Implementação paralela da transformada de distância euclidiana exata

Julio Cesar Torelli

Implementação paralela da transformada de distância euclidiana exata

Julio Cesar Torelli

Orientador: ***Prof. Dr. Odemir Martinez Bruno***

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

"VERSÃO REVISADA APÓS A DEFESA"

Data da Defesa:	19/08/2005
Visto do Orientador:	

USP – São Carlos
Outubro/2005

Aos meus pais, que me proporcionaram a educação que me permitiu chegar até aqui.

Agradecimentos

Ao Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (ICMC-USP), pela oportunidade da realização do curso de mestrado.

Ao prof. Odemir Martinez Bruno, pela orientação, disposição e entusiasmo. Pelo bom humor e paciência.

Ao prof. Luciano da Fontoura Costa, do Instituto de Física de São Carlos da Universidade de São Paulo (IFSC-USP), por permitir o uso do agregado de computadores do seu grupo de pesquisa.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), pela concessão da bolsa de mestrado.

Aos professores e funcionários do ICMC-USP.

Aos colegas da pós-graduação, Daniel, Christian, Rodrigo Plotze, Maurício, Mário Pazoti, Ricardo Fabbri, e a todos os outros que fizeram deste, um período de crescimento, discussões, trocas e lazer.

Resumo

Transformada de distância euclidiana (TDE) é a operação que converte uma imagem binária composta de pontos de objeto e de fundo em outra, chamada mapa de distâncias euclidianas, onde o valor armazenado em cada ponto corresponde à menor distância euclidiana entre este ponto e o fundo da imagem. A TDE é muito utilizada em visão computacional, análise de imagens e robótica, mas é uma transformação muito demorada, principalmente em imagens 3-D. Neste trabalho são utilizados dois tipos de computadores paralelos, (i) multiprocessadores simétricos (SMPs) e (ii) agregados de computadores, para reduzir o tempo de execução da TDE. Dois algoritmos de TDE são paralelizados. O primeiro, um algoritmo de TDE por varredura independente, é paralelizado em um SMP e em um agregado. O segundo, um algoritmo de TDE por propagação ordenada, é paralelizado no agregado.

Abstract

The Euclidean distance transform is the operation that converts a binary image made of object and background pixels into another image, the Euclidean distance map, where each pixel has a value corresponding to the Euclidean distance from this pixel to the background. The Euclidean distance transform has important uses in computer vision, image analysis and robotics, but it is time-consuming, mainly when processing 3-D images. In this work two types of parallel computers are used to speed up the Euclidean distance transform, (i) symmetric multiprocessors (SMPs) and (ii) clusters of workstations. Two algorithms are parallelized. The first one, an independent line-column Euclidean distance transform algorithm, is parallelized on a SMP, and on a cluster. The second one, an ordered propagation Euclidean distance transform algorithm, is paralellized on a cluster.

Sumário

Capítulo 1

Introdução

1.1 Panorama do trabalho

Transformada de distância (TD) (ROSENFELD; PFALTZ, 1966) é a operação que converte uma imagem binária[1] composta de pontos de objeto (*foreground*) e de fundo (*background*) em outra, usualmente chamada *mapa de distâncias*, onde o valor armazenado em cada ponto corresponde à distância entre este ponto e o ponto de fundo mais próximo (CUISENAIRE, 1999; EGGERS, 1998).

A distância entre os pontos de uma imagem pode ser calculada com diferentes métricas. A métrica mais natural, e também a mais utilizada em processamento de imagens, é a *euclidiana*, principalmente porque ela é invariante à rotação (ZAMPIROLLI, 2003). A transformada de distância que utiliza a métrica euclidiana é chamada *transformada de distância euclidiana* (TDE).

A TDE é amplamente utilizada em processamento de imagens. São exemplos de aplicações/transformações onde ela é utilizada: dilatação e erosão (RUSS, 2002), diagramas de Voronoi (YE, 1988), esqueletização (SHIH; PU, 1995), *watershed* (VICENT; SOILLE, 1991) e dimensão fractal (FALVO; BRUNO, 2003). A TDE também é utilizada em navegação robótica (ZEILINSKY, 1992).

O primeiro algoritmo de transformada de distância euclidiana foi proposto em 1980 por Danielsson (DANIELSSON, 1980). Depois dele muitos outros foram propostos. Alguns destes algoritmos são ditos *não exatos* porque eles não computam um mapa de distâncias completamente livre de erros, ao contrário dos algoritmos *exatos*. Enquanto que para algumas aplicações um mapa de distâncias não exato é aceitável, para outras um mapa com um erro mínimo pode levar a resultados indesejados. Por exemplo, o esqueleto obtido a partir de um

[1] Imagem onde o valor de cada ponto é zero ou um, sendo que o conjunto de pontos com valor zero representa o fundo da imagem e o conjunto de pontos com valor um representa o objeto.

mapa de distâncias não exato pode ficar desconectado (GE; FITZPATRICK, 1996), violando uma propriedade fundamental desta representação.

A Figura 1.1 mostra uma imagem binária e o seu *mapa de distâncias euclidianas*. Note que, para cada ponto em (a), o ponto correspondente em (b) armazena a menor distância euclidiana entre este ponto e o fundo da imagem.

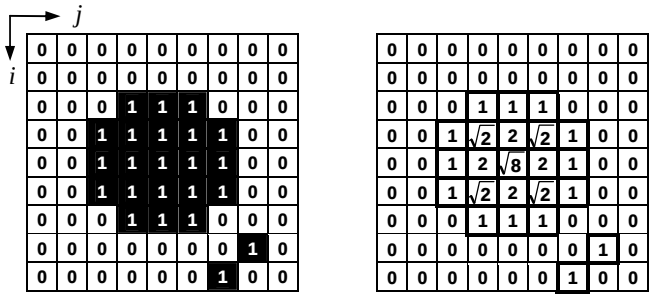


Figura 1.1. Transformada de distância euclidiana. Uma imagem binária 2-D (a) e o seu mapa de distâncias euclidianas (b).

Apesar do conceito de transformada de distância ser simples, os algoritmos exatos de TDE são difíceis de entender, implementar e, principalmente, apresentam elevado tempo de execução, principalmente os algoritmos 3-D. Por exemplo, o algoritmo de TDE de Saito e Toriwaki (SAITO; TORIWAKI, 1994), que está entre os mais rápidos da literatura (CUISENAIRE, 1999; FABBRI, 2004), pode levar aproximadamente 3 minutos para computar o mapa de distâncias de uma imagem 700×700×700 em um PC com um processador Pentium 4 de 2.8 GHz e 1.5 GB de memória RAM. É por isso que tem sido pesquisado o uso de *computadores paralelos* para executar a TDE.

Por computador paralelo entende-se um sistema computacional contendo múltiplos processadores que cooperam para resolver um problema mais rapidamente. Yamada (1984) foi o primeiro a propor um algoritmo paralelo de TDE exata, um algoritmo SIMD para *meshes*. Chen e Chuang (1995) também propuseram algoritmos SIMD para *meshes*. Algoritmos SIMD para hipercubos foram propostos por Lee et al. (1996) e Lee et. al (1997). Algoritmos SIMD também foram propostos por Takala e Viitanen (1999).

Embora existam diversos trabalhos que utilizam computadores paralelos para executar a TDE, não existem trabalhos que utilizam multiprocessadores simétricos (SMPs), nem agregados de computadores (*clusters*), que são os tipos de computadores paralelos mais comuns/utilizados atualmente.

1.2 Objetivo

O principal objetivo deste trabalho é reduzir o tempo de execução da transformada de distância euclidiana exata 3-D[2] através da sua paralelização em multiprocessadores simétricos e em agregados de computadores.

1.3 Metodologia

O primeiro passo na busca de tal objetivo foi a escolha do algoritmo a ser paralelizado**.** Diversos algoritmos exatos de TDE existem, alguns bastante eficientes, outros extremamente lentos. Quando se paraleliza um problema o correto/ideal é escolher o melhor algoritmo seqüencial. Dos algoritmos seqüenciais existentes, seis (CUISENAIRE; MACQ, 1999a; EGGERS, 1998; LOTUFO; ZAMPIROLLI, 2001; MAURER; QI; RAGHAVAN, 2003; SAITO; TORIWAKI, 1994; SHIH; WU, 2004a) são com freqüência referenciados na literatura por serem considerados os mais rápidos e serem, segundo seus autores, exatos. Estes algoritmos foram estudados e comparados por Fabbri (2004) e pelo autor do presente trabalho[3]. Por meio deste estudo comparativo, que constituiu a base para a escolha dos algoritmos paralelizados no presente trabalho, verificou-se que um destes algoritmos, o de Shih e Wu, não é exato[4]. Por isso ele foi desconsiderado. Dos cinco algoritmos restantes, três (LOTUFO; ZAMPIROLLI, 2001; MAURER; QI; RAGHAVAN, 2003; SAITO; TORIWAKI, 1994) pertencem a uma classe de algoritmos de TDE que Cuisenaire (1999) chama de *varredura independente*, e os outros dois (CUISENAIRE; MACQ, 1999a; EGGERS, 1998) a uma classe que ele chama de *propagação ordenada*. Os algoritmos de uma mesma classe têm em comum a forma como eles lêem/processam a imagem. Os algoritmos de TDE por varredura independente processam cada linha da imagem independentemente umas das outras. Em seguida, utilizando os valores armazenados nas linhas, processam as colunas. Os algoritmos de TDE por propagação ordenada calculam as distâncias dos pontos de borda (ou seja, dos pontos de objeto com pelo menos um ponto de fundo na sua vizinhança) e então

[2] O processamento de imagens volumétricas tem se tornado cada vez mais comum. Por envolver uma enorme quantidade de dados, o tempo de execução dos algoritmos 3-D de TDE é muito maior que o dos 2-D. Por isso a paralelização dos algoritmos 3-D é ainda mais importante.

[3] Os resultados deste estudo não serão detalhados aqui uma vez que se encontram na dissertação do aluno Ricardo Fabbri e em um artigo que está sendo desenvolvido e que será submetido a uma revista internacional.

[4] O autor do presente trabalho implementou um algoritmo 3-D do mesmo autor, publicado em um outro artigo (SHIH; WU, 2004b), e verificou que ele também não é exato.

transmitem/propagam estas informações para os outros pontos da imagem em ordem crescente de distâncias.

Nos estudos realizados por Fabbri (2004) com a colaboração do autor do presente trabalho, o algoritmo de Maurer Jr., Qi e Raghavan foi o mais rápido em grande parte das imagens teste. Em algumas imagens, no entanto, ele foi mais lento que todos. Já o algoritmo de Saito e Toriwaki foi, em média, bastante rápido e, o mais importante, não foi o mais lento com nenhuma delas. Além disso, este algoritmo é um dos mais fáceis de entender e implementar. O algoritmo de Lotufo e Zampirolli não foi o mais rápido com nenhuma das imagens. O algoritmo de Eggers apresentou desempenhos extremos, sendo o mais rápido em algumas imagens, mas também o mais lento em várias outras. O algoritmo de Cuisenaire e Macq apresentou desempenho médio superior ao de Eggers, mas inferior ao de Maurer Jr, Qi e Raghavan e Saito e Toriwaki.

Pelos motivos descritos acima o algoritmo escolhido para ser paralelizado foi o de Saito e Toriwaki. Para este algoritmo foram desenvolvidas estratégias de paralelização para SMPs e para agregados de computadores. Estas estratégias foram implementadas em um SMP e em um agregado. Os resultados da implementação no SMP foram excelentes. Os resultados da implementação no agregado foram apenas razoáveis quando imagens menores que 700×700×700 foram processadas. Em busca de melhores resultados no agregado decidiu-se pela paralelização de mais um algoritmo no agregado. Apesar do algoritmo de Maurer Jr, Qi e Raghavan ser o mais rápido entre os algoritmos restantes, ele é um algoritmo de TDE por varredura independente, assim como o de Saito e Toriwaki. Como os algoritmos de uma mesma classe lêem/processam a imagem da mesma forma, as possibilidades de paralelização destes algoritmos, e possivelmente os resultados desta paralelização, devem ser muito parecidos. Por isso os algoritmos de Maurer Jr, Qi e Raghavan e Lotufo e Zampirolli não foram escolhidos. Dos algoritmos de TDE por propagação ordenada, o melhor, segundo Fabbri (2004), é o de Cuisenaire e Macq. Porém, Cuisenaire e Macq utilizam um algoritmo 2-D de TDE por propagação ordenada em cada plano da imagem; depois utilizam parte do algoritmo de Saito e Toriwaki para calcular as distâncias 3-D. Por isso, a versão 3-D dos algoritmos de Saito e Toriwaki e de Cuisenaire e Macq são muito parecidas, e as possibilidades de paralelização destes algoritmos são praticamente as mesmas. Assim, o outro algoritmo escolhido para ser paralelizado foi o de Eggers.

