
Multithread	11938	9118	8148	6826
Percentual	21,4%	25,9%	22,1%	19,3%
Aceleração	1,27	1,34	1,28	1,23

Tabela 7.10: Tempos de execução em *ms* e Aceleração para o *kernel* Radix para um conjunto de dados de 16384 chaves.

32 *bits* cada uma.

Aceleração	Paralela	*Multithread*	Eficiência	Paralela	*Multithread*
$S_{f_4} = T_2/T_4$	1,23	1,30	E_{f_4}	0,30	0,32
$S_{f_8} = T_2/T_8$	1,45	1,46	E_{f_8}	0,18	0,18
$S_{f_{16}} = T_2/T_{16}$	1,79	1,74	$E_{f_{16}}$	0,11	0,10

Tabela 7.11: Aceleração e Eficiência obtidas para o conjunto de dados de 16384 em relação a 2 processadores.

Aceleração	Paralela	*Multithread*	Eficiência	Paralela	*Multithread*
$S_{v_4} = T_2/T_4$	1,30	1,23	E_{v_4}	0,32	0,30
$S_{v_8} = T_4/T_8$	1,12	1,17	E_{v_8}	0,13	0,14
$S_{v_{16}} = T_8/T_{16}$	1,19	1,23	$E_{v_{16}}$	0,07	0,07

Tabela 7.12: Aceleração e Eficiência obtidas para o conjunto de dados de 16384 chaves.

Na tabela 7.11 é mostrada a aceleração e a eficiência obtida no *kernel* Radix. A aceleração foi calculada em relação a 2 processadores, e a eficiência foi denominada E_f pois também foi calculada em relação a 2 processadores. Não foram realizados testes com um único processador (versão serial) pois o programa utilizado era uma versão paralela, e a versão serial não estava disponível. Na tabela 7.12 é mostrada a aceleração e a eficiência para o *kernel* Radix, mas nesta tabela a aceleração e a eficiência foram calculadas com o número de processadores variável, e a eficiência foi denominada E_v.

No gráfico 7.7 e na Tabela 7.13 é mostrado o tempo de execução para o *kernel* Radix, e neste experimento tanto o número de processadores quanto o conjunto de dados foi variado. O conjunto de dados utilizado foi de 8192 chaves para 2 processadores, 16384 chaves para 4 processadores, 32768 chaves para 8 processadores e 65536 chaves para 16 processadores. Os ganhos de desempenho são de 25,9%, 19,4%, 14,8% e 12,0% para 2,4,8 e 16 processadores respectivamente.

