Universidade Federal do Paraná

Departamento de Informática

Sérgio Luiz Marques Filho & Roberto André Hexsel

Uso de Threads para Esconder Latência em Aplicações Paralelas com MPI

Relatório Técnico
RT-DINF 001/2005

Curitiba, PR
2005

Uso de Threads para Esconder Latência em Aplicações Paralelas com MPI

Sérgio Luiz Marques Filho & Roberto A Hexsel

Departamento de Informática
Universidade Federal do Paraná
Centro Politécnico, C Postal 19081
81531-990 Curitiba, PR
sergio,roberto@inf.ufpr.br

Abstract This paper presents a performance evaluation of parallel programs that employ threads to hide communication latency in an Ethernet based cluster. In the experiments were used three kernels, namely Fast Fourier Transform (FFT), LU Factorization and Radix Sorting. New multi-threaded versions were developed in order to gauge the performance gains with respect to the "pure" MPI versions. The results for clusters with 2, 4, 8 and 16 processors show a performance gain of 15-17% and 18-22%, for FFT and Radix kernels, respectively. The results for LU show a performance loss between 60-80% with respect to the original version, in the 16 node cluster. Whatever the gains, these must be balanced against the increase in complexity of the code.

Resumo Neste trabalho avalia-se a utilização de *threads* em conjunto com a troca de mensagens do padrão MPI para esconder a latência da comunicação da rede Ethernet em um aglomerado. Foram utilizados três *kernels*, Fast Fourier Transform (FFT), Fatorização LU e Ordenação Radix, dos quais foram desenvolvidas novas versões que utilizam o padrão MPI em conjunto com a programação *multithread*. Os resultados da avaliação em aglomerados com 2, 4, 8 e 16 processadores mostram ganho de desempenho para os *kernels* FFT e Radix, da ordem de 15-17% e 18-22%, respectivamente. Com o *kernel LU* houve perda de desempenho na versão modificada com relação à versão paralela original, da ordem de 60-80%. O eventual ganho em desempenho deve ser considerado à luz da maior complexidade do código.

Introdução

Durante o tempo de uso de uma certa tecnologia de redes locais, algo como 5-7 anos para cada geração da Ethernet (10, 100 Mbps, 1 ou 10 Gbps) por exemplo, estarão disponíveis no mercado processadores com capacidade de processamento numa faixa da ordem de 1:10 e 1:24. Isso ocorre porque a capacidade de processamento cresce a uma taxa de aproximadamente 1,58% ao ano, e em períodos de 5 e 7 anos a capacidade cresce 10 e 24 vezes, respectivamente [1]. Quando uma nova geração de tecnologia de rede é disponibilizada no ano A, os processadores daquele ano mal tem capacidade para ocupar uma fração da banda disponível. Na véspera da introdução na nova geração, os processadores do ano A+5 terão capacidade para saturar a rede com alguma folga.

Ao longo do período útil de uma geração tecnológica, certas técnicas de implementação de programas paralelos tornam-se viáveis por causa do aumento na capacidade dos

processadores. Assim, o uso da programação por troca de mensagens torna-se mais atrativo porque o custo adicional da comunicação, imposto pelo software, pode ser amortizado com uma nova geração de processadores. A utilização de processos leves, ou *threads*, permite esconder parte da latência na comunicação se a aplicação é codificada com *threads* distintos para a computação e para a comunicação.

Este artigo apresenta resultados experimentais para medir os ganhos de desempenho que podem ser obtidos da utilização de *threads* em conjunto com a troca de mensagens em aglomerados de até 16 nós. Para a avaliação foram empregados três programas de testes, ou *kernels*, que são o Fast Fourier Transform (FFT), Fatorização LU e Ordenação Radix, programados para execução paralela com as primitivas de comunicação da *Message Passing Interface* (MPI). Os experimentos comparam uma versão paralela dos programas com uma versão modificada pelos autores com *threads* distintos para computação e comunicação.

O texto está organizado da seguinte forma. A Seção 1 discute alguns trabalhos relacionados ao que é descrito aqui. A Seção 2 apresenta a metodologia e o ambiente da avaliação. As Seções 3, 4, e 5 descrevem os programas de teste e os resultados dos experimentos. Nossas conclusões estão na Seção 6.

1 Trabalhos Relacionados

O trabalho descrito em [2] contém uma modelagem e predição de desempenho das primitivas da biblioteca MPI. São modeladas as primitivas da biblioteca MPI e a predição de desempenho é validada pela medição dos tempos de comunicação destas primitivas em programas simples. 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 ponto-a-ponto cresce linearmente com o tamanho das mensagens, enquanto que o tempo de comunicação para as primitivas de comunicação coletiva cresce linearmente com o tamanho das mensagens e logaritmicamente com o número de processadores.

O conjunto de aplicativos SPLASH-2 é descrito em [3], com a caracterização quantitativa dos programas com relação às interações com a arquitetura dos computadores, em especial: (i) balanceamento de carga, (ii) razão entre a comunicação e a computação, (iii) o tráfego, (iv) localidade espacial, e (v) como essas propriedades são escaláveis quando o tamanho do problema, e/ou número de processadores, aumentam.

Os resultados em [4, 5] indicam que quanto menor o tamanho das mensagens (blocos), a vantagem da utilização das transferências 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 [4, 5] também mostram que a utilização de memórias cache primárias de tamanho maior podem diminuir a vantagem da transferência em blocos em multiprocessadores com cache coerentes.

O artigo [6] 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, bem como os efeitos da variação do tamanho do problema. A conclusão é de que a freqüência da comunicação tende a crescer na 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.

Dois artigos analisam métodos para sobreposição da latência no acesso à memória. Em [7] são verificados quatro métodos para esconder a latência no acesso à memória:

(i) memória com cache coerente, (ii) modelos relaxados de consistência de memória, (iii) busca antecipada controlada por software, e (iv) múltiplos contextos ou *threads*. Em [8] são abordados os métodos de busca antecipada e em menor escala múltiplos contextos. Os dois trabalhos mostram 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. Através de combinações destas técnicas pode-se obter um melhor desempenho do que com cada uma das técnicas individualmente. Utilizando combinações destas técnicas pode-se obter ganhos da ordem de 4 até 7 vezes. Quanto ao *kernel* LU, [7] reporta que quando o custo da troca de contexto é elevado, o desempenho do *kernel* é pior do que a versão sem trocas de contexto. Um comportamento similar foi observado nos experimentos descritos adiante.