1.4 Organização do trabalho

Este trabalho está dividido da seguinte forma. No Capítulo 2 a TDE é formalmente definida. Neste mesmo capítulo é apresentada uma revisão sobre algoritmos e aplicações da TDE. No Capítulo 3 é apresentada uma revisão sobre computação paralela / processamento paralelo, onde são descritas as arquiteturas dos computadores paralelos utilizados neste trabalho, multiprocessadores simétricos e agregados de computadores, bem como as ferramentas utilizadas para a programação nestes computadores, OpenMP e MPI. No Capítulo 4 são descritos os algoritmos de TDE escolhidos para a paralelização, bem como as estratégias de paralelização desenvolvidas. Ainda neste capítulo são apresentados detalhes de implementação destas estratégias em um SMP e em um agregado. Os resultados (tempo de execução) obtidos com estas implementações também são apresentados. Finalmente, no Capítulo 5, é feita a conclusão, uma lista das principais contribuições deste trabalho e uma lista de atividades que podem ser desenvolvidas futuramente.

Capítulo 2

Transformada de distância euclidiana

Este capítulo tem por objetivo estabelecer o conceito de transformada de distância euclidiana, uma vez que na literatura são utilizadas definições ligeiramente diferentes que podem causar confusão. Além disso, diversos conceitos e convenções necessários à compreensão deste trabalho, bem como os principais algoritmos de TDE, são apresentados.

2.1 Definições

Dada uma imagem binária *F* contendo um objeto *O*, a transformada de distância euclidiana gera uma imagem *M*, usualmente chamada de mapa de distâncias euclidianas, onde o valor armazenado em cada ponto *p* é a menor distância euclidiana entre este ponto e o fundo da imagem *O'*:

$$M(p) = \min \{\text{dist}_e(p, q) \mid q \in O'\} \tag{2.1}$$

É chamado 'objeto', *O*, o conjunto de pontos com valor 1. Tal conjunto pode ser conexo ou não. É chamado 'fundo da imagem', *O'*, o seu complemento, que é composto pelos pontos com valor 0.

A distância euclidiana entre dois pontos, *p* e *q*, cujas coordenadas são, respectivamente, (*i*, *j*) e (*x*, *y*), é dada por:

$$dist_e(p,q) = dist_e((i,j),(x,y)) = \sqrt{(i - x)^2 + (j - y)^2} \tag{2.2}$$

2.2 Aplicações da TDE

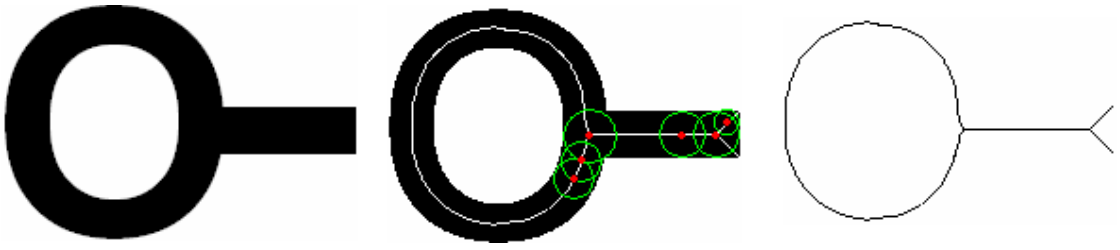
A TDE é um instrumento importante para as áreas de visão, análise de imagens, robótica, entre outras, sendo utilizada em diversas aplicações/transformações como:

- **Erosão e dilatação**. Erosão e dilatação são, respectivamente, transformações através das quais são removidos e adicionados pontos nas bordas do objeto de uma imagem binária (Figura 2.1[5]) com o objetivo de separar porções deste objeto que foram unidas, ou de unir porções que foram separadas durante o processo de binarização (RUSS, 2002). Se o mapa de distâncias de uma imagem for limiarizado (RUSS, 2002) em um nível *r*, obtém-se uma imagem onde os pontos de objeto são aqueles cuja distância em relação ao fundo é maior que *r*. Isso é o mesmo que erodir a imagem original por um disco de raio *r*. Portanto, uma vez calculado o mapa de distâncias da imagem, basta limiarizá-lo para obter a erosão com o raio desejado. De forma semelhante, se o mapa de distâncias da imagem invertida[6] for limiarizado têm-se a dilatação da imagem original.

(a) (b) (c)

Figura 2.1. Erosão e dilatação. Imagem original (a); imagem após erosão (b); e imagem após dilatação (c).

- **Esqueletização**. Esqueleto é a representação filiforme de um objeto. Formalmente ele pode ser definido como sendo o conjunto de pontos no centro dos círculos maximais inscritos no objeto (PAVLIDIS, 1982), que nada mais são do que os picos de máxima do mapa de distâncias da imagem (FALVO; BRUNO, 2003; PAGLIERONI, 1992). Portanto, uma vez calculado o mapa de distâncias, utilizando um algoritmo para detectar tais picos obtém-se o esqueleto (PAGLIERONI, 1992; SHIH; PU, 1995).



(a) (b) (c)

Figura 2.2. Esqueleto. Um objeto (a); alguns dos círculos maximais inscritos neste objeto (b); e o seu esqueleto (c).

[5] Estas imagens foram obtidas de http://www.reindeergraphics.com/tutorial/index.shtml.
[6] Imagem onde os pontos de objeto são representados pelo valor 0, e o pontos de fundo pelo valor 1.

- **Separação de objetos sobrepostos** via segmentação por *watershed*. Suponha que se quer contar o número de células em uma imagem microscópica. Um possível problema é ilustrado na Figura 2.3(a): três células levemente sobrepostas aparecem como um único componente conexo. Para que a contagem seja correta é necessário separar as células sobrepostas. Para isto, basta inverter[7] o mapa de distâncias desta imagem, que é mostrado na Figura 2.3(b)[8], e então aplicar uma transformada *watershed* (VICENT; SOILLE, 1991) sobre ele. A imagem após a segmentação é mostrada na Figura 2.3 (c).

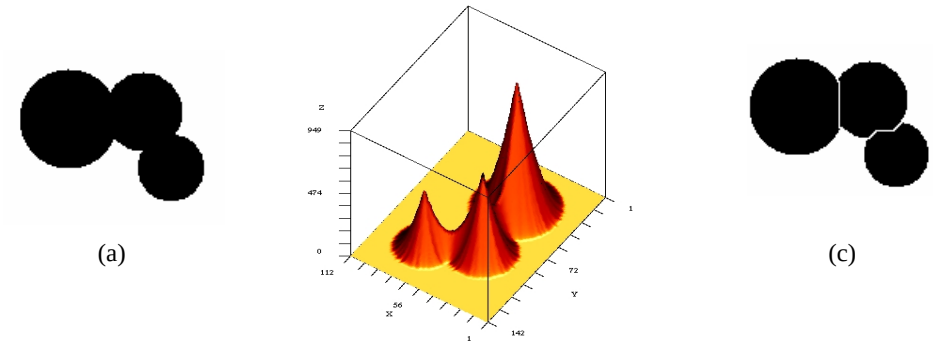


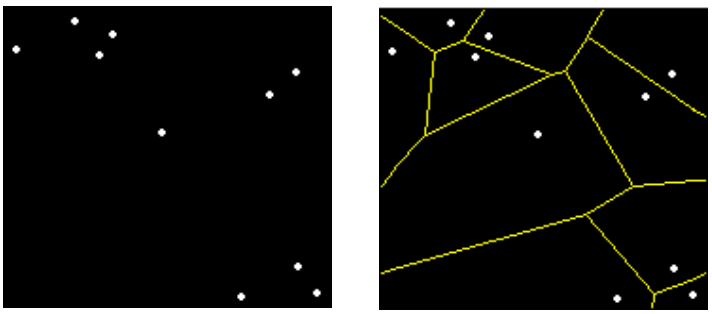
Figura 2.3. Segmentação por *watershed*. Imagem binária com três objetos (células) sobrepostos (a); mapa de distâncias euclidianas desta imagem (b); e imagem após segmentação (c).

- **Diagramas de Voronoi.** Dada uma imagem (Figura 2.4 (a)) com um conjunto isolado de pontos s_i (i = 1, 2, ..., N) chamados *sites*, o diagrama de Voronoi (Figura 2.4(b)) desta imagem consiste do seu particionamento em N regiões de forma que em uma região R_i estejam todos os pontos da imagem cujo *site* mais próximo é s_i (COSTA; CESAR JUNIOR, 2001; SAITO; TORIWAKI, 1994). Assim como o mapa de distâncias de uma imagem armazena em cada ponto a distância até o *site* mais próximo, o seu diagrama de Voronoi armazena em cada ponto o rótulo[9] do *site* mais próximo. Assim, o diagrama de Voronoi pode ser gerado por um algoritmo de transformada de distância euclidiana que calcula e armazena no mapa a distância até o *site* mais próximo e simultaneamente no diagrama o *rótulo* deste *site*.

[7] Por inverter entende-se substituir cada valor de distância *d* por *-d*.

[8] A figura mostra o mapa de distâncias da imagem como uma superfície topográfica com alturas proporcionais às distâncias.

[9] O rótulo é a informação que identifica um *site* unicamente. Por exemplo, cada *site* pode ser identificado por um número.



(a) (b)

Figura 2.4. Diagrama de Voronoi. Imagem com dez *sites* (pontos brancos) (a); e o diagrama de Voronoi desta imagem (b).

- **Dimensão Fractal** é definida por Costa e Bianchi (2002) como sendo uma medida de complexidade de um objeto. Na prática a dimensão fractal pode ser definida como sendo uma medida efetiva da superfície de contato de um objeto com o seu meio (COSTA, BIANCHI, 2002). Com a transformada de distância é possível determinar a dimensão fractal de um objeto mapeando todo o espaço (interno e externo) com o qual a borda do objeto possui interface (FALVO; BRUNO, 2003). Aplica-se então o método de "*Minkowsky-Sausage*" para a determinação da dimensão fractal do objeto (TRICOT, 1995).

- **Navegação robótica.** Um robô pode utilizar a TDE para gerar o mapa de distâncias da imagem da cena onde ele está, cujos pontos brancos representam obstáculos, e utilizar este mapa para, por exemplo, evitar colisões enquanto ele se movimenta (KOLOUNTZAKIS; KUTULAKOS, 1992; ZEILINSKY, 1992).

- **Medidas de forma relacionadas à distância** (ROSENFELD; PFALTZ, 1968)**.** O máximo do mapa de distâncias, por exemplo, é a largura do objeto contido na imagem. A distribuição das distâncias do mapa também é um descritor bastante útil de uma forma (FABBRI, 2004).

- Aplicações da TDE também podem ser encontradas em **botânica** (TRAVIS; HIRST; CHESSON, 1996), **medicina** (COSTA, 2000; SAITO; TORIWAKI, 1994) e **geologia** (PARKER, 1997).

2.3 Algoritmos de TDE

2.3.1 TDE por força bruta

A aplicação direta do conceito de transformada de distância euclidiana conduz ao seguinte algoritmo, que é freqüentemente chamado de *TDE por força bruta*: (i) para cada ponto *p*, calcule a sua distância em relação a cada ponto de fundo; (ii) defina o valor de *p* no mapa como sendo a menor destas distâncias. Obviamente, se *p* é um ponto de fundo ele já possui o seu valor final, que é a distância zero.

A complexidade do algoritmo de TDE por força bruta depende do conteúdo da imagem. Para uma imagem $n \times n$ com k pontos de objeto e, portanto, $n^2 - k$ pontos de fundo, são necessárias $k * (n^2 - k)$ comparações. Assim, se $k = n^2 / 2$, ou seja, se metade dos pontos da imagem são pontos de objeto, o número de comparações necessárias é $n^2 / 2 * (n^2 - n^2 / 2) = n^4 / 4$. Portanto, ele é um algoritmo $O(n^4)$.

2.3.2 Algoritmos eficientes de TDE

Considerando que para computar o mapa de distâncias de uma imagem todos os pontos desta imagem devem ser visitados pelo menos uma vez, uma solução com complexidade $O(n^2)$, para imagens $n \times n$, é considerada ideal/ótima. Em busca desta solução pesquisadores têm explorado a redundância ou localidade de aspectos da métrica euclidiana de diversas formas, o que tem originado diferentes algoritmos de TD. Uma propriedade explorada por muitos algoritmos de TD é enunciada a seguir:

Propriedade. Dado um ponto *p*, existe um outro *q*, em uma vizinhança de *p*, cujo ponto de fundo mais próximo é também o ponto de fundo mais próximo de *p*. Formalmente:

$$\exists\, q \in N(p) \mid EV(p) = EV(q), \tag{2.3}$$

onde $N(p)$ representa uma vizinhança de *p*, e $EV(p)$ o elemento de Voronoi mais próximo de *p*, neste caso o ponto de fundo mais próximo de *p*.

Segundo esta propriedade é possível inferir o ponto de fundo mais próximo de *p* a partir dos seus vizinhos. Esta propriedade, no entanto, não vale para a métrica euclidiana em grades discretas, como mostra a Figura 2.5 (a). Na figura há uma imagem binária com três pontos de fundo, *a*, *b* e *c*. Nesta imagem o ponto de fundo mais próximo do ponto de objeto *p* é o ponto *a*, enquanto que o ponto de fundo mais próximo dos seus 4-vizinhos é *b* ou *c*. Assim, um algoritmo de TDE que utiliza os valores armazenados na vizinhança-de-4 de *p*

para inferir o valor de *p*, o fará de forma errada. O uso de uma vizinhança-de-8 evitaria este erro. Na imagem da Figura 2.5 (b), no entanto, mesmo o uso de uma vizinhança-de-8 não evitaria um erro no ponto *p*, porque nenhum dos seus 8-vizinhos tem o ponto *a* como ponto de fundo mais próximo. Para evitar este erro uma vizinhança ainda maior deveria ser utilizada. De acordo com Cuisenaire e Macq (1999a) o tamanho da vizinhança necessária para algumas imagens pode ser tão grande a ponto de tornar os algoritmos que a utilizam ineficientes.