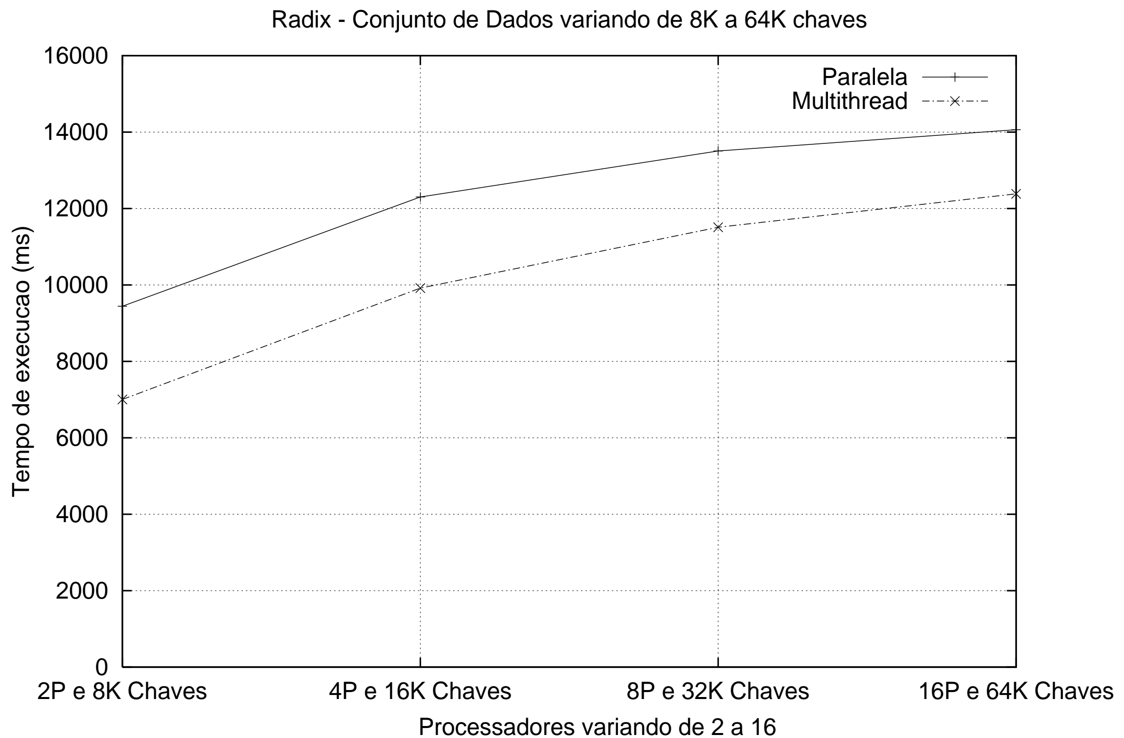


Figura 7.7: Resultados obtidos no *kernel* Radix com variação no número de processadores e no tamanho do conjunto de dados.

Versão	$P = 2$	$P = 4$	$P = 8$	$P = 16$
Paralela	9445	12300	13506	14062
Multithread	7002	9918	11511	12384
Percentual	25,9%	19,4%	14,8%	12,0%

Tabela 7.13: Tempos de execução em *ms* para o *kernel* Radix com variação no número de processadores e no tamanho do conjunto de dados.

Apesar deste *kernel* apresentar granularidade pequena, a abordagem de divisão da comunicação em duas *threads* distintas – uma responsável pela comunicação que chegava ao processador e a outra responsável pela comunicação que saía do processador – foi acertada como comprova o ganho de desempenho mostrado nos gráficos das figuras 7.6 e 7.7.

Na Tabela 7.11 pode-se observar uma diminuição na eficiência do programa quando utiliza-se número de processadores igual ou superior a 8 processadores em relação à eficiência para 4 processadores. Isto pode-se observar tanto para a versão *multithread* quanto para a versão paralela original. Isto deve-se ao fato de que no *kernel* radix não se utiliza as primitivas de comunicação coletiva do MPI nas fases em que há comunicação neste

algoritmo. No *kernel* radix a comunicação dá-se através das primitivas de comunicação ponto-a-ponto do MPI. No trabalho de [Hél03] é mostrado que o tempo despendido nas primitivas de comunicação coletiva do MPI apresenta crescimento logarítmico a medida em que o número de processadores cresce, enquanto que o tempo despendido nas primitivas de comunicação ponto-a-ponto do MPI apresenta crescimento linear a medida em que o número de processadores cresce. O desempenho deste *kernel* pode ser melhorado caso utilize-se as primitivas de comunicação coletiva do MPI, principalmente na fase em que há a troca de chaves entre os processadores. Esta alteração não foi realizada pois não faz parte do escopo desta dissertação.

As duas versões do *kernel* Radix analisadas nesta dissertação empregam tamanho máximo das mensagens de 1500 *bytes*, [1] que é o tamanho máximo de um pacote em rede *Ethernet*.

Após a realização dos experimentos nos *kernels* FFT, *LU* e Radix, pode-se verificar que a abordagem da utilização de *threads* em conjunto com o padrão MPI pode diminuir o tempo de execução dos programas paralelos. Após o cálculo das médias harmônicas dos valores obtidos nos experimentos, os ganhos foram de 22% no *kernel* Radix, e 16% no *kernel* FFT. Já o comportamento observado no *kernel* *LU* indica que esta abordagem nem sempre compensa o esforço adicional de programação.

[1] Incluindo os cabeçalhos IP, TCP e MPI.

CAPÍTULO 8

CONCLUSÃO

Neste trabalho foram desenvolvidas três versões paralelas dos *kernels Fast Fourier Transform* (FFT), fatorização *LU* e ordenação Radix. Estas novas versões utilizavam a biblioteca para troca de mensagens MPI em conjunto com o pacote de *threads* POSIX.

Conforme mostrado na literatura [KS96], ao encontrar uma operação que bloqueie uma determinada *thread*, uma troca de contexto ocorrerá passando o processamento da *thread* bloqueada para uma outra *thread* que poderá seguir com o seu processamento. O objetivo inicial desta dissertação era explorar a capacidade das *threads* separando o processamento de um programa em duas *threads* distintas, uma responsável pelos cálculos e a outra responsável pela comunicação entre os processadores, desta maneira sobrepondo-se o tempo gasto na comunicação com processamento útil, e assim diminuir o tempo total de execução dos programas paralelos.