2 Metodologia e Programas de Testes

Um *kernel* de teste é uma pequena parte de um programa real, geralmente a parte com a computação mais intensiva, escolhida com o intuito de avaliar o desempenho de uma determinada máquina ou subsistema. Foram escolhidos três programas de testes, *Fast Fourier Transform* (FFT), Fatorização LU e Ordenação Radix. Estes *kernels* apresentam granularidades (taxas de comunicação para a computação) bastante distintas [3, 9], como mostra a Tabela 1, onde P representa o número de processadores e N representa o tamanho do conjunto de dados.

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

Tabela 1: Taxa de Comunicação para Computação dos programas de teste.

Nos experimentos descritos adiante foram utilizadas duas versões de cada um dos *kernels* para avaliar os eventuais ganhos de desempenho decorrentes do uso de paralelismo intra- e inter-nós. A primeira é a *versão paralela original*, na qual dá-se prioridade à comunicação não-bloqueante na tentativa de sobrepor comunicação com computação, mas sem a utilização de múltiplas *threads*. A segunda é a versão dos mesmos programas com *múltiplas threads*. Ambas as versões utilizam a biblioteca MPI para comunicação entre os processadores.

2.1 Metodologia de Medição de Desempenho

Para cada experimento foram realizados pelo menos 20 execuções dos programas e então é calculada a média dos tempos de execução. Dentre os tempos medidos, os maiores valores são descartados, pois os menores tempos geralmente são os mais precisos porque sofrem menor influência do sistema operacional e de outras aplicações [2].

Dentro dos limites impostos pelos computadores disponíveis, essencialmente a capacidade de memória de 128 Mbytes, buscou-se maximizar o tamanho dos conjuntos de dados para que estes fossem próximos de valores representativos das condições de uso dos

programas, o que aumenta a granularidade e facilita a exploração das propriedades da programação com múltiplas *threads*.
Ambiente de Testes Os nodos utilizados para a realização dos experimentos são 16 nós de processamento com um processador AMD Duron de 1.2 GHz, 128 Mbytes de memória RAM, 128 Kbytes de memória cache L1 e 64 Kbytes de memória cache L2, rede de interconexão Fast-Ethernet, sistema operacional GNU/Linux Kurumin 4.0 com kernel 2.4.25, pacote de *threads* Pthreads [10], e a implementação MPICH 1.2 do padrão MPI.
Sincronização das threads Como o bloqueio de uma *thread* não causa o bloqueio de uma outra *thread*, para toda barreira da biblioteca MPI existente nos programas originais com correspondente na $thread_{\mathrm{i}}$, foi necessário incluir um evento de sincronização na $thread_{\mathrm{j,k}}$ para que esta(s) não continuasse(m) o processamento enquanto a $thread_{\mathrm{i}}$ inicial permanecia bloqueada na barreira MPI do código original.

3 Fast Fourier Transform – FFT

O *kernel FFT* é uma versão complexa e unidimensional do algoritmo *Fast Fourier Transform* descrito em [11], e é o componente principal de algumas aplicações tais como processamento de sinais, reconhecimento de voz, processamento de imagens, análise sísmica de petróleo, e dinâmica dos fluidos.

O conjunto de dados para o algoritmo consiste de (i) uma *matriz de entrada* de n pontos complexos que serão transformados (n é uma potência de 2), (ii) outros n pontos complexos que são chamados de *raízes da unidade* que representam as n raízes complexas do número real 1, e (iii) uma *estrutura intermediária* de dados que contém n pontos complexos e é utilizada para manter valores intermediários calculados pelo algoritmo. As estruturas são organizadas como matrizes de $\sqrt{n} \times \sqrt{n}$ elementos, e a matriz com as raízes da unidade não é atualizada após a inicialização do programa. O conjunto de dados é distribuído entre os processadores em blocos contíguos de linhas das matrizes, como mostra a Figura 1.

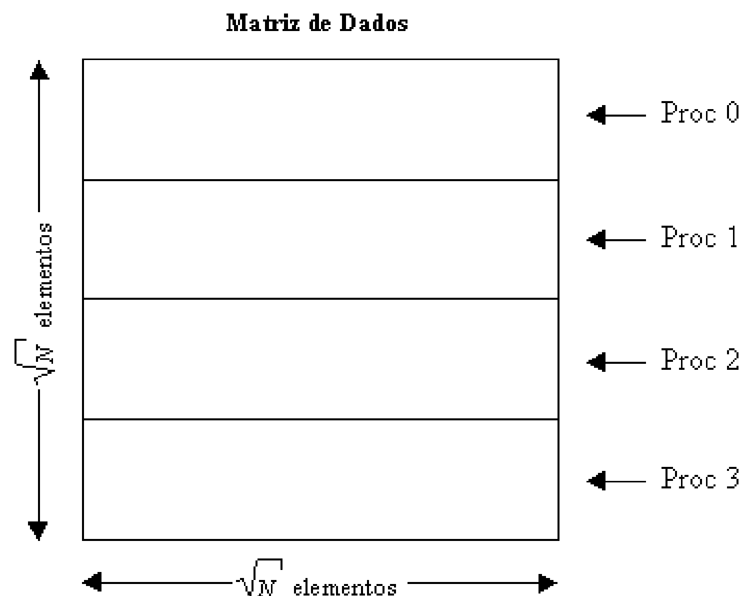

Figura 1: Distribuição do conjunto de dados no kernel FFT.

O algoritmo da FFT consiste de seis passos: (1) a matriz de entrada é transposta na estrutura intermediária, (2) são realizadas FFTs unidimensionais em cada linha da matriz intermediária, e os resultados são armazenados novamente na matriz intermediária, (3) os elementos correspondentes da matriz raízes da unidade são aplicados a cada elemento da matriz intermediária, (4) a matriz intermediária é transposta na matriz de entrada, (5) são realizadas FFTs unidimensionais em cada linha da matriz de entrada com os resultados sendo armazenados na própria matriz de entrada, e finalmente (6) a matriz de entrada é transposta na matriz intermediária.

A comunicação entre os processadores ocorre apenas durante as fases em que há transposição das matrizes. O bloco de dados que pertence ao próprio processador é sempre transposto localmente, enquanto que os outros $P - 1$ blocos são transmitidos aos 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 2. Durante o terceiro passo do algoritmo, os elementos da matriz raízes da unidade são multiplicados pelos seus pares na matriz de entrada, e durante o segundo e o quinto passos do algoritmo, as FFTs unidimensionais também referenciam a matriz raízes da unidade.