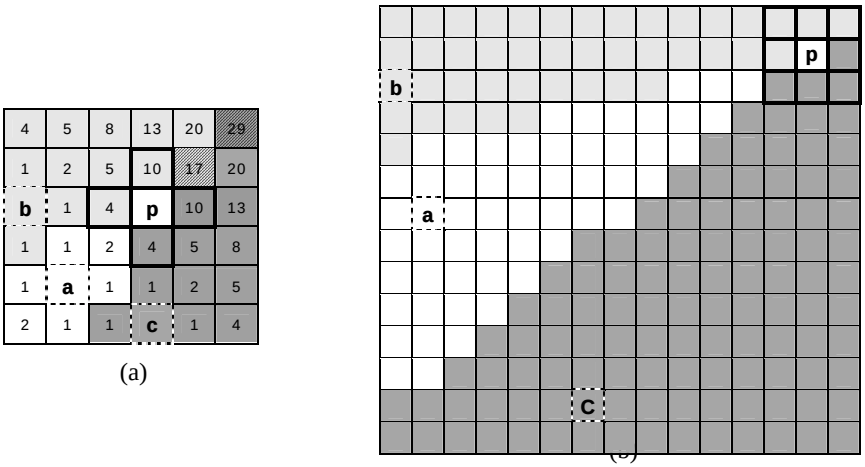


Figura 2.5. Problema da conexidade do diagrama de Voronoi discreto. Imagem com um ponto *p* cujo ponto de fundo mais próximo é diferente do ponto de fundo mais próximo dos seus 4-vizinhos (a); uma imagem com um ponto *p* cujo ponto de fundo mais próximo é diferente do ponto de fundo mais próximo dos seus 8-vizinhos (b). Nestas imagens são utilizadas cores diferentes para ilustrar as regiões de Voronoi. Os pontos com a cor branco estão mais próximo de *a*; os pontos com a cor cinza claro estão mais próximos de *b*; os ponto com a cor cinza escuro estão mais próximos de *c*; e os pontos hachurados estão à mesma distância de *b* e *c*.

Um algoritmo que não calcula a distância de todos os pontos do mapa corretamente é chamado *algoritmo de TDE não exata*. Em algumas aplicações algoritmos não exatos não podem ser utilizados porque podem levar a resultados indesejados. Por exemplo, (i) o esqueleto obtido a partir de um mapa de distâncias não exato pode ficar desconectado (GE; FITZPATRICK, 1996), violando uma propriedade crucial desta representação; (ii) na erosão de uma imagem binária podem aparecer ou faltar pontos nas bordas do objeto erodido (CUISENAIRE; MACQ, 1999a). Por isso, para estas, e para muitas outras aplicações, *algoritmos de TDE exata* são indispensáveis.

2.4 Tipos de algoritmos de TDE

Segundo Cuisenaire (1999), os algoritmos de TDE, de acordo com o método de leitura/processamento da imagem, podem ser classificados como: *TDE por varredura (raster scanning)*, *TDE por varredura independente (independent line-column)* ou *TDE por*

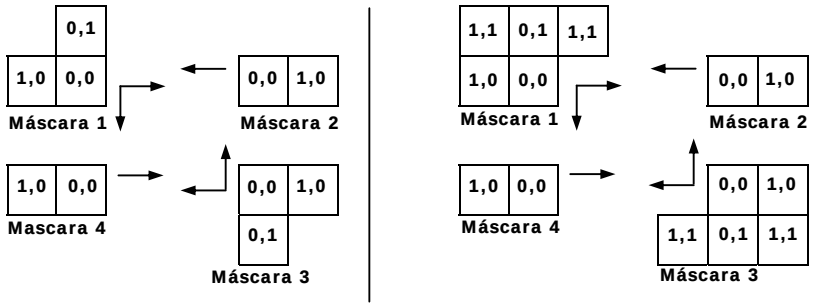
propagação ordenada (ordered propagation). Os algoritmos por varredura utilizam máscaras para processar a imagem linha por linha, de cima para baixo, da esquerda para a direita e, em seguida, no sentido contrário. Algoritmos por varredura independente processam cada linha da imagem independentemente umas das outras. Em seguida, utilizando os valores armazenados nas linhas, processam as colunas. Os algoritmos por propagação ordenada calculam a distância dos pontos de borda (ou seja, dos pontos de objeto com pelo menos um ponto de fundo na sua vizinhança) e então transmitem/propagam este informação para os outros pontos da imagem em ordem crescente de distâncias (FABBRI, 2004).

Nas seções seguintes é apresentada uma revisão sobre algoritmos de TDE. Comenta-se sobre os principais algoritmos de cada classe e, para cada classe, um algoritmo é detalhado. Tal detalhamento tem o objetivo único e exclusivo de apresentar os conceitos e de ilustrar o princípio de funcionamento dos algoritmos daquela classe. Por isso, os algoritmos escolhidos para serem detalhados foram os mais simples.

2.4.1 TDE por varredura

Os algoritmos de TDE por varredura utilizam máscaras para processar/varrer a imagem linha por linha, sendo que pelo menos duas varreduras são necessárias, uma de cima para baixo e outra de baixo para cima. O algoritmo de TDE por varredura mais conhecido é o de **Danielsson** (DANIELSSON, 1980). Embora ele não seja exato, ele é um dos algoritmos mais simples, sendo apresentado aqui com o objetivo único e exclusivo de ilustrar o princípio de funcionamento dos algoritmos desta classe.

O algoritmo de Danielsson não gera um mapa com as distâncias propriamente ditas, mas sim um mapa com os valores absolutos das coordenadas relativas do ponto de fundo mais próximo. Por isso ele é dito um algoritmo de TDE *vetorial*. Dada uma imagem binária o algoritmo gera o seu mapa de distâncias vetorial em quatro varreduras usando as máscaras da Figura 2.6 (a).



(a) (b)

Figura 2.6. Máscaras de Danielsson. Máscaras do algoritmo 4SEDT (a); e máscaras do algoritmo 8SEDT (b).

O mapa de distâncias deve ser inicializado da seguinte forma: $M(i, j) = (0, 0)$, se este é um ponto de fundo na imagem de entrada, ou $M(i, j) = (\infty,\infty)$, caso contrário ($\infty$ é qualquer número maior que o número de linhas e colunas da imagem). Começando do topo, a *máscara 1* é movida da esquerda para a direita e, em seguida, a *máscara 2* da direita para a esquerda na mesma linha, linha por linha. Em cada ponto de objeto, os vetores das máscaras são adicionados aos vetores correspondentes do mapa. O novo valor do ponto de objeto é então definido como sendo o menor destes valores/vetores. Depois a imagem é varrida de baixo para cima, da direita para a esquerda com a *máscara 3* e da esquerda para a direita com a *máscara 4*. A Figura 2.7 mostra uma imagem binária (a) e o seu mapa de distâncias antes (b) e depois (c) da execução do algoritmo.

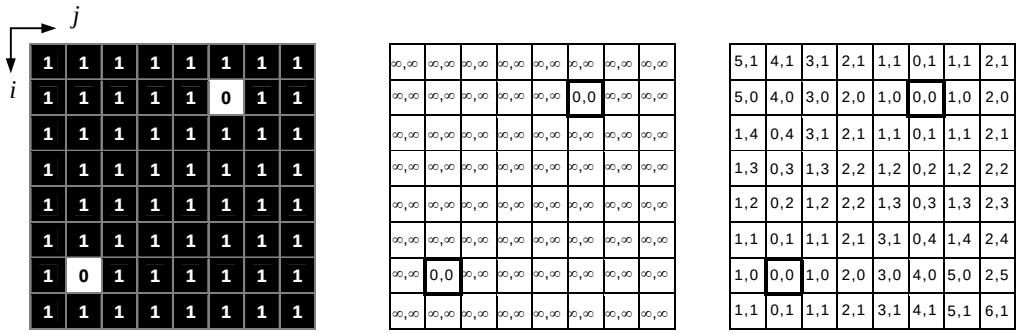


Figura 2.7. TDE de Danielsson. Uma imagem binária com dois pontos de fundo (a); o mapa de distâncias desta imagem na inicialização do algoritmo (b); e este mapa após a execução do algoritmo (c).

Este algoritmo é chamado 4SEDT (*Four-point Sequential Euclidean Distance Transformation*) porque utiliza máscaras contendo os vizinhos 4-conectados para propagar as distâncias/coordenadas. Por utilizar este tipo/tamanho de máscara, se um ponto de objeto tiver como ponto de fundo mais próximo um que não é o ponto de fundo mais próximo de um dos seus 4-vizinhos, a sua distância não será calculada corretamente. Para evitar este tipo de erro Danielsson sugere o uso de máscaras maiores, as máscaras da Figura 2.6 (b). Quando estas máscaras são utilizadas o algoritmo é chamado 8SEDT. Porém, como foi descrito na Seção 2.3.2, existem imagens com pontos cujo ponto de fundo mais próximo é diferente do ponto dos seus 8-vizinhos, e nestes pontos ocorrerão erros.

Existem vários algoritmos de TDE por varredura. Alguns destes algoritmos são apenas um aprimoramento do algoritmo de Danielsson. Por exemplo, **Ye** (1988) utiliza as mesmas máscaras de Danielsson, porém com sinal, para gerar um mapa com as coordenadas relativas com sinal do ponto de fundo mais próximo, e não apenas os seus valores absolutos.

Cuisenaire e Macq (1999b) utilizam o algoritmo 4SEDT de Danielsson e depois detectam e corrigem os erros gerados por ele. Para detectar e corrigir os erros, primeiro eles detectam os pontos nas extremidades das regiões de Voronoi geradas pelo 4SEDT. A partir destes pontos, utilizando uma equação simples eles detectam e corrigem os pontos com erro. Recentemente **Shi e Wu** (2004) propuseram um algoritmo de TDE que computa o mapa de distâncias euclidianas de uma imagem binária em duas varreduras utilizando máscaras de tamanho 3×3. Segundos os autores o algoritmo é exato. Para Fabbri (2004), no entanto, ele não é exato.

2.4.2 TDE por varredura independente

Em imagens 2-D os algoritmos de TDE por varredura independente constroem a TD 1-D para cada linha (ou coluna) independentemente e, depois, utilizando esse resultado, constroem a TD 2-D completa. O algoritmo de **Chen e Chuang** (1995) é um dos algoritmos de TDE exata por varredura independente mais fáceis de entender e implementar, sendo por isso apresentado aqui. Além da imagem/matriz de entrada, F, ele utiliza outras duas matrizes, P e Q, do mesmo tamanho da imagem. Para cada ponto (i, j) em F os correspondentes em P e Q armazenam, respectivamente, as coordenadas do ponto de fundo mais próximo acima (e inclusive) e abaixo de (i, j).

O mapa de distâncias deve ser inicializado da seguinte forma: $M(i, j) = 0$, se este é um ponto de fundo na imagem de entrada, ou $M(i, j) = \infty$, caso contrário (∞ é qualquer número maior que a máxima distância possível na imagem). As matrizes auxiliares devem ser inicializadas da seguinte forma: $P(i, j) = Q(i, j) = (\infty, \infty)$, como mostra a figura seguinte:

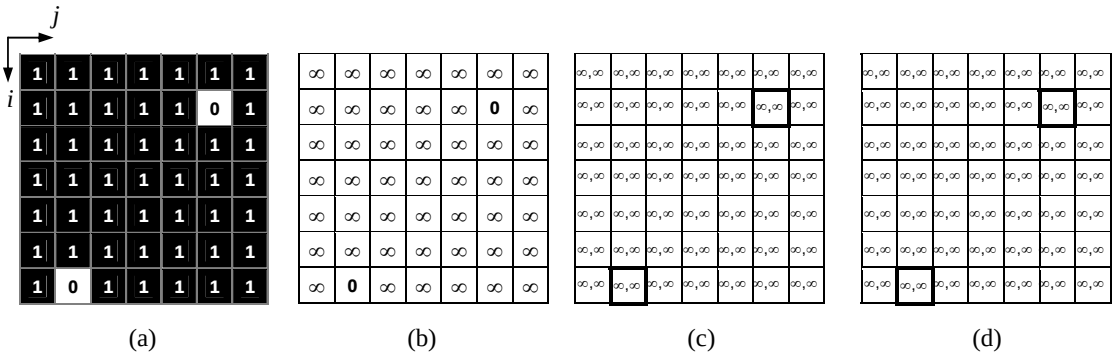


Figura 2.8. Imagens para o algoritmo de Chen e Chuang. Uma imagem binária (a); o mapa de distâncias na inicialização do algoritmo (b); e as matrizes auxiliares P (c) e Q (d) na inicialização do algoritmo.