Após o desenvolvimento das versões *multithread* dos programas, os tempos de execução dos programas foram comparados com os tempos de execução dos mesmos programas paralelos que também utilizavam a biblioteca MPI mas que não utilizavam *threads*.

Foram realizados testes com os programas variando-se o número de processadores de 2 até 16 processadores, sendo que o tamanho do conjunto de dados foi mantido fixo em um experimento e em outro experimento o tamanho do conjunto de dados foi variado à medida em que o número de processadores aumentava.

Os resultados mostram ganho de desempenho para o *kernel* FFT ao empregar-se *threads* em conjunto com a biblioteca MPI. O desempenho da versão *multithread* do *kernel* Radix também foi melhor que a versão paralela original. Com o *kernel LU* houve perda de desempenho na versão *multithread* em relação à versão paralela original.

No *kernel* FFT existe comunicação apenas nas fases de transposição da matriz que contém o conjunto de dados. Nesta fase foi explorada a sobreposição da comunicação existente na transposição das linhas da matriz com o cálculo realizado nas linhas que foram transpostas da matriz. Os resultados obtidos mostram que a versão *multithread* do *kernel* FFT é em média 16% mais rápida que a versão paralela original.

No *kernel* Radix foi utilizado uma abordagem diferente de divisão das *threads*. Neste *kernel* foram criadas três *threads*, uma responsável pela criação do histograma local, criação do histograma global e cálculo da distribuição das chaves na terceira fase do algoritmo, e as outras duas *threads* foram destinadas uma ao envio das chaves e a outra pela recepção das chaves enviadas por outros processadores. Desta maneira a comunicação envolvida no envio das mensagens era sobreposta com a comunicação envolvida no recebimento das mensagens. Os resultados obtidos mostram que a versão *multithread* do *kernel* Radix é em média 22% mais rápida que a versão paralela original.

No *kernel LU* buscou-se sobrepor a comunicação existente na fase em que os blocos de perímetro são atualizados, mas o *overhead* inserido pela criação das *threads* e a concorrência entre as *threads* em fases quando não existe comunicação entre os processadores causam perda de desempenho na versão *multithread* em relação à versão paralela original. Os resultados obtidos mostram que na versão *multithread* do *kernel LU* houve um decréscimo da ordem de 23,5% em média no tempo de execução da versão *multithread* em relação à versão paralela original.

Com base nos resultados dos experimentos realizados com os programas de testes, observou-se que a utilização da programação *multithread* em conjunto com o padrão MPI pode trazer redução significativa no tempo de execução de programas paralelos. A utilização de *threads* em conjunto com MPI traz um aumento na complexidade de desenvolvimento dos programas, além da complexidade inerente envolvida no desenvolvimento de um programa paralelo. Verificou-se também que os ganhos de desempenho obtidos com a utilização de *threads* em conjunto com MPI nem sempre são garantidos.

8.1 Trabalhos Futuros

Outros trabalhos podem ser realizados de maneira a complementar e estender o estudo aqui descrito. Alguns destes são mencionados a seguir.

A avaliação do desempenho de programas paralelos utilizando MPI em conjunto com outras bibliotecas de *threads* como OpenMP, *threads* do sistema operacional Linux, bem como outros pacotes de *threads* existentes, pode complementar o estudo aqui realizado, pois outros pacotes de *threads* apresentam diferentes características de controle das *threads* que pode culminar em resultados diferentes dos obtidos neste estudo.

Como no *kernel LU* na implementação *multithread* a sobreposição da comunicação com computação só foi possível na segunda fase do algoritmo, esta sobreposição também pode ser estendida para a terceira fase do algoritmo, desde que sejam realizadas alterações de modo a iniciar individualmente o cálculo dos blocos interiores que já possuam os blocos de perímetro necessários para este cálculo, assim os blocos interiores não necessitam esperar por todos os blocos de perímetro como acontece nas versões paralela e *multithread*. Espera-se com isto que o tempo de execução da versão *multithread* aproxime ou supere o tempo da versão paralela original.