Os passos 2 e 3 são agrupados porque ambos fazem referência à matriz raízes da unidade, aproveitando-se a localidade de referência aos dados que estão na memória cache do processador [11]. Ainda, nos passos 3 e 5, a multiplicação dos elementos da matriz ocorre sobre a topologia de uma “borboleta” organizada para aproveitar a localidade espacial dos dados na memória cache [5].

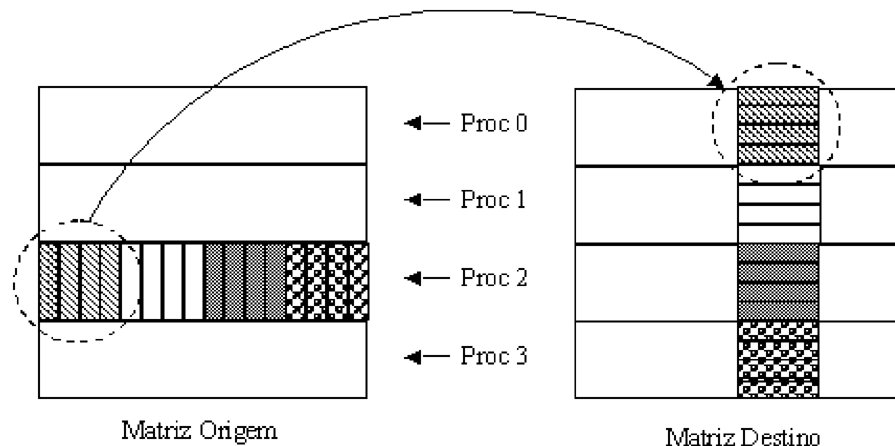

Figura 2: Fase de transposição da matriz.

3.1 Versão com threads do kernel FFT

O *kernel FFT* original utiliza o algoritmo denominado *Six-Step FFT*, que requer três transposições de matrizes nas fases 1, 4 e 6. O algoritmo requer duas aplicações de FFTs unidimensionais no conjunto de dados (fases 2 e 5). Na fase 3, é realizada uma aplicação da matriz raízes da unidade na matriz de dados. Na versão original, a fase de transposição da matriz é completada para só então iniciarem as outras fases do algoritmo.

Como exposto anteriormente, as fases 2 e 3 do algoritmo foram combinadas numa única fase de computação, e a fase 5 também é uma fase computacional, enquanto que as fases 1, 4 e 6 exigem comunicação. O desempenho do programa pode ser melhorado caso o

programa seja reestruturado de maneira a sobrepor comunicação com computação entre as fases do algoritmo. Um processador computa as FFTs unidimensionais e aplica a matriz raízes da unidade a todas as linhas da matriz que estão em sua posse. Para sobrepor a comunicação com computação, as linhas que estão de posse de um processador podem ser transpostas individualmente (comunicação). Após a transposição de uma linha, realiza-se FFTs unidimensionais e aplica-se a matriz raízes da unidade naquela linha, podendo-se transpor outra linha da matriz enquanto são computadas as FFTs unidimensionais e a aplicação da matriz raízes da unidade. A Figura 3 mostra a transposição individual das linhas da matriz.

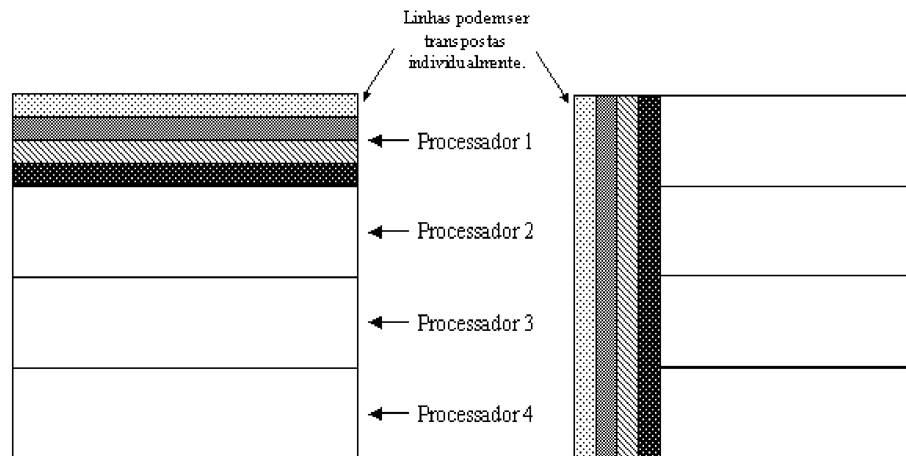

Figura 3: Linhas sendo transpostas individualmente.

O *kernel* original foi modificado para que a transposição da matriz, e a computação nas FFTs unidimensionais e aplicação das raízes da unidade, sejam realizadas linha a linha. A comunicação é não-bloqueante e existem pontos de sincronização no código para garantir a coerência dos dados, e não é possível sobrepor comunicação com computação na transposição final.

Para a implementação com *threads* do *kernel FFT*, o processamento foi dividido em duas funções. A primeira função é denominada `FFT1D_thread` e é a *thread* de computação com a tarefa de realizar os cálculos propriamente ditos. A segunda função é denominada `TRANSPOSE_thread` e é a *thread* de comunicação, que controla as trocas de mensagens entre os processadores. Assim, a função `FFT1D_thread` implementa as fases 2, 3 e 5 do algoritmo, enquanto que a função `TRANSPOSE_thread` implementa as fases 1, 4 e 6. Durante a transposição da matriz, blocos de elementos de tamanho $\sqrt{N}/P$ são enviados a $P-1$ processadores, e um bloco de tamanho $\sqrt{N}/P$ é transposto localmente.

3.2 Resultados para FFT

O *kernel FFT* apresentou ganho de desempenho ao utilizar-se *threads* em conjunto com a biblioteca MPI. Os gráficos na Figura 4 mostram no eixo vertical o tempo de execução do programa, e no eixo horizontal o número de processadores (2, 4, 8 e 16). Neste experimento (gráfico da esquerda) o conjunto de dados foi mantido fixo em 1.048.576 (2^{20}) números complexos de dupla precisão. Os ganhos no desempenho são de 20,6%, 20,4%, 9,5% e 19% para 2, 4, 8 e 16 processadores respectivamente, com média de 17,5%. A Tabela 2 mostra os ganhos de desempenho da versão com threads com relação à versão original. A média harmônica das taxas é mostrada na coluna da direita.