O algoritmo realiza duas transformações 1-D, uma para cada direção de coordenada:

Transformação 1. Para cada ponto (*i, j*) encontre o ponto de fundo mais próximo acima (e inclusive) dele na coluna *j* e o ponto de fundo mais próximo abaixo dele na coluna *j*; armazene as coordenadas destes pontos em *P* e *Q*, respectivamente. O algoritmo para esta transformação é ilustrado a seguir, onde *n* e *m* representam, respectivamente, o número de linhas e colunas da imagem:

```
para  i = 0  até  n - 1  {
        para  j = 0  até  m - 1  {
                para  x = n - 1  até  0  {
                        se  F(x, j) = 0  e  x > i
                                Q(i, j) = (x, j)
                        se  F(x, j) = 0  e  x ≤ i  e  P(i, j) = (∞, ∞)
                                P(i, j) = (x, j)
                }
        }
}
```

As Figuras 2.9(a) e 2.9(b) ilustram o conteúdo das matrizes *P* e *Q* após a *transformação 1*:

∞,∞	∞,∞	∞,∞	∞,∞	∞,∞	∞,∞	∞,∞
∞,∞	∞,∞	∞,∞	∞,∞	∞,∞	1, 5	∞,∞
∞,∞	∞,∞	∞,∞	∞,∞	∞,∞	1, 5	∞,∞
∞,∞	∞,∞	∞,∞	∞,∞	∞,∞	1, 5	∞,∞
∞,∞	∞,∞	∞,∞	∞,∞	∞,∞	1, 5	∞,∞
∞,∞	∞,∞	∞,∞	∞,∞	∞,∞	1, 5	∞,∞
∞,∞	6, 1	∞,∞	∞,∞	∞,∞	1, 5	∞,∞

(a)

∞,∞	6,1	∞,∞	∞,∞	∞,∞	1,5	∞,∞
∞,∞	6,1	∞,∞	∞,∞	∞,∞	∞,∞	∞,∞
∞,∞	6,1	∞,∞	∞,∞	∞,∞	∞,∞	∞,∞
∞,∞	6,1	∞,∞	∞,∞	∞,∞	∞,∞	∞,∞
∞,∞	6,1	∞,∞	∞,∞	∞,∞	∞,∞	∞,∞
∞,∞	6,1	∞,∞	∞,∞	∞,∞	∞,∞	∞,∞
∞,∞	∞,∞	∞,∞	∞,∞	∞,∞	∞,∞	∞,∞

(b)

Figura 2.9. Matrizes após *transformação 1*. Matrizes *P* (a) e *Q* (b) após a *transformação 1*.

Transformação 2. Calcule a distância de cada ponto (*i, j*) até os pontos de fundo cujas coordenadas foram armazenadas na linha *i* em *P* e *Q* durante a primeira transformação. Defina *M*(*i, j*) como sendo a menor destas distâncias. Formalmente:

$$M(i, j) = \min \{ M(i, j),\ d_e(\ (i, j),\ P(i, y)\ )\ ,\ d_e(\ (i, j), Q(i, y));\quad 0 \le y \le m - 1 \} \qquad (2.4)$$

O algoritmo para esta transformação é ilustrado a seguir:

```
Para  i = 0  até  n - 1  {
        Para  j = 0  até  m - 1  {
                Para  y = 0  até  m - 1  {
                        Se  M(i, j) > d_e ( (i, j), P(i, y))
                                M(i, j) = d_e ( (i, j), P(i, y) )
                        Se  M(i, j)  > d_e ( (i, j), Q(i, y))
                                M(i, j) = d_e ( (i, j), Q(i, y) )
                }
        }
}
```

Note que na *transformação 1* a imagem é varrida coluna por coluna e que as colunas são varridas independentemente umas das outras. Na *transformação 2* a imagem é varrida linha por linha. Na *transformação 1* é calculada a distância de cada ponto até os pontos de fundo mais próximos na sua coluna. Utilizando esse resultado, na *transformação 2* é calculada a distância final de cada ponto. Este algoritmo, que é simples, é também ineficiente. Primeiro, porque são necessárias duas matrizes do tamanho da imagem de entrada. Segundo, porque o número de operações necessárias é $\Theta(n^3)$. Um algoritmo eficiente de TDE por varredura independente foi proposto por **Saito e Toriwaki** (SAITO; TORIWAKI, 1994). Em uma primeira transformação ele calcula a distância de cada ponto até o ponto de fundo mais próximo na sua linha. Depois, em uma segunda transformação, utilizando os valores da primeira e propriedades baseadas em interseção de parábolas, computa a distância final de cada ponto. Fabbri (2004) provou experimentalmente que este algoritmo é um dos mais rápidos da literatura, apesar de teoricamente ele ser $O(n^3)$. Segundo Cuisenaire (1999) o algoritmo de Saito e Toriwaki é um dos mais rápidos em imagens 3-D. Algoritmos eficientes de TDE por varredura independente também foram propostos por Koloutzakis e Kutulakos (1992) e Hirata (1996). Baseados nos trabalhos de Shih e Mitchell (SHI; MITCHELL, 1992) e Huang e Mitchell (HUANG; MITCHELL, 1994), **Lotufo e Zampirolli** (2001) propuseram um algoritmo que computa o mapa de distâncias erodindo cada coluna e, em seguida, cada linha da imagem por elementos estruturantes 1-D. Recentemente **Maurer Jr., Qi e Raghavan** (2003) propuseram um algoritmo de TDE por varredura independente que é $O(n^2)$. Nos experimentos realizados por Fabbri (2004) este algoritmo se mostrou mais rápido que o de Saito e Toriwaki na maioria dos casos. Entretanto, ele é um algoritmo difícil de entender, porque a descrição do algoritmo no artigo é bastante vaga, e implementar.

2.4.3 TDE por propagação ordenada

Nos algoritmos de TDE por propagação ordenada as distâncias são calculadas dos pontos de objeto mais externos para os mais internos. Primeiro são calculadas as distâncias dos pontos de borda. Estas distâncias são então transmitidas/propagadas para os seus vizinhos, que as utilizam para inferir a sua distância. Estes então propagam a sua distância para os seus vizinhos, e assim sucessivamente. A Figura 2.10 ilustra a ordem em que as distâncias são calculadas.

(a) (b) (c)

Figura 2.10. Propagação ordenada. Em uma primeira iteração os algoritmos de TDE por propagação ordenada calculam a distância dos pontos de borda (b). Na segunda calculam a distância dos vizinhos destes pontos (c), e assim sucessivamente.

Um dos algoritmos mais simples de TDE por propagação ordenada é aquele de **Lotufo, Falcão e Zampirolli** (2000), que é uma adaptação do algoritmo de Dijkstra (DIJKSTRA, 1959), um algoritmo utilizado para calcular o caminho de custo mínimo entre os vértices de um grafo. No algoritmo de Lotufo, Falcão e Zampirolli um ponto ou é *temporário* ou *permanente*. É chamado de temporário todo ponto cuja distância final ainda não foi calculada. Uma vez calculada/encontrada a distância final de um ponto ele passa a ser um ponto permanente e a sua distância não pode mais ser alterada. No mapa de distâncias cada ponto, além de armazenar a distância até o ponto de fundo mais próximo, aponta para ele, ou seja, armazena também as coordenadas do ponto de fundo mais próximo.

No inicio da execução do algoritmo todos os pontos são ditos temporários. Todo ponto de fundo armazena o valor de distância zero e aponta para ele mesmo. Todo ponto de objeto é inicializado com o valor de distância ∞ e aponta para (∞, ∞). A propagação então acontece da seguinte forma:

1. Encontre o ponto temporário *p* com a menor distância (caso mais de um ponto apresente a menor distância escolha qualquer um deles);
2. Transforme *p* em um ponto permanente;
3. Para cada ponto temporário *v* na vizinhança-de-8 de *p*, faça:
    a. Calcule a distância entre *v* e o ponto de fundo apontado por *p*;

b. Se esta distância for menor que a distância atualmente armazenada em *v*, faça:
    i. Substitua a distância atual pela nova;
    ii. Faça *v* apontar para o mesmo ponto de fundo que *p;*
4. Repita os passos 1, 2 e 3 enquanto existirem pontos temporários.

O procedimento de encontrar o ponto temporário com a menor distância (Passo 1) é o gargalo mais importante do algoritmo. Os autores sugerem o uso de uma *fila de prioridades* (KNUTH, 1998) ou de um *bucket* (KNUTH, 1998) para armazenar os pontos temporários. Utilizando uma fila de prioridades é possível encontrar o ponto com a menor distância em tempo $O(\log n)$, supondo que a fila tem n pontos temporários. Utilizando um *bucket* é possível encontrar este ponto em tempo $O(1)$.

Como o algoritmo computa a distância de cada ponto (Passo 3a) em relação à fonte (ou seja, em relação ao ponto de fundo mais próximo) apontada por algum ponto da sua vizinhança-de-8, ele não é exato, pelo motivo descrito na Seção 2.3.2. Dos algoritmos exatos de TDE por propagação ordenada dois, Eggers (1998) e Cuisenaire e Macq (1999a), se destacam por causa da sua eficiência.

O algoritmo de **Eggers** (1998), que é $O(n^3)$, consiste de uma série de iterações. Em uma iteração i um conjunto de pontos, chamado *conjunto de contorno*, propaga informação para os seus vizinhos. Os vizinhos cujas distâncias são alteradas/atualizadas formam o conjunto de contorno da próxima iteração, $i + 1$. Na maioria dos algoritmos 2-D de TDE por propagação ordenada a informação armazenada em um ponto é propagada para todos os seus 8-vizinhos. Eggers mostrou que isto não é necessário. Ele divide os pontos no conjunto de contorno em *primários* e *secundários*. Cada ponto p em qualquer um dos conjuntos tem associado a ele um índice k, que indica a direção, ou seja, o vizinho $N_k(p)$, para o qual ele deve transmitir a sua distância. Um ponto de contorno primário transmite a sua distância para apenas três dos seus vizinhos, $N_{k+1}(p)$, $N_{k-1}(p)$ e $N_k(p)$, enquanto que um ponto secundário transmite para apenas um, $N_k(p)$. O que determina se um ponto é primário ou secundário é o seu índice k. Se k é ímpar o ponto é dito primário; se k é par o ponto é dito secundário. Os vizinhos de um ponto p são numerados como a seguir:

$$\begin{array}{ccc} N_3(p) & N_2(p) & N_1(p) \\ N_4(p) & p & N_0(p) \\ N_5(p) & N_6(p) & N_7(p) \end{array}$$

Na prática Eggers utiliza quatro listas encadeadas: *ListaP1*, *ListaP2*, *ListaS1* e *ListaS2*. *ListaP1* armazena os pontos de contorno *primários* da iteração corrente e *ListaP2* os pontos de contorno *primários* da próxima iteração. Da mesma forma, *ListaS1* e *ListaS2* armazenam os pontos de contorno *secundários* da iteração corrente e da próxima, respectivamente. O algoritmo de Eggers é descrito a seguir:

1. Inicialize todo ponto de objeto com o valor de distância ∞;
2. Inicialize o *conjunto de contorno* (ou seja, as listas) fazendo, para cada ponto p:
   - Para k = 0, 2, 4 e 6, se $N_k(p)$ é um ponto de objeto insira p em *ListaP1* com índice de direção igual a $k + 1$;
3. Enquanto (*ListaP1* $\cup$ *ListaS1*) $\neq \varnothing$, faça:
   - iteração = iteração + 1;
   - addir = 2 * iteração - 1; adind = 2 * addir;

   a. Para cada ponto p em *ListaS1* com índice k faça:
      i. Se $N_k(p) > p$ + addir então
         - Atualize o vizinho N_k de p, $N_k(p)$, com o valor de distância p + addir, e insira $N_k(p)$ em *ListaS2* com índice de direção k.

   b. Para cada ponto p em *ListaP1* com índice k faça:
      i. Se $N_k(p) > p$ + addind então
         - Atualize o vizinho N_k de p, $N_k(p)$, com o valor de distância p + adind, e insira $N_k(p)$ em *ListaP2* com índice de direção k.
      ii. Se $N_{k+1}(p) > p$ + addir então
         - Atualize o vizinho N_{k+1} de p, $N_{k+1}(p)$, com o valor de distância p + addir, e insira $N_{k+1}(p)$ em *ListaS2* com índice de direção $k + 1$.
      iii. Se $N_{k-1}(p) > p$ + addir então
         - Atualize o vizinho N_{k-1} de p, $N_{k-1}(p)$, com o valor de distância p + addir, e insira $N_{k-1}(p)$ em *ListaS2* com índice de direção $k - 1$

   c. Faça *ListaP1* = *ListaP2*; *ListaP2* = $\varnothing$; *ListaS1* = *ListaS2*; e *ListaS2* = $\varnothing$;

A Figura 2.11 ilustra a execução deste algoritmo com uma imagem contendo um único ponto de fundo no centro. Note que a distância do ponto de fundo p é propagada para um ponto q através do menor caminho, que consiste de 2 * $d_\infty(p, q) - d_1(p, q)$ vizinhos diretos e

$d_1(p, q)$ - $d_\infty(p, q)$ vizinhos indiretos, onde $d_1(p, q)$ e $d_\infty(p, q)$ são, respectivamente, as distâncias *cityblock* e *chessboard* entre p e q.