A comunicação global apresenta um comportamento diferente do apresentado pela comunicação ponto-a-ponto, pois a comunicação global utiliza algoritmos baseados em árvores para a disseminação das mensagens [Hél03]. A verificação da sobreposição da comunicação com a computação em programas que utilizam comunicação global mais freqüentemente complementaria o estudo aqui realizado.

As *threads* permitem a utilização de diferentes políticas de escalonamento. Esta característica das *threads* não foi verificada nesta dissertação por limitações no pacote de *threads* utilizado. A utilização de diferentes políticas de escalonamento pode levar a resultados diferentes dos obtidos nesta dissertação, por isto uma avaliação do desempenho dos programas de testes verificado com a utilização de diferentes políticas de escalonamento pode complementar o estudo aqui descrito.

BIBLIOGRAFIA

[AJK+91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. *Proceedings of the 18 International Symposium on Computer Architecture*, pages 254–263, May 1991.

[Ble91] Blelloch G. E. et. al. A comparison of sorting algorithms for the connection machine CM-2. *In Symposium on Parallel Algorithms and Architectures*, 1:3–16, 1991.

[Dan93] Dan I. Moldovan. *Parallel Processing from Applications to Systems*, volume 1. Morgan Kauffman Publishers, first edition, 1993.

[Dav90] David H. Bailey. FFTs in external or hierarchical memory. *Journal of Supercomputing*, 4:23–25, 1990.

[EC93] Edil S. T. Fernandes and Claudio L. de Amorim. *Arquiteturas Paralelas Avançadas.* VI Escuela Brasileño - Argentina de Informática, 1993.

[For95] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *Technical Report. University of Tennessee*, 1.1, June 1995.

[For97] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. *Technical Report. University of Tennessee*, July 1997.

[Gre95] Gregory F. Pfister. *In Search of Clusters.* Prentice Hall PTR, 1995.

[GV93] G. A. Geist and V.S. Sunderam. The evolution of the PVM concurrent computing system. *Preceedings of the 26th IEEE Compcon Symposium*, 1:471–478, 1993.

[Hél03] Hélio Marci de Oliveira. *Modelagem e Predição de Desempenho de Primitivas de Comunicação MPI.* Thesis (M.Sc.), Universidade de São Paulo, São Paulo, 2003.

[JD96] John L. Hennessy and David A. Patterson. *Computer Architecture: A quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, second edition, 1996.

[JD97] John L. Hennessy and David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, third edition, 1997.

[JD98] JunSeong Kim and David J. Lilja. Characterization of communication patterns in message-passing parallel scientific applications programs. *In Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing*, 1:202–216, 1998.

[JM] João Paulo F. W. Kitajima and Marco Aurélio de Souza Mendes. Introdução aos processos leves (Threads). `ftp://ftp.dcc.ufmg.br/pub/ftp/research/kitajima/jai96.ps`, acessado em 08/09/2004.

[JSD96] Jack J. Dongarra, Steve W. Otto, and David Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 37(7):84–90, July 1996.

[KS96] Kay A. Robbins and Steven Robbins. *Practical Unix programming (A guide to concurrency, communication, and MultiThreading)*. Prentice Hall, 1996.

[M. 66] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54:1901–1909, 1966.

[Mat97] Matthew Flint Arnett. *Desvendando o TCP/IP*. Campus, 1997.

[Mic87] Michael J. Quinn. *Designing efficient algorithms for parallel computers*, volume 1. McGraww-Hill, first edition, 1987.

[MSS+] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. MPI: The Complete Reference. `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`, acessado em 01/08/2004.

[PAR94] PARKBENCH Committee/Assembled by R. Hockney (Chairman) and M. Berry (Secretary). PARKBENCH report: Public international benchmarks for parallel computers. *j-SCI-PROG*, 3(2):101–146, 1994.

[Pet97] Peter S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997.

[RADT97] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. *Comunications of the ACM*, pages 85–97, Feb 1997.

[SJJ94] S. C. Woo, J. P. Singh, and J. L. Hennessy. The performance advantages of integrating message passing in cache-coherent multiprocessors. *ACM SIGPLAN Notices*, 29:219–229, 1994.

[SME+95] S. C. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Jun. 1995.

[Ste96] Steven Cameron Woo. *The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors.* Thesis (Ph.D.), Stanford University, `ftp://www-flash.stanford.edu/pub/flash`, 1996.

[TCW92] Thomas M. Warschko, Christian G. Herter, and Walter F. Tichy. Latency hiding in parallel systems: A quantitative approach. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 51–61, 1992.