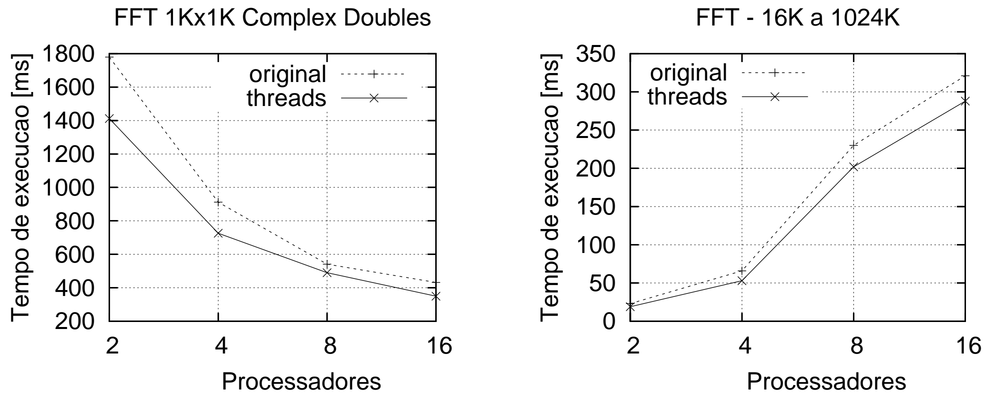

Figura 4: Resultados do kernel FFT, conjunto de dados fixo (esq) e variável (dir).

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 unidimensionais a medida em que as linhas eram transpostas.

	Conjunto de dados fixo (2^{20} elementos)				
proc	2	4	8	16	méd
$T_{\text{thr}}/T_{\text{par}}$	0,794	0,796	0,906	0,810	0,825
	Conjunto de dados variável				
|conj dados|	16K	64K	256K	1024K	
$T_{\text{thr}}/T_{\text{par}}$	0,826	0,803	0,878	0,897	0,850

Tabela 2: Ganhos no FFT.

O gráfico da direita na Figura 4 também mostra o tempo de execução quanto o tamanho do conjunto de dados é escalado com o número de processadores. 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 é quadruplicada quando o número de processadores dobra. Assim, foram realizados testes com 16.384 números complexos com 2, 65.536 números complexos em 4, 262.144 números complexos em 8, e 1.048.576 números complexos em 16 processadores. Os ganhos de desempenho são de 17,4%, 19,7%, 12,2% e 10,3% para 2, 4, 8 e 16 processadores respectivamente, e média harmônica de 15%. O *kernel FFT* apresenta um aumento significativo no tempo de execução para o conjunto de dados variável. Isso decorre da diferença nas taxas de crescimento do conjunto de dados $[2 \cdot P \rightarrow 4 \cdot N]$ e da granularidade $[(P-1)/(P \log N)]$.

4 Fatorização LU

O *kernel LU* efetua a fatorização LU de uma matriz densa, e é representativo de muitos problemas de álgebra linear como fatorização QR e fatorização Cholesky. O algoritmo e sua implementação original estão descritos em [5, 1]. O programa fatora uma matriz densa A no produto de uma matriz triangular inferior L (*lower*) e uma matriz triangular superior U (*upper*). Para uma matriz de tamanho $n \times n$, o tempo de execução é $\mathcal{O}(n^3)$, e paralelismo é proporcional a n^2.

A matriz A de tamanho $n \times n$ a ser fatorada é decomposta em blocos de tamanho $B \times B$, com computação e comunicação efetuadas sobre estes blocos. Os blocos a serem 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: (1) na iteração i o processador que possui o bloco diagonal $A[i, i]$ faz a fatoração do bloco diagonal; (2) 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; e (3) 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 envolve o maior número de elementos a serem atualizados.

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 existe comunicação quando os processadores que possuem os blocos de perímetro fazem uso do valor atualizado do bloco diagonal para computarem os blocos de perímetro. No terceiro passo ocorre 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.

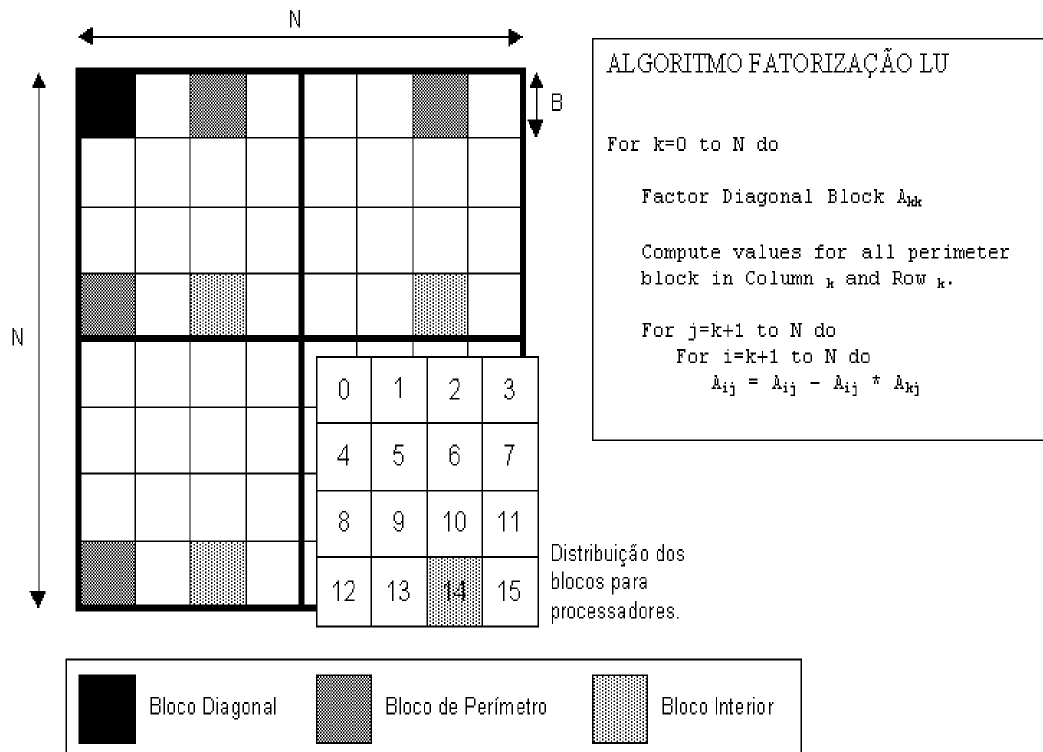

Figura 5: Distribuição do conjunto de dados para os processadores do kernel LU.