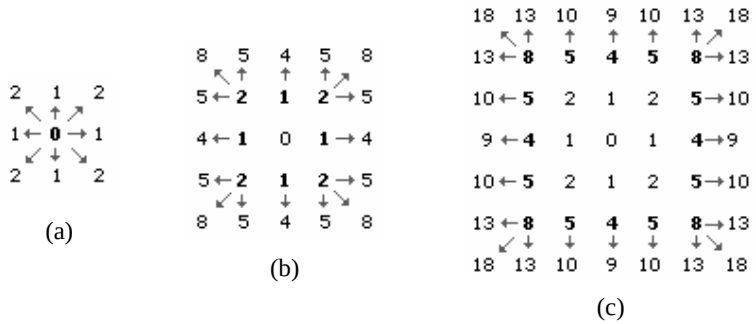


Figura 2.11. Propagação ordenada de Eggers. Na primeira iteração (a) o ponto de fundo no centro da imagem propaga a sua distância, 0 (zero), para todos os seus vizinhos; na segunda iteração (b) os pontos que foram atualizados na primeira propagam a sua distancia para os seus vizinhos; os pontos primários propagam a sua distância para três dos seus vizinhos e os pontos secundários propagam para apenas um; na terceira iteração (c) os pontos atualizados na segunda transmitem sua distância para os seus vizinhos.

Outro algoritmo eficiente de TDE por propagação ordenada é o de **Cuisenaire e Macq** (1999a). Em uma primeira etapa eles utilizam o mesmo algoritmo de Lotufo, Falcão e Zampirolli (2000), porém, com uma vizinhança-de-4. Como descrito na Seção 2.3.2, caso algum ponto de objeto tenha como ponto de fundo mais próximo um ponto diferente daquele dos seus 4-vizinhos, ocorrerão erros. Por isso Cuisenaire e Macq utilizam vizinhanças maiores nas fronteiras das regiões de Voronoi para corrigir possíveis erros. O tamanho da vizinhança utilizada é determinado em tempo de execução a partir dos valores/distâncias armazenados no mapa.

2.4.4 Outros algoritmos de TDE

Existem ainda algoritmos de TDE que não se enquadram perfeitamente em nenhuma das classes sugeridas por Cuisenaire, por exemplo, o algoritmo exato de Costa (2000). Este algoritmo utiliza uma estrutura de dados que o autor chama de SEDR (*sorted Euclidean distance representation*). Tal estrutura contém cada uma das distâncias possíveis em grades discretas e os vetores relativos (ou seja, os deslocamentos) para os pontos que apresentam aquela distância. A SEDR é implementada como uma lista encadeada, como ilustra a Figura 2.12. Nesta figura são ilustradas as três primeiras distâncias possíveis em uma imagem

discreta. Na prática o número de distâncias na SEDR depende do tamanho e do conteúdo da imagem, ou melhor, da máxima distância possível na imagem.

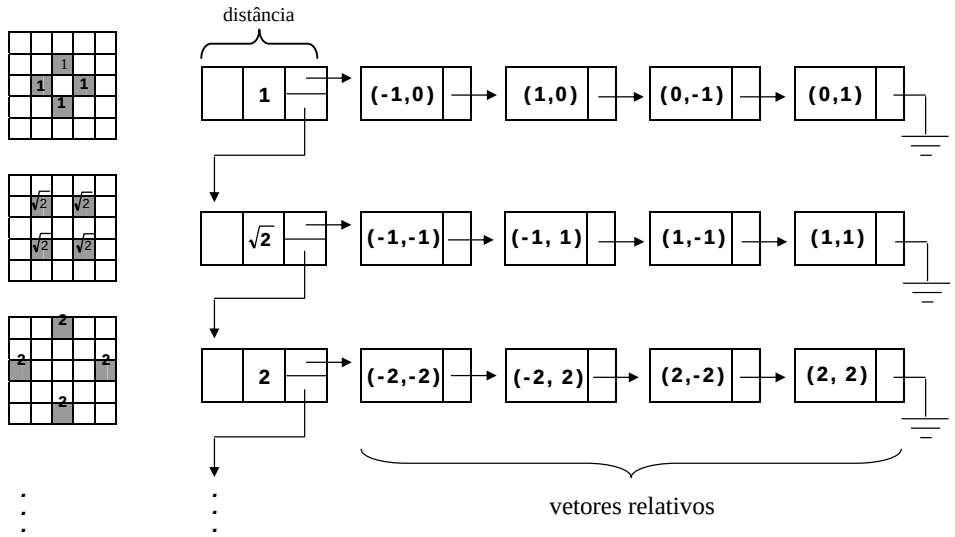


Figura 2**.12**. **SEDR (*sorted Euclidean distance representation*).** A SEDR pode ser gerada em tempo de execução ou carregada de um arquivo. Note que as distâncias são armazenadas em ordem crescente.

O mapa de distâncias deve ser inicializado da seguinte forma: $M(i, j) = 0$, se este é um ponto de fundo na imagem de entrada, ou $M(i, j) = \infty$, caso contrário (∞ é qualquer número maior que a máxima distância possível na imagem). O algoritmo age varrendo os pontos de borda para cada distância em SEDR. Em cada ponto de borda os vetores relativos da distância corrente são adicionados às coordenadas do ponto de borda. Se a distância armazenada no ponto apontado for maior que a distância corrente, ela é substituída. Este procedimento é ilustrado no código seguinte. Neste código, cada uma das distâncias na SEDR é identificada por um índice, sendo que os índices vão de 1 a *nd*, onde *nd* é o número total de distâncias na SEDR. Por exemplo, a primeira/menor distância em grades discretas é a distância 1, por isso o seu índice é o 1. A segunda menor distância é a $\sqrt{2}$, por isso o seu índice é o 2, e assim sucessivamente. Os vetores associados a uma distância também são identificados por um índice, sendo que os índices vão de 1 a *nv*. Os pontos de borda foram previamente detectados, por um algoritmo de detecção de bordas, e armazenados em *PontosBorda*. Cada ponto de borda em *PontosBorda* é identificado por um índice, sendo que os índices vão de 1 a *np*.

```
para  e = 1  até  SEDR_nd( ) {   // SEDR_nd retorna o número de distâncias na SEDR
    d = SEDR_dist(e);   // SEDR_dist(e) retorna o valor da distância com índice e
    para  v = 1  até  SEDR_nv(e) {   // SEDR_nv(e) retorna o nº de vetores associados à distância com índice e
        di = SEDR_coordenada_i (v, e);   // SEDR_coordenada_i(v, e) retorna a coordenada i do vetor v da distância e
        dj = SEDR_coordenada_j (v, e);   // SEDR_coordenada_j(v, e) recupera a coordenada j do vetor v da distância e
        para  p = 1  até  PontosBorda_np ( ) {   // PontosBorda_np ( ) retorna o número de pontos de borda
            i = PontosBorda_coordenada_i (p) + di;   // adiciona a coordenada i do vetor à coordenada do ponto
            j = PontosBorda_coordenada_j (p) + dj;   // adiciona a coordenada j do vetor à coordenada do ponto
            se M(i, j) > d
                M(i, j) = d;
        }
    }
}
```

Capítulo 3

Processamento paralelo

Este capítulo traz uma revisão sobre *processamento paralelo*[10], que é a solução empregada neste trabalho para a redução do tempo de execução da TDE. Para a realização de processamento paralelo é necessário um computador paralelo e um programa paralelo. Este capítulo tem por objetivo estabelecer estes e outros conceitos, bem como apresentar a arquitetura dos computadores e as ferramentas de programação e análise de desempenho utilizados neste trabalho.

3.1 Introdução

Desde o surgimento dos primeiros computadores muita coisa mudou. Os processadores, em especial, tornaram-se muito mais rápidos. Porém, assim como a capacidade dos processadores, a complexidade dos problemas também aumentou. Atualmente, problemas como os de simulação de problemas estruturais, de cristalografia, tomografia, dinâmica de proteínas, química quântica, meteorologia, etc., não podem ser resolvidos em tempo hábil por um computador seqüencial (isto é, por um computador contendo um único processador) porque requerem capacidade de processamento maior do que a que um processador sozinho pode oferecer. Para resolver estes problemas são então utilizados computadores paralelos.

3.2 Computadores paralelos

Por computador paralelo entende-se um sistema computacional contendo múltiplos processadores que se comunicam e cooperam para resolver grandes problemas mais rapidamente (ALMASI; GOTTLIEB, 1994). Os processadores e a memória de um computador podem ser organizados de diversas formas. Por isso existem diferentes arquiteturas e algumas taxonomias que agrupam arquiteturas com características comuns.

[10] Por processamento paralelo entende-se o uso simultâneo de múltiplos processadores para resolver um problema, neste caso a TDE, mais rapidamente.

3.2.1 Classificação dos computadores paralelos

A classificação/taxonomia proposta por Flynn (1972) é a mais conhecida. Segundo Flynn um computador pode operar com um ou múltiplos *fluxos de instruções*[11], e cada instrução pode operar sobre um ou múltiplos *fluxos de dados*[12]. Com base nas possíveis unicidade ou multiplicidade dos fluxos de instruções e de dados Flynn define quatro classes/tipos de computadores: SISD, SIMD, MISD e MIMD.

3.2.1.1 SISD (***S**ingle **I**nstruction Stream, **S**ingle **D**ata Stream*)

Nos computadores da classe SISD (Figura 3.1) existe um único fluxo de instruções. Estas instruções são executadas seqüencialmente e cada instrução opera sobre um único fluxo de dados. Pertencem a esta classe os computadores seqüenciais (HENNESSY; PATTERSON, 2003; STALLINGS, 2002) – apesar dos computadores seqüenciais atuais utilizarem processadores superescalares (STALLINGS, 2002), que são capazes de executar um número limitado de instruções de um programa em paralelo, eles ainda são considerados arquiteturas SISD, porque possuem uma única unidade de controle, isto é, um único fluxo de instruções (QUINN, 1994).

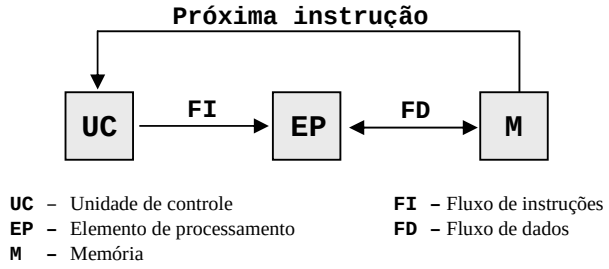


Figura 3.1. Arquitetura SISD. Arquiteturas SISD possuem um único fluxo de instruções que opera sobre um único fluxo de dados.

3.2.1.2 SIMD (***S**ingle **I**nstruction Stream, **M**ultiple **D**ata Streams*)

Em arquiteturas SIMD (Figura 3.2) existe um único fluxo de instruções que opera sobre múltiplos fluxos de dados. Nestas máquinas uma única unidade de controle comanda vários elementos de processamento que executam simultânea e sincronizadamente a mesma instrução sobre conjuntos de dados distintos (por exemplo, sobre os vários elementos de um vetor). Dentro desta categoria estão os processadores vetoriais e matriciais (STALLINGS, 2002; TANENBAUM; GOODMAN, 1999).

---

[11] Um fluxo de instruções corresponde a um contador de programas; assim, um sistema com *n* CPUs tem *n* contadores de programa e, portanto, *n* fluxos de instruções (TANENBAUM; GOODMAN, 1999).

[12] Um fluxo de dados corresponde a um conjunto de operandos (TANENBAUM; GOODMAN, 1999).

3.2.1.3 MISD (*Multiple Instruction Streams, Single Data Stream*)

Em arquiteturas MISD, por definição, múltiplos fluxos de instruções operam sobre um mesmo fluxo de dados ao mesmo tempo. Na prática, computadores com esta arquitetura nunca foram implementados (HENNESSY; PATTERSON, 2003; STALLINGS, 2002).

3.2.1.4 MIMD (*Multiple Instruction Streams, Multiple Data Streams*)

Em arquiteturas MIMD existem dois ou mais processadores, cada qual executando um fluxo de instruções distinto sobre um fluxo de dados próprio. Existem, portanto, múltiplas unidades de controle que operam simultaneamente, porém independentemente umas das outras (Figura 3.3). Na prática, isso significa a possibilidade de execução de vários programas diferentes, ou de várias instâncias de um mesmo programa, ao mesmo tempo. Atualmente, a maioria dos sistemas de computação utilizados para processamento paralelo são do tipo MIMD.

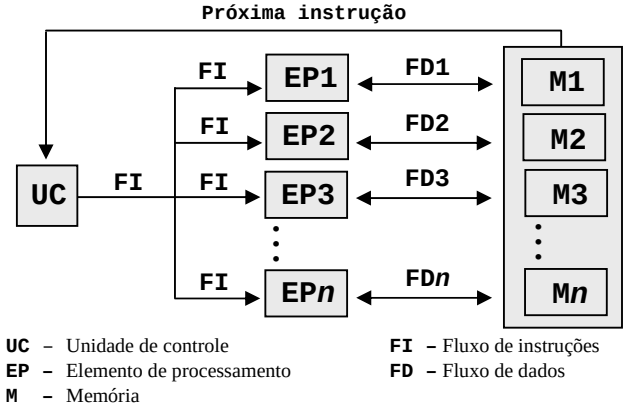


Figura 3.2. Arquitetura SIMD. Arquiteturas SIMD possuem um único fluxo de instruções; porém, cada instrução é executada simultaneamente por todos os elementos de processamento sobre dados distintos.