O *kernel LU* utiliza a comunicação *iniciada pelo transmissor:* após um determinado bloco ser fatorado, o processador transmite-o 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 efetua a fatoração do bloco diagonal no primeiro passo de cada iteração do algoritmo, e o transmite para cada processador que seja proprietário de blocos de perímetro. Cada processador que detém blocos de perímetro aguardará pela finalização da transmissão do bloco 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, cada processador proprietário de um daqueles blocos iniciará uma nova transmissão 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, este deve aguardar pela finalização da transmissão de dois outros blocos, um é proveniente do processador proprietário do bloco de perímetro na mesma linha do bloco interior, e outro 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, este não necessita aguardar pela transmissão pois o bloco de perímetro pode ser acessado diretamente da memória local.

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 na medida em que novos processadores são acrescentados para a resolução do problema.

O conjunto de dados do *kernel LU* é uma matriz de tamanho $n \times n$. O caminho natural para codificar este programa é a utilização de uma matriz bidimensional, mas com uma linguagem orientada a linhas, como a linguagem C na qual foram desenvolvidos os programas de teste, os elementos de um bloco não são armazenados em posições consecutivas de memória. Este fato traz perda de desempenho porque a unidade de transferência do *kernel* é um bloco. Assim, os elementos do bloco deveriam ser armazenados em posições contíguas de memória antes da transmissão, aumentando seu custo. Para contornar este problema, o conjunto de dados do *kernel* é armazenado 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 6.

4.1 Versão com threads da Fatoração 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. Só é possível aproveitar a sobreposição da comunicação com computação na fase em que os blocos de perímetro são calculados, porque não há comunicação durante o cálculo do bloco diagonal, e a fase de atualização dos blocos de perímetro só pode ser iniciada após a chegada do bloco diagonal. Ainda, a fase de atualização dos blocos interiores só tem início após a chegada de todos os blocos de perímetro.

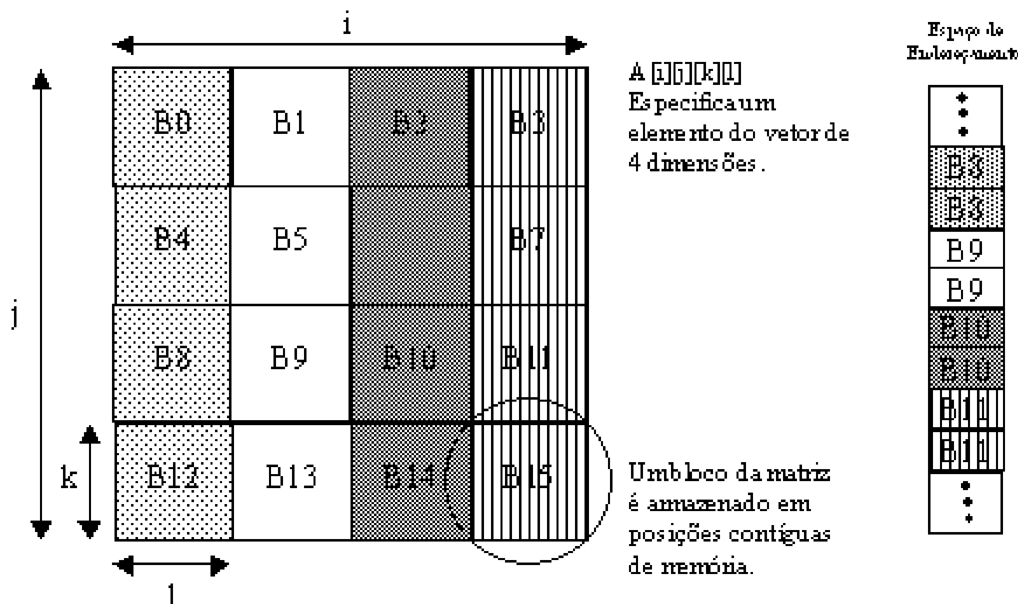

Figura 6: Distribuição dos blocos usando uma matriz de quatro dimensões.

Este *kernel* tem três pontos de comunicação. O primeiro ponto de comunicação está no final da função `lu0`. Assim que o bloco diagonal é fatorado, este é enviado aos outros processadores que estão bloqueados enquanto aguardam pelo bloco diagonal. O segundo e o terceiro pontos de comunicação estão no final das funções `bdiv` e `bmodd`, que computam os blocos de perímetro e os transmitem aos outros processadores, para os proprietários de blocos interiores possam fatorá-los. Os processadores não ficam bloqueados esperando pelos blocos de perímetro. Não há outros pontos de comunicação após a atualização dos blocos interiores, mas existe uma barreira antes da atualização dos 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 é a fase que demanda maior tempo de processamento neste *kernel*.

Na versão com *threads* do *kernel LU*, o processamento é similar ao do *kernel FFT*: o trabalho é dividido entre duas funções que realizam tarefas de comunicação e computação, chamadas de `lu_comm` e `lu_compute`, respectivamente. Não é necessário adicionar primitivas de sincronização explícitas para controlar o início do processamento pelas *threads* porque os eventos bloqueantes de comunicação são suficientes para garantir a coerência dos resultados.

Segundo Woo *et al* [5] o desempenho do *kernel LU* é melhor com blocos de 8×8 ou 16×16 elementos porque blocos com estes tamanhos cabem na cache primária dos processadores. Tal não é o caso na versão com *threads* pois um bloco de 64 elementos (8×8) é processado muito rapidamente, antes que finalize o *quantum* da *thread*. Assim, a *thread* de comunicação não chega a competir pelo processador com a *thread* de computação, e portanto não há sobreposição da comunicação com computação. É necessário que as *threads* liberem o processador (`pthread_yield`) após o processamento de um bloco. Para blocos maiores (64×64) isso não ocorre porque os blocos são suficientemente grandes para existir competição entre as *threads*.

No código existem vários laços que calculam os endereços de memória dos blocos da matriz principal para serem fatorados pelas funções `lu0`, `bdiv`, `bmodd` e `bmod`. Estes laços foram duplicados nas *threads* de computação e de comunicação, o que aumenta o número de instruções executadas pelos processadores. O impacto da duplicação será discutido adiante.

4.2 Resultados para 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, e os dados na Tabela 3. Com conjunto de dados de tamanho fixo ($2K \times 2K$) e 4, 8 e 16 processadores as perdas no desempenho são de 11%, 61% e 85% respectivamente. Neste caso, a variação dos resultados é tão grande que a média não é um bom indicador de desempenho.

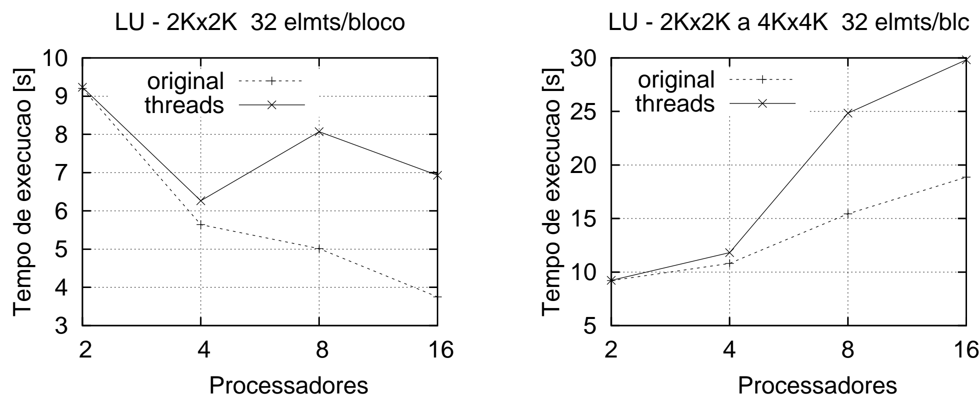

Figura 7: Resultados do LU para um conjunto de dados fixo (esq) e variável (dir).

No gráfico da direita na Figura 7 é mostrado o tempo de execução para o *kernel LU* quando o conjunto de dados é escalado com o número de processadores. Os conjuntos de dados utilizados foram de 2.048 elementos para 2, 2.560 elementos para 4, 3.232 elementos para 8, e 4.096 elementos para 16 processadores. Os tamanhos foram escolhidos porque o *kernel LU* apresenta crescimento do tempo de execução proporcional a $\mathcal{O}(n^3)$. Os tamanhos dos conjuntos de dados foram arredondados para permitir a organização em blocos de 32×32 elementos. Com conjunto de dados variável e 4, 8 e 16 processadores as perdas no desempenho são de 10%, 61% e 58% respectivamente.

	Conjunto de dados fixo (2048×2048)				
proc	2	4	8	16	méd
$T_{\text{thr}}/T_{\text{par}}$	1,003	1,111	1,609	1,848	1,349
	Conjunto de dados variável				
$\sqrt{\text{conj dados}}$	2048	2560	3232	4096	
$T_{\text{thr}}/T_{\text{par}}$	1,003	1,094	1,610	1,581	1,293

Tabela 3: Resultados no LU.

A perda de desempenho mostrada nos gráficos da Figura 7 deve-se, em parte, 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, na última fase não

existe comunicação entre os processadores, e a fase onde é possível a sobreposição é curta em relação as outras fases.

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, quando não existe comunicação entre os processadores. Esta fase foi implementada na *thread* de computação e isto contribuiu para a perda de desempenho porque ocorrem 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 Pthreads não permite dar exclusividade a uma *thread* uma vez iniciado o processamento.

No programa original do *kernel LU* existem laços que calculam os endereços dos blocos que estão sendo processados e o número de mensagens que serão enviadas. Na implementação multithread as *threads* foram dividas em *thread* de computação e *thread* de comunicação, e houve portanto a duplicação os laços que calculam os endereços dos blocos correntes e o número de mensagens. A *thread* de comunicação realiza processamento adicional para computar estes valores, o que também contribui para a perda de desempenho.

5 Ordenação Radix

Funções de ordenação são amplamente utilizadas em aplicações tais como sistemas de gerenciamento de bancos de dados e aplicações aeroespaciais. O *kernel Radix* é um algoritmo de ordenação de números inteiros e é descrito [12]. A avaliação deste *kernel* é relevante porque sua execução requer a transmissão de grande quantidade de dados entre os processadores durante a ordenação.

A cada processador é designado o mesmo número N de chaves para serem ordenadas. Cada iteração do algoritmo é composta por três fases: (1) cada processador examina todas as chaves em seu poder e constrói um *histograma local* que representa a distribuição dos dígitos na porção de chaves em seu poder, (2) todos os processadores combinam os seus histogramas locais em um *histograma global* que indica a soma da distribuição total de dígitos nas chaves, e (3) os processadores fazem uso do histograma global para fazer a permutação das chaves que cada processador possui. As chaves são escritas uma a uma em um vetor e são enviadas ao processador destino. Ocorrem iterações até que todos os dígitos tenham sido verificados e a ordenação tenha sido finalizada. Um esquema do algoritmo do *kernel Radix* é mostrado na Figura 8.

Em contraste com outros algoritmos de ordenação, *Radix* não utiliza apenas comparações para determinar a ordem das chaves. O algoritmo analisa 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 [13].

O algoritmo original examina as chaves que serão ordenas 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, as chaves são ordenadas de acordo com o bloco de r bits corrente na iteração. O algoritmo requer b/r passos.

Inicialmente, é determinada a posição (*rank*) de cada chave, que é a posição da chave na ordem de saída. As chaves são então trocadas nas posições determinadas por *rank*. A posição de cada chave é definida pela expressão [*(chave≫deslocamento) & mascara*], onde *chave* é o número que está sendo analisado, *deslocamento* é o produto *iteração*$\times r$. *Iteração* varia de 0 a b/r, e *mascara* é $(b-1)$.