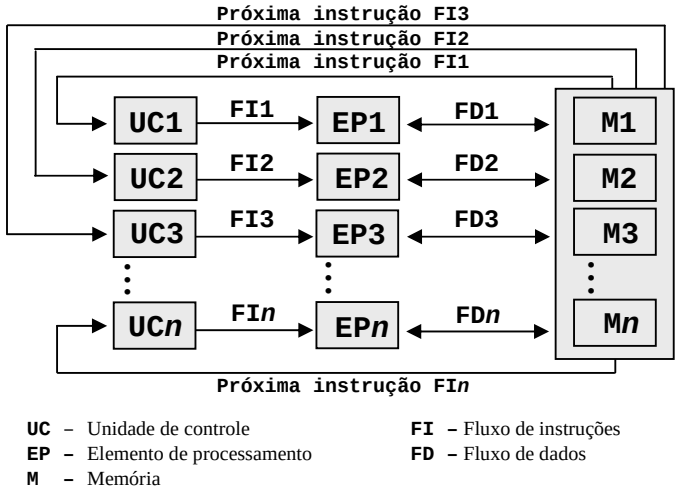


Figura 3.3. Arquitetura MIMD. Em arquiteturas MIMD cada processador executa um fluxo de instruções distinto de forma independente dos demais.

A classe MIMD pode ainda ser subdividida de acordo com o método de comunicação entre os processadores em: MIMD com memória compartilhada e MIMD com memória distribuída (DUNCAN, 1990; STALLINGS, 2002; TANENBAUM; GOODMAN, 1999).

(3.2.1.4.1) MIMD com memória compartilhada. Nas arquiteturas MIMD com memória compartilhada (Figura 3.4(a)) existe uma memória comum a todos os processadores, que então se comunicam (trocam dados) lendo e escrevendo nela. As arquiteturas MIMD com memória compartilhada mais comuns atualmente são os multiprocessadores simétricos (SMPs) e os sistemas com acesso não-uniforme à memória (NUMA). Em um SMP processadores e memória são interconectados por um barramento, ou algum outro tipo de circuito de interconexão interno, e o tempo de acesso a qualquer posição da memória é o mesmo, independente do processador que o está realizando (CULLER; SINGH; GUPTA, 1999; STALLINGS, 2002). Por este motivo, ele é dito um sistema UMA (*uniform memory access*), isto é, um sistema com acesso uniforme à memória. Em um sistema NUMA (CULLER; SINGH; GUPTA, 1999; QUINN, 1994), ao contrário, os tempos variam. Tipicamente, em um sistema NUMA os processadores são organizados em nós. Cada nó contém um ou mais processadores e uma quantidade de memória RAM. Apesar da memória estar fisicamente distribuída entre o nós, existe um único espaço de endereçamento. Porém, para qualquer processador o tempo de acesso a um endereço local é menor do que o tempo de acesso a um endereço em um nó remoto.

(3.2.1.4.2) MIMD com memória distribuída. Em arquiteturas MIMD com memória distribuída não existe compartilhamento de memória (TANENBAUM; GOODMAN, 1999). Conforme ilustrado na Figura 3.4(b), cada processador tem uma memória própria (privada). Os processadores se comunicam trocando/passando mensagens através de uma rede, que pode ser uma rede proprietária de alta velocidade ou uma rede comercial. Dentre os sistemas MIMD com memória distribuída, os agregados de computadores (ou *clusters* de computadores) são os que experimentam maior sucesso atualmente, por causa principalmente da sua relação custo/benefício. Um agregado consiste de uma coleção de computadores autônomos e de uma rede (por exemplo, uma rede *Fast Ethernet*), que interconecta estes computadores. A redução do preço dos computadores pessoais e o aumento na velocidade das redes de comunicação fizeram com que este tipo de sistema emergisse como uma boa opção de sistema para processamento paralelo.

Nas seções seguintes são apresentadas as arquiteturas dos computadores utilizados neste trabalho, isto é, dos multiprocessadores simétricos e dos agregados de computadores.

Também são apresentados os modelos e ferramentas de programação utilizados nestes tipos de arquiteturas.

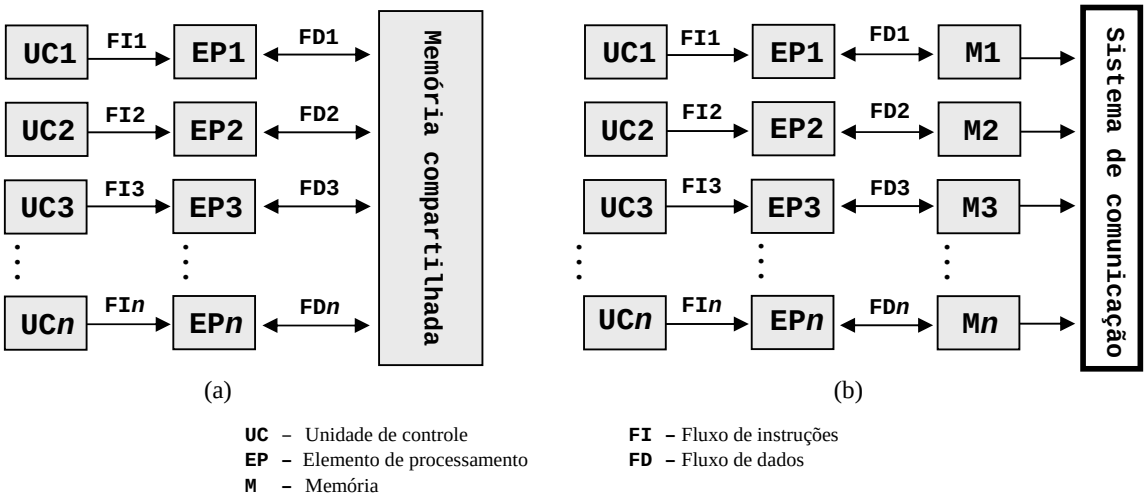


Figura 3.4. Arquiteturas MIMD com memória compartilhada e distribuída. Nas MIMD compartilhada existe uma memória comum (a) e nas MIMD distribuída cada processador tem uma memória privada (b).

3.2.2 Multiprocessadores simétricos

Um multiprocessador simétrico, ou SMP (do inglês *Symmetric MultiProcessor*), é um computador composto de dois ou mais processadores de propósito geral conectados entre si por um barramento ou alguma outra forma de circuito de interconexão interno (STALLINGS, 2002). Estes processadores compartilham a memória principal do sistema, assim como os dispositivos de E/S. SMPs apresentam:

- **Acesso uniforme à memória.** Em um SMP o tempo de acesso à memória é o mesmo, não importa o endereço do acesso ou o processador que o está realizando (CULLER; SINGH; GUPTA, 1999; PATTERSON; HENNESSY; GOLDBERG, 1996; STALLINGS, 2002).

- **Processadores funcionalmente idênticos.** Todos os processadores de um SMP são capazes realizar as mesmas operações, ou seja, não existem processadores dedicados à realização de tarefas específicas, como por exemplo, executar processos de sistema ou iniciar operações de E/S (ALMASI; GOTTLIEB, 1994; STALLINGS, 2002; TANENBAUM; GOODMAN, 1999).

3.2.2.1 Esquemas de interconexão

Os processadores de um SMP podem ser conectados à memória de diversas formas. As formas (esquemas de interconexão) mais comuns são: barramento de tempo compartilhado, redes *crossbar* e redes multiestágio.

(3.2.2.1.1) Barramento de tempo compartilhado. O barramento (Figura 3.5) consiste de um conjunto de linhas de metal em uma placa de circuito impresso através das quais os processadores, módulos de E/S e memória transmitem sinais de endereço, dados e controle (comandos e informações de temporização). O barramento é um mecanismo de tempo compartilhado, ou seja, em um dado instante apenas um dos dispositivos pode transmitir sinais através das suas linhas. Supondo que um processador deseja ler um dado da memória principal, o que ele faz primeiramente é verificar se o barramento não está sendo utilizado por outro dispositivo. Se o barramento está livre, ele simplesmente coloca o endereço do dado desejado nas linhas de endereço e através das linhas de controle ordena a leitura. Se, ao contrário, o barramento estiver sendo utilizado por qualquer outro dispositivo, por exemplo, um outro processador lendo/escrevendo na memória, ele tem que aguardar. Por isso o barramento é um ponto de gargalo que torna não viável a construção de um SMP com muitos processadores, uma vez que em grande parte do tempo muitos deles estariam ociosos aguardando pelo seu uso. Segundo Tanenbaum e Goodman (1999) a organização por barramento limita o número de processadores em 16 ou 32. Mesmo este número só é possível quando os processadores são equipados com um ou mais níveis de memória *cache*.

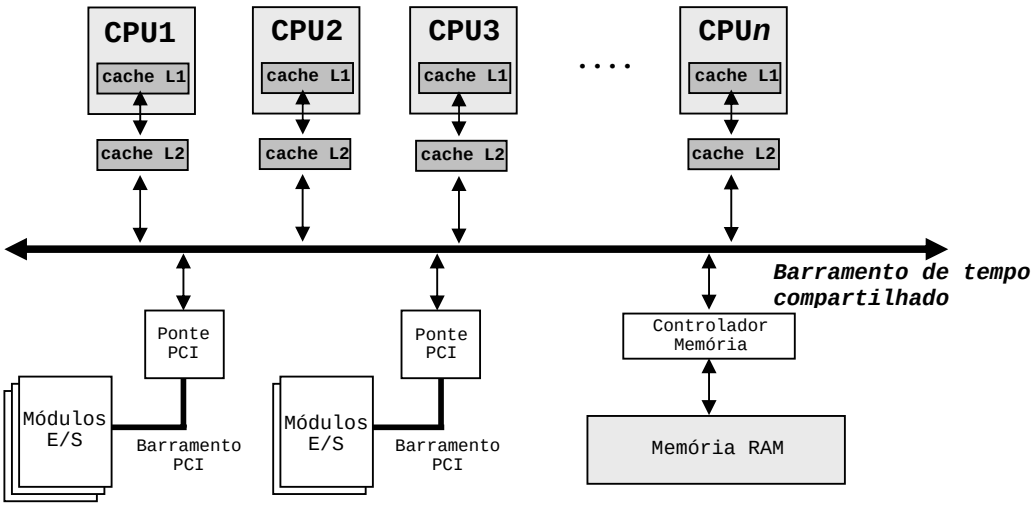


Figura 3.5. SMP com barramento de tempo compartilhado. O barramento é o esquema de interconexão mais comum, porque é o mais simples/barato, sendo muito parecido com aquele encontrado nos computadores pessoais.

(3.2.2.1.2) *Crossbar*. A construção de um SMP com um número maior de processadores é possível com uma rede *crossbar*. Enquanto que em um SMP com barramento de tempo compartilhado existe um único caminho (conjunto de linhas) ligando os processadores à memória, em uma rede *crossbar* existem vários. Em geral a memória é organizada em vários

módulos e existe um caminho distinto para cada módulo. A Figura 3.6 ilustra uma rede *crossbar* 8 × 8, que conecta 8 processadores a 8 módulos de memória. Esta configuração permite até 8 acessos simultâneos à memória, um para cada módulo – cada módulo pode responder a um processador por vez, mas todos podem responder simultaneamente a processadores diferentes. Em cada ponto de intersecção da rede *crossbar* existe um *crosspoint*, um *switch* que é eletronicamente aberto e fechado para permitir (fechado), ou não (aberto), a conexão dedicada ponto a ponto entre um par processador-memória. Segundo Hwang (1993) a complexidade de um único *crosspoint* é quase a mesma de um barramento. Como em uma *crossbar* $n \times n$ existem n^2 *crosspoints*, o custo de uma rede deste tipo é muito alto. Por isso, apesar do número de processadores poder ser muito grande, na prática, por questões financeiras, ele acaba não sendo muito maior que o de um SMP com barramento.

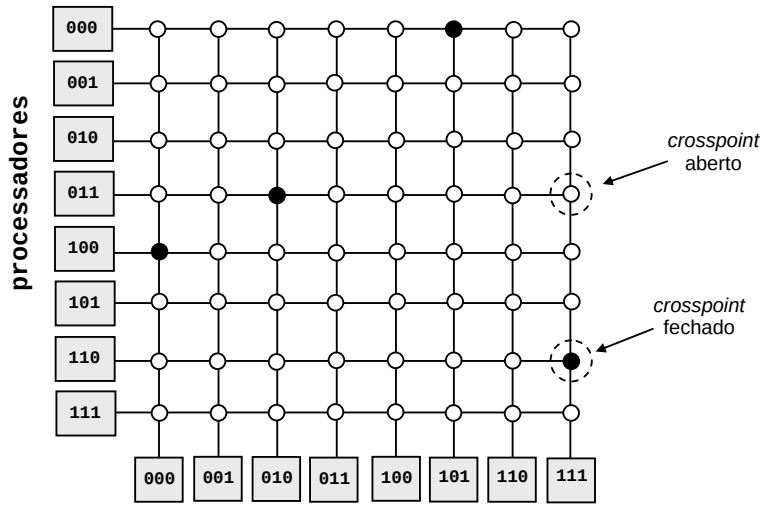


Figura 3.6. Rede *crosbbar* 8 × 8. Na rede da figura até oito acessos simultâneos podem ser feitos, um para cada módulo de memória. Ainda na figura, quatro acessos em paralelo estão sendo realizados entre os processadores-memória: 000-101, 011-010, 100-000, 110-111.

(3.2.2.1.3) Redes multiestágio. Redes multiestágio são construídas a partir de *switches* $A \times B$ (ou seja, *switches* com A entradas e B saídas). Diferentes tipos/classes de redes multiestágio podem ser construídas e o que varia de uma classe para a outra é o tipo de *switch* utilizado e o *padrão de conexão* entre os estágios. Tanenbaum e Goodman (1999) descrevem o funcionamento de uma rede *omega* (Figura 3.7) em um sistema com 8 processadores e 8 módulos de memória. Esta rede utiliza *switches* 2×2 e um padrão de conexão conhecido como *perfect shuffle* (ALMASI; GOTTLIEB, 1994; TANENBAUM; GOODMAN, 1999). Nesta rede as leituras/escritas são baseadas em mensagens compostas de quatro campos: (i) *módulo*: indica o módulo de memória desejado; (ii) *endereço*: indica um endereço dentro do módulo; (iii) *código de operação*: indica o tipo de operação (leitura ou escrita); e (iv) *valor*: é um

campo opcional que contêm um operando usado em operações de escrita. Supondo que o processador 010 deseja ler um dado da memória 110 ele envia uma mensagem de leitura (campo *operação* = leitura) para a *switch* 1C contendo o valor 110 no campo *módulo*. Este *switch* verifica o primeiro *bit* do campo módulo (um *switch* no primeiro estágio da rede verifica apenas o primeiro *bit* do campo *módulo*), que neste caso é 1. Um valor 1 indica que a mensagem deve prosseguir pela saída inferior do *switch* , neste caso indo para o *switch* 2B. O *switch* 2B, no estágio 2, analisa o segundo *bit* do campo *módulo*, que também é 1, e transmite a mensagem para o *switch* 3D. Este *switch* verifica o terceiro *bit* do campo *módulo* que é 0 (zero). Um valor 0 indica que a mensagem deve prosseguir pela saída superior. Então, neste exemplo, a mensagem chega até o seu destino, o módulo de memória 110. O caminho de transmissão desta mensagem é destacado com serrilhados na Figura 3.7. A construção de uma rede deste tipo é menos custosa que uma *crossbar*. Em um sistema com 8 processadores e 8 módulos de memória são utilizados apenas 12 *switches*, contra 64 na *crossbar* – para uma rede $n \times n$ utiliza-se $log_2\ n$ estágios e $n/2$ *switches* por estágio, num total de $(log_2\ n) * n/2$ *switches*. A desvantagem de uma rede multiestágio em relação à *crossbar*, é que nem sempre dois acessos podem ocorrer em paralelo, mesmo que estes sejam para módulos diferentes. Suponha que no sistema da Figura 3.7 o processador 111 deseja ler o módulo de memória 111 enquanto o processador 010 deseja ler o módulo 110. Embora os módulos acessados sejam diferentes, as mensagens de ambos os processadores passam pela *switch* 3D, onde pode ocorrer um conflito e uma delas ter que esperar.

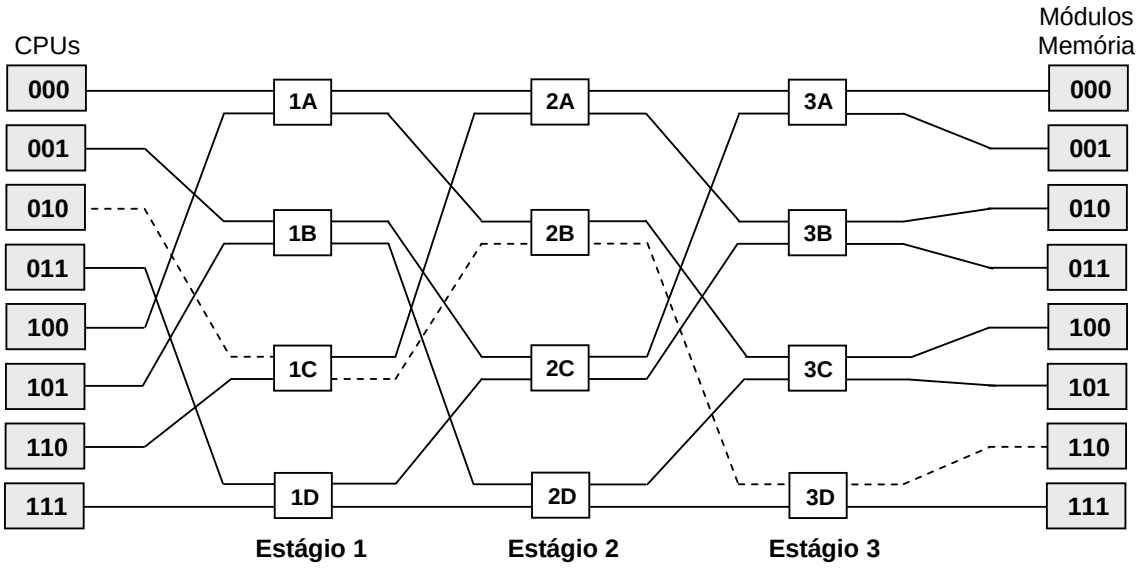


Figura 3.7. Rede multiestágio. Rede de 3 estágios com *switches* 2×2 (12 no total).

3.2.2.2 Execução de programas

Os processadores de um SMP operam sobre o controle de um único sistema operacional que escalona os processos (ou *threads*) de usuário e de sistema entre os processadores disponíveis, ou seja, qualquer processo pode ser executado em qualquer processador. Tanenbaum (2001) relaciona três métodos de escalonamento possíveis para SMPs, (i) *timesharing*, (ii) *space sharing* e (iii) *gang scheduling*, sendo o primeiro o mais comum deles – os sistemas operacionais Windows e Linux, em suas versões para multiprocessadores, utilizam este método de escalonamento, ou alguma variante dele.

No método *timesharing* cada processo executa em um dos processadores por um intervalo de tempo (*quantum*), usualmente 20 ou 30 ms, ou até realizar uma operação bloqueante (por exemplo, uma operação de E/S). Neste momento o processador onde ele executava passa a executar o escalonador de processos do sistema operacional, que verifica a fila de processos prontos e escolhe para ser executado aquele com maior prioridade. Este método de escalonamento é ilustrado na Figura 3.8 (TANENBAUM, 2001).

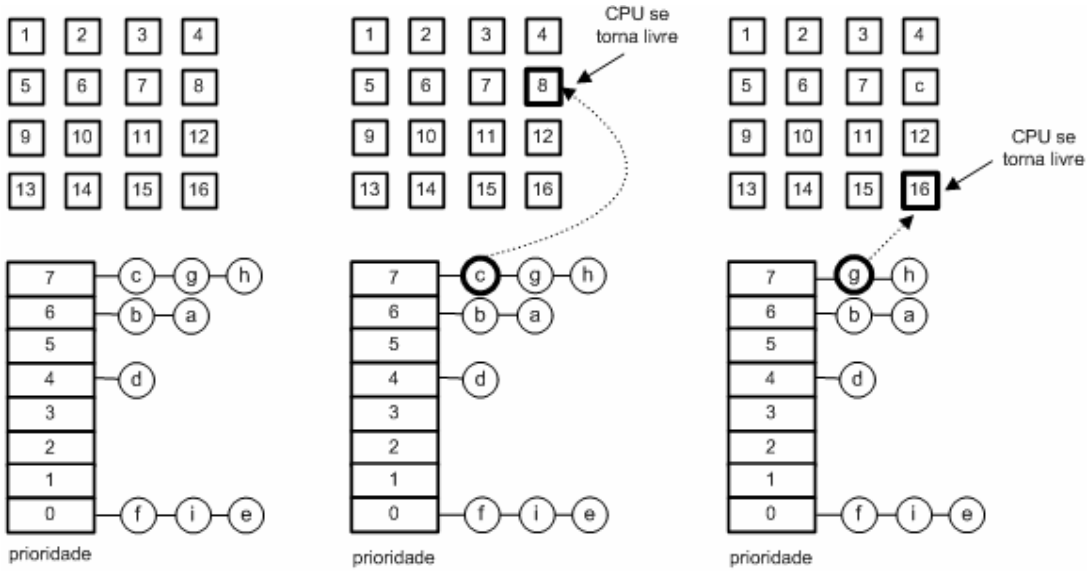


Figura 3.8. Escalonamento de processos segundo o método *timesharing*.

A figura mostra um SMP com 16 processadores com um sistema operacional que trabalha com 8 níveis de prioridade (0 a 7). Para cada processo criado é dada uma prioridade. Como no SMP da figura o número de processos é maior que o número de processadores, os processos que não estão em execução são colocados na fila de processos prontos, ou seja, na fila de processos que estão aguardando para serem executados, sendo que existe uma fila para cada nível de prioridade. Em (a) todos os processadores estão ocupados, cada um executando

um processo diferente, sendo que existem 9 processos aguardando. Quando um processador se tornar livre (ou seja, quando o processo que ele está executando terminar ou quando se esgotar o *quantum* deste processo) outro processo será escolhido para executar neste processador. Em (b) o processador de número 8 se tornou livre e o escalonador escolheu o primeiro processo da fila de maior prioridade, neste caso o processo C. O processo que estava sendo executado tem seu estado salvo e é inserido na fila de processos prontos para, futuramente, ser executado neste ou em outro processador. Em (c) outro processador se tornou livre e o primeiro processo da fila de maior prioridade foi escolhido para executar.

Um programa composto de múltiplos processos (ou de múltiplas *threads*) pode, portanto, ter seu tempo de execução reduzido quando executando em um SMP, uma vez que estes processos podem ser escalonados para executar ao mesmo tempo em processadores diferentes.

3.2.2.3 O SMP utilizado neste trabalho

O SMP utilizado neste trabalho tem quatro processadores UltraSPARC III de 750 MHz e 4GB de memória RAM. Cada processador tem 8 MB de memória *cache*. O mecanismo de interconexão é um barramento e o sistema operacional é o Solaris 8.

3.2.3 Agregados de computadores

Um agregado de computadores é uma coleção de computadores completos, interconectados, trabalhando juntos como um recurso de computação unificado, que cria a ilusão de constituir uma única máquina. O termo ‘computador completo’ significa um sistema que pode operar por si próprio, independente do agregado. Na literatura, cada computador que compõe um agregado é chamado nó (STALLINGS, 2002).

A grande idéia por trás dos agregados é que alguém pode construir um equipamento com alto poder computacional conectando diversos computadores (computadores pessoais, estações de trabalho ou mesmo SMPs). A boa relação custo/benefício deste tipo de arquitetura é talvez a maior responsável pela sua grande aceitação, tanto no meio acadêmico, como na indústria. Por exemplo, um agregado construído com uma dúzia de computadores pessoais, que executa um certo programa de simulação numérica em um dia, será provavelmente mais barato que um único computador paralelo que executa este mesmo programa no mesmo tempo (PASIN; KREUTZ, 2003).

São características dos agregados (PASIN; KREUTZ, 2003; STALLINGS, 2002):

- **Independência.** Cada nó de um agregado é um computador completo, com processador(es), memória e disposistivos de E/S. Cada nó executa um sistema operacional que não precisa necessariamente ser o mesmo dos outros nós. Um nó pode ser removido do agregado sem que isto afete o funcionamento dos outros.

- **Imagem única do sistema.** Geralmente, os nós de um agregado são programados de forma que o usuário tenha a impressão de estar usando um recurso computacional único. Isto é possível devido a camadas de software, dentro ou fora do sistema operacional.

- **Conexão especializada.** Os nós de um agregado são usualmente interconectados por algum tipo de rede rápida de tecnologia aberta, por exemplo, *Fast Ethernet* e *Gigabit Ethernet*.

3.2.3.1 Execução de programas

Diferente do que ocorre em um SMP, onde todos os processos executam em uma única máquina sob o controle de um único sistema operacional, em um agregado os processos executam em nós independentes, que executam o seu próprio sistema operacional. Para ter seu tempo de execução reduzido, um programa composto de múltiplos processos deve ter estes processos distribuídos entre estes nós. A distribuição pode ser feita pelo sistema operacional, ou por uma biblioteca de troca de mensagens. São exemplos de sistema operacional e biblioteca de troca de mensagens, respectivamente, o openMosix (http://openmosix.sourceforge.net) e a MPICH[13] (http://www-unix.mcs.anl.gov/mpi/mpich/).

3.2.3.2 O agregado utilizado neste trabalho

Neste trabalho foi utilizado o agregado de computadores do Grupo de Visão Cibernética do Instituto de Física de São Carlos (IFSC) da Universidade de São Paulo (USP). O agregado é composto de dez máquinas, sendo que cada máquina apresenta a seguinte configuração: um processador Pentium 4 de 2.8 GHz com 512 KB de memória *cache*, e 1.5 GB de memória RAM; o sistema operacional é o openMosix, kernel 2.4.6. As máquinas do agregado estão interligadas por um *switch* Fast Ethernet.

[13] MPICH é uma implementação gratuita do padrão MPI (*Message Passing Interface*) (http://www.mpi-forum.org/).