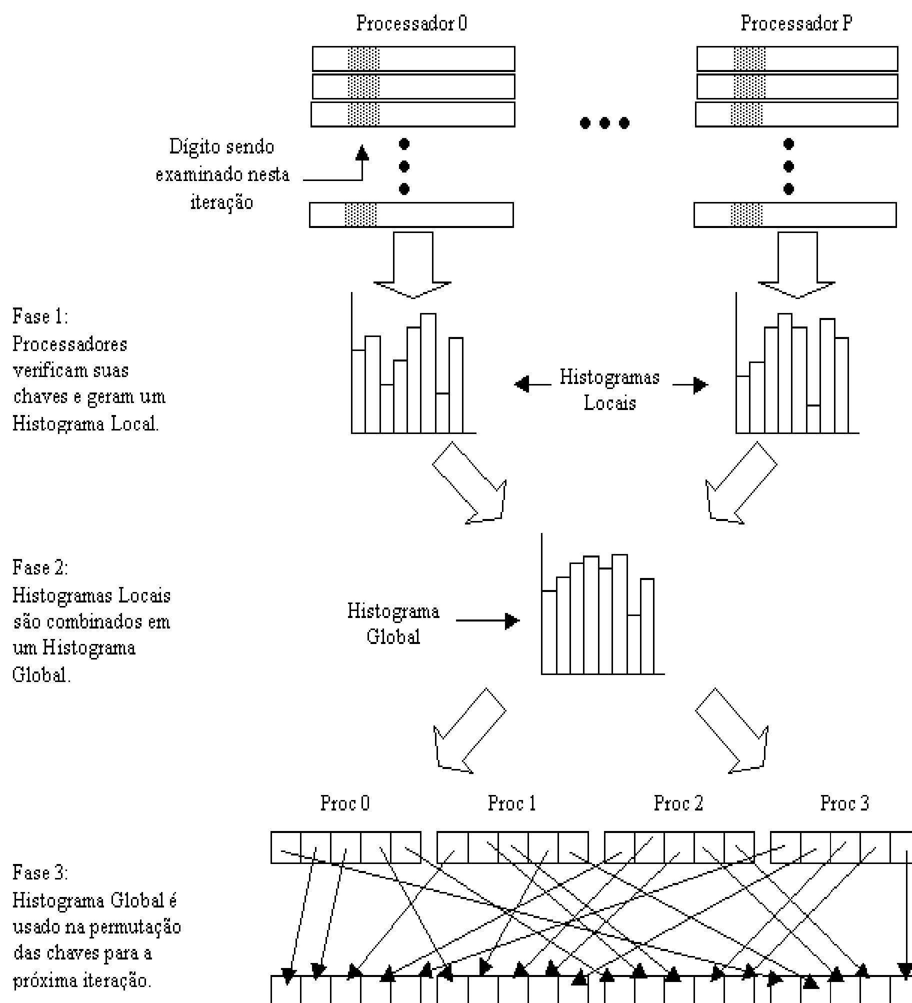

Figura 8: Algoritmo do kernel Radix.

O conjunto de dados do *kernel Radix* consiste das chaves a serem ordenadas, armazenadas em vetores de inteiros, e um segundo vetor de inteiros que atua como uma estrutura intermediária. As chaves são copiadas do vetor de inteiros para a estrutura intermediária durante as iterações. O tamanho dos vetores é igual ao número de chaves a processar. São ainda necessárias estruturas para armazenar o histograma local e o global. Cada processador mantém o seu histograma local e uma cópia do histograma global.

Um fator que influencia fortemente o desempenho do algoritmo *Radix* é o valor de r. O algoritmo executa uma iteração para cada bloco de r das chaves, e portanto 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 [12].

A comunicação neste algoritmo ocorre principalmente na terceira fase de cada iteração, quando as chaves são permutadas entre os processadores, com padrão *todos-para-todos*. Comunicação também ocorre na segunda fase do algoritmo, quando os histogramas locais são agrupados no histograma global, embora esta seja 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 processador em uma única transferência, como mostra a Figura 9.

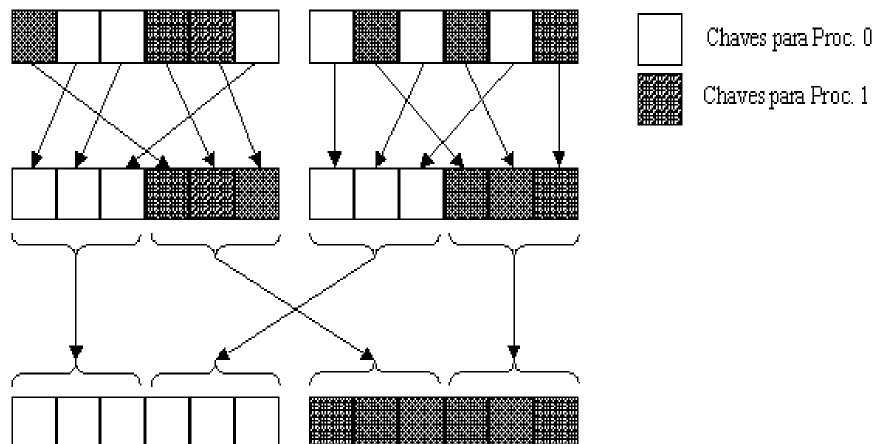

Figura 9: Fase de permutação das chaves.

5.1 Versão com threads da Ordenação Radix

O *kernel Radix* é composto de três funções principais, uma para cada fase do algoritmo. A função `histogram` é responsável por calcular o histograma local, o que envolve apenas processamento local. A função da segunda fase é denominada `all_scan_bucket`, e realiza uma varredura 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 P_0 e é finalizada no processador P_n. Após a compilação do histograma global apenas o processador P_n detém o histograma global e envia-o aos outros processadores. Nesta fase o número de mensagens é $2(n{-}1)$ e o tamanho das mensagens é $2^r \times \text{sizeof}(int)$.

A função da terceira fase é denominada `all_coalesce`, quando as chaves que estão na posse dos processadores são trocadas de acordo com o histograma global. Esta função efetua a ordenação do bloco de r bits da iteração corrente e envia as chaves aos processadores destino pela função denominada `handle_outgoing_msgs`, e recebe as chaves provenientes dos outros processadores com a função `handle_incoming_msgs`. Nestas, a comunicação é não-bloqueante e esta é a fase que apresenta o maior volume de comunicação.

No desenvolvimento da versão com *threads* do *kernel Radix* foi utilizada uma abordagem diferente daquela dos *kernels* LU e FFT, com relação as tarefas atribuídas à cada *thread*. Neste caso, o programa é composto por três funções principais, denominadas `radix_comput`, `radix_incoming` e `radix_outgoing`. A *thread* `radix_comput` cria o histograma local, o histograma global, e prepara a distribuição das chaves entre os processadores. É importante notar que esta *thread* também realiza comunicação na fase em que o histograma global é criado, e que neste *kernel* ocorre comunicação em todas as *threads*.

Esta abordagem foi empregada porque este *kernel* apresenta granularidade pequena, e o algoritmo original tem duas fases distintas, com o envio das chaves para os outros processadores, e o recebimento das chaves provenientes dos demais processadores, e estas fases se dão em momentos distintos da execução do programa. Buscou-se então paralelizar os eventos de comunicação no envio e no recebimento das mensagens. Foi necessária a inclusão de barreiras nas *threads* para garantir que todas as *threads* estejam sincronizadas a cada iteração do algoritmo. O programa original fora otimizado para enviar o maior número de chaves possíveis em um bloco de 1460 bytes, que é o tamanho máximo de um pacote IP em rede Ethernet.

5.2 Resultados para Ordenação Radix

O *kernel Radix* apresentou ganho de desempenho ao utilizar-se *threads* em conjunto com a biblioteca MPI, como mostrado no gráfico da esquerda na Figura 10 e na Tabela 4. Os ganhos de desempenho são de 21,4%, 25,9%, 22,1% e 19,3% para 2, 4, 8 e 16 processadores respectivamente, e média de 22%.

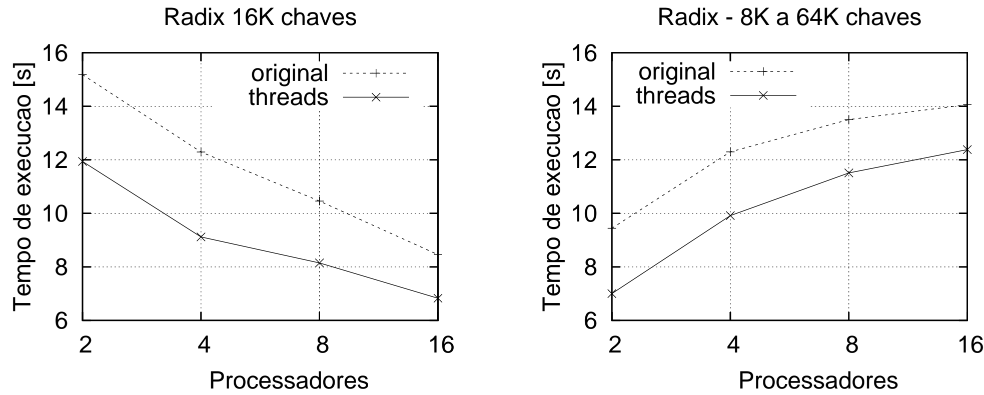

Figura 10: Resultados do kernel Radix para conjunto de dados fixo (esq) e variável (dir).

A Figura 10 também mostra o tempo de execução para o *kernel Radix*com o conjunto de dados escalado com o número de processadores. O conjunto de dados utilizado foi de 8.192 chaves para 2, 16.384 chaves para 4, 32.768 chaves para 8, e 65.536 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, e média de 18%.

	Conjunto de dados fixo (16K chaves)				
proc	2	4	8	16	méd
$T_{\text{thr}}/T_{\text{par}}$	0,786	0,742	0,779	0,807	0,778
	Conjunto de dados variável				
|conj dados|	8K	16K	32K	64K	
$T_{\text{thr}}/T_{\text{par}}$	0,741	0,806	0,852	0,881	0,818

Tabela 4: Ganhos no Radix.

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 da Figura 10.

6 Conclusão

Este trabalho descreve experimentos para avaliar possíveis ganhos de desempenho decorrentes da utilização de *threads* em conjunto com a comunicação por troca de mensagens. Para tanto, a partir das versões originais dos três *kernels* FFT, LU e Radix, foram desenvolvidas novas versões que utilizam a biblioteca para troca de mensagens MPI em conjunto com o pacote de *threads* POSIX.

Os resultados da execução destes programas em aglomerados com 2, 4, 8 e 16 processadores apresentam ganho de desempenho para os *kernels* FFT e Radix, da ordem de 15-17% e 18-22%, respectivamente. Com o *kernel LU* houve perda de desempenho na versão modificada com relação à versão paralela original, da ordem de 60-80%. No *kernel LU*, a tentativa de sobrepor a comunicação com computação na fase em que os blocos de perímetro são atualizados introduz elevado *overhead* pela concorrência entre as *threads* em fases quando não existe comunicação entre os processadores.

Com base nos resultados dos experimentos realizados, observa-se que a utilização da programação *multithread* em conjunto com a troca de mensagens do padrão MPI pode trazer redução significativa no tempo de execução de programas paralelos. Contudo, a utilização de *threads* em conjunto com MPI causa aumento significativo na complexidade dos programas por conta do paralelismo intra-nó, além da complexidade inerente envolvida no desenvolvimento de um programa paralelo. Além disso, ganhos de desempenho com a utilização de *threads* em conjunto com MPI nem sempre são garantidos porque dependem do paralelismo intrínseco à aplicação.

Referências

[1] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[2] Helio Marci de Oliveira. *Modelagem e Predição de Desempenho de Primitivas de Comunicação MPI*. Dissertação de Mestrado, Universidade de São Paulo, São Paulo, 2003.

[3] S C Woo, M Ohara, E Torrie, J P Singh, and A Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc ISCA-22*, pages 24–36, June 1995.

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

[5] 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.

[6] Jun Seong 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*, pages 202–216, 1998.

[7] A Gupta, J L Hennessy, K Gharachorloo, T Mowry, and W-D Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proc ISCA 18*, pages 254–263, May 1991.

[8] T M Warschko, C G Herter, and W F Tichy. Latency hiding in parallel systems: A quantitative approach. In *Proc ASPLOS V*, pages 51–61, 1992.

[9] D E Culler, R M Karp, D A Patterson, A Sahay, K E Schauser, E Santos, R Subramonian, and T von Eicken. LogP: Towards a realistic model of parallel computation.

In *4th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming*, pages 262–273, 1993.

[10] Kay A Robbins and Steven Robbins. *Practical Unix Programming – A Guide to Concurrency, Communication, and MultiThreading*. Prentice-Hall, 1996.

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

[12] G E Blelloch and C E Leiserson and B M Maggs and C G Plaxton and S J Smith and M Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *ACM Proc Symp on Parallel Algorithms and Architectures*, pages 3–16, 1991.

[13] John L Hennessy and David A Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, third edition, 1997.