3.3 Programação paralela

Resolver um problema mais rapidamente utilizando para isto os múltiplos processadores de um SMP, ou de um agregado, requer a divisão/decomposição deste problema em partes e a distribuição destas partes entre estes processadores para execução simultânea. Na prática, isso significa a construção de um programa (*programa paralelo*) composto de múltiplos processos (ou de múltiplas *threads*), onde cada processo (ou *thread*), executando em um processador diferente, resolve uma parte do problema, para que o problema na sua totalidade seja resolvido mais rapidamente.

3.3.1 Programação paralela em SMPs

O desenvolvimento de programas paralelos para SMPs é realizado através de ferramentas que utilizam um modelo de programação paralela chamado *modelo de memória compartilhada*. Programas baseados neste modelo criam múltiplos processos ou, mais freqüentemente, múltiplas *threads* dentro de um único processo, que cooperam (trocam dados) via endereços de memória compartilhados – a construção de programas compostos de um único processo com múltiplas *threads*, ao invés de múltiplos processos, é mais comum atualmente porque têm entre outras vantagens: (i) o tempo gasto na criação de uma *thread* é da ordem de milhares de vezes menor que o de um processo; e (ii) o custo de manutenção/controle de uma *thread*, para o sistema operacional, é menor que o de um processo.

Ferramentas para programação *multithread*[14] diferem quanto ao nível de abstração ou, em outras palavras, quanto à forma de criação, destruição e sincronização das *threads*. De um lado estão ferramentas como a POSIX Threads (http://www.opengroup.org/austin/papers/backgrounder.html), onde o programador tem que, explicitamente, criar e destruir as *threads*, assim como controlar todo e qualquer tipo de sincronização. Do outro lado estão as ferramentas baseadas em diretivas de compilação onde o programador utiliza comentários especiais (diretivas) para "marcar" as partes do programa que devem ser paralelizadas. Então, a partir destas diretivas, um compilador gera automaticamente o código paralelo. A programação *multithread* baseada em diretivas, através da ferramenta OpenMP (http://www.openmp.org), tem se tornado padrão em arquiteturas com memória compartilhada.

[14] Programação com múltiplas *threads*.

3.3.2 Programação paralela em agregados de computadores

Em arquiteturas MIMD com memória distribuída, especialmente em agregados, o modelo de programação paralela mais utilizado é o *modelo de passagem de mensagens*. Programas baseados neste modelo criam múltiplos processos que não compartilham memória e que, por isso, trocam dados entre si através do envio (SEND) e recepção (RECEIVE) de mensagens. Em geral o programador desenvolve o seu programa em uma linguagem seqüencial, como C/C++ e Fortran, com chamadas para uma biblioteca com funções de envio e recepção de mensagens, como por exemplo as bibliotecas MPI (*Message Passing Interface*) e PVM (*Parallel Virtual Machine*).

3.4 Ferramentas para programação paralela

Atualmente, as ferramentas mais utilizadas para a programação paralela em SMPs e agregados são, respectivamente, OpenMP e MPI. Estas foram as ferramentas utilizadas para a implementação dos algoritmos paralelos desenvolvidos neste trabalho, sendo descritas a seguir.

3.4.1 OpenMP

O OpenMP surgiu em 1997 e atualmente está na versão 2.0. O OpenMP propriamente dito é apenas um padrão que especifica um conjunto de diretivas de compilação, funções e variáveis de ambiente para a programação paralela em arquiteturas com memória compartilhada com as linguagens Fortran 77, Fortran 90, C e C++. Tal padrão é disponibilizado para o programador por meio de compiladores. Ao compilar um programa Fortran ou C/C++ contendo diretivas OpenMP com um compilador OpenMP (ou seja, um compilador que implementa a especificação OpenMP), o programador obtém um programa paralelo.

3.4.1.1 Modelo de execução

Um programa OpenMP inicia com uma única *thread*, a *thread* mestre, que executa até encontrar uma região paralela[15]. Neste ponto ela cria outras *threads*. Então todas as *threads*, inclusive a mestre, executam o código dentro desta região. Quando as *threads* completam a execução do código na região paralela elas se sincronizam e, com exceção da *thread* mestre,

[15] Parte do código que deve ser executada por múltiplas *threads*, ou seja, parte de código onde o paralelismo acontece.

terminam. A execução então continua seqüencialmente com a *thread* mestre até que ela encontre uma nova região paralela ou até que o programa termine. Este modelo de execução é conhecido como *fork-join* (Figura 3.9).

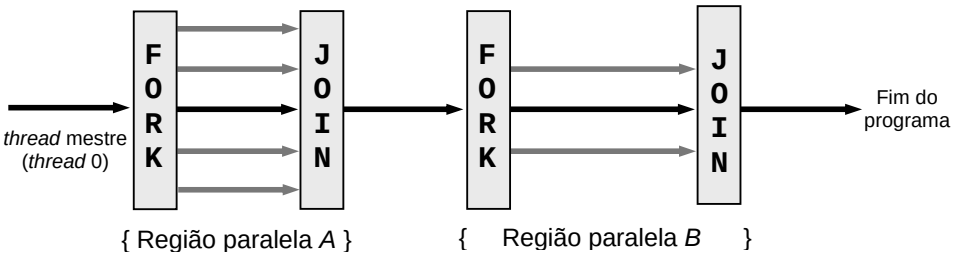


Figura 3.9. Modelo de execução OpenMP. Um programa com duas regiões paralelas, *A* e *B*. Para executar a primeira região são criadas outras quatro *threads*. Para executar a segunda são criadas duas.

3.4.1.2 Diretivas de paralelização

A criação das *threads* em um programa OpenMP é responsabilidade do compilador. O programador precisa apenas definir, através de diretivas[16], as regiões paralelas. O OpenMP oferece duas diretivas para definição de regiões paralelas, uma delas é utilizada para a definição de regiões contendo laços *for*. Portanto, para pararelizar um laço *for* define-se uma região paralela e, dentro dela, o laço. A outra diretiva é utilizada para a paralelização de qualquer outro tipo de código (NCSA, 2003a). Estas diretivas são apresentadas a seguir.

(3.4.1.2.1) Diretiva *parallel for*. Esta diretiva é utilizada para a paralelização de laços *for*, conforme ilustrado no código seguinte:

Paralelização de laços – diretiva parallel for

```
.
.   // parte seqüencial do programa //
.
.
56. #pragma omp parallel for
57. {
58.     for(i = 1; i <= n; i++)
59.        a[i] = a[i] * b[i - 1];
60. }
```

As chaves nas linhas 57 e 60 definem/delimitam uma região paralela.

No código anterior, quando a *thread* mestre encontra a diretiva **`#pragma omp parallel for`** ela cria *t threads* para executar as *n* iterações do laço "dentro" dela. Por

[16] Uma diretiva consiste de uma linha de código com significado especial para o compilador. Uma diretiva OpenMP, na linguagem C/C++, é identificada pela sentinela **`#pragma omp`**

padrão, cada *thread* executará *n/t* iterações. Neste código por exemplo, se $n = 101$ e $t = 2$, a primeira *thread* executará as iterações de $i = 1, 2, ..., 51$ e a segunda as de $i = 52, ..., 101$.

(3.4.1.2.2) Diretiva *parallel*. Esta diretiva é utilizada para paralelizar qualquer parte de um programa. Seu uso é ilustrado no programa/código seguinte. Tal programa faz a soma dos elementos de um vetor *v*. Para isso, este vetor é "dividido" em partes de igual tamanho, uma para cada *thread*.

Paralelização de regiões – diretiva parallel

```
01. #include <omp.h>
    .
    .
    .
18. qtde_elem   = tamanho_vetor(v) / num_threads;
19.
20. omp_set_num_threads (num_threads);
21.
22. #pragma omp parallel private(inicio, fim, meu_rank)
23. {
24.     meu_rank = omp_get_thread_num();
25.     inicio   = qtde_elem * meu_rank;
26.     fim      = inicio + qtde_elem;
27.
28.     for (i = inicio; i < fim; i++)
29.         soma_parc[meu_rank] = soma_parc[meu_rank] + v[i];
30. }
31.
32. for (i = 0; i < num_threads; i++)
33.     soma_total = soma_total + soma_parc[i];
34.
35. printf("Resultado da soma: %d", soma_total);
36. .
37. .
```

No código anterior, a diretiva `parallel` (linha 22) define uma região paralela (linha 24 a 29). O número de *threads* que executam esta região é definido na linha 20, com a função `omp_set_num_threads`, como sendo igual a `num_threads`. A quantidade de elementos de `v` que cada *thread* irá somar é definida na linha 18 em `qtde_elem` (para simplificar o código, está-se assumindo que o número de elementos do vetor sempre é múltiplo de `num_threads`). O primeiro e o último elemento, ou seja, o pedaço do vetor atribuído a cada *thread*, é calculado com base no seu *rank*[17], que é obtido através da função `omp_get_thread_num`, como pode ser visto nas linhas 24, 25 e 26. A soma dos elementos é efetivamente realizada na linha 29 em um laço *for* que é executado por todas as *threads*. Dentro deste laço utiliza-se um vetor, `soma_parc`, onde cada *thread* armazena, na posição correspondente ao seu *rank*, o

[17] Cada *thread* dentro de uma região paralela é identificado por um *rank*. Em uma região com *t threads* os *ranks* variam de 0 a $t - 1$ (o *rank* 0 sempre identifica a *thread* mestre).

resultado da soma do seu pedaço do vetor. No fim da região paralela (linha 30) todas as *threads*, com exceção da mestre, são destruídas. Então, a mestre computa o resultado final da soma (linha 32) utilizando os valores em `soma_parc`.

Diferente da diretiva `parallel for` que divide automaticamente as iterações de um laço *for* entre todas as *threads*, um laço *for* dentro de uma região definida com a diretiva `parallel` é executado inteiramente por todas as *threads*. Neste exemplo utilizou-se as variáveis `meu_rank`, `inicio` e `fim` para definir diferentes "pedaços" do vetor para diferentes *threads*. Embora mais complexa, a diretiva `parallel` pode ser utilizada para paralelizações de qualquer natureza. Através do *rank* das *threads* poder-se-ia, por exemplo, designar tarefas diferentes para cada *thread* (por exemplo, `if meu_rank = 0` executar tarefa *x*, `if meu_rank = 1` executar tarefa *y*, etc.).

3.4.1.3 Cláusulas

Cláusulas são palavras chave utilizadas junto com as diretivas geralmente para modificar o comportamento padrão destas. O OpenMP especifica várias cláusulas. Algumas delas são descritas a seguir.

(3.4.1.3.1) Cláusula *private*. Todas as variáveis dentro de uma região paralela são, por padrão, compartilhadas entre todas as *threads*. Isto nem sempre é desejado, porque pode levar a condições de corrida[18], como no código seguinte (esquerda):

Paralelização de laços – cláusula private

```
//Código com condição de corrida
01. #pragma omp parallel for
02. {
03.   for(i = 1; i <= n; i++){
04.      aux  = 7 * a[i];
05.      a[i] = aux;
06.   }
07. }
```

```
//Código sem condição de corrida
01. #pragma omp parallel for private (aux)
02. {
03.    for(i = 1; i <= n; i++){
04.       aux  = 7 * a[i];
05.       a[i] = aux;
06.    }
07. }
```

No código da esquerda, a variável `aux` é acessada e alterada por todas as *threads*. Se a primeira *thread* executa `aux = 7 * a[1]`, por exemplo, e em paralelo a segunda executa `aux = 7 * a[52]`, quando a primeira *thread* executar a linha 5 do laço, o valor de `aux`, que deveria ser `7 * a[1]`, pode ser `7 * a[52]`. Este tipo de problema pode ser resolvido com o uso da cláusula **`private`**. Esta cláusula, no código da direita, especifica que deve ser alocada

[18] Situação onde o valor de uma variável depende da ordem de execução das *threads.*

memória para a variável `aux` em cada *thread* (ou seja, deve existir uma variável `aux` privada em cada *thread*).

(3.4.1.3.2) Cláusula *default*. Se o número de variáveis privadas dentro de uma região paralela é muito grande, ao invés de identificar cada uma delas na cláusula `private`, é mais simples utilizar a cláusula `default(private)`. Neste caso todas as variáveis da região paralela serão consideradas privadas.

(3.4.1.3.3) Cláusula *shared*. No caso da cláusula `default(private)` ter sido utilizada, qualquer variável compartilhada entre as *threads* deve ser identificada com a cláusula `shared`.

(3.4.1.3.4) Cláusula *firstprivate*. A cláusula `private` aloca espaço para uma variável em cada *thread* sem inicializá-la (a inicialização deve ocorrer dentro do laço). Existem situações, no entanto, onde pode ser necessária a inicialização de uma variável, em cada *thread*, com o valor que ela tinha na região serial[19]. Isto é feito com a cláusula `firstprivate`.

(3.4.1.3.5) Cláusula *lastprivate*. De forma semelhante, pode-se desejar manter o valor de uma variável privada após a finalização de uma região paralela. Neste caso, identifica-se esta variável com a cláusula `lastprivate`.

3.4.1.4 Funções

O OpenMP, além de diretivas, especifica funções. Algumas das funções mais utilizadas são descritas a seguir.

(3.4.1.4.1) `omp_set_num_threads`. Define o número de *threads* a ser utilizado nas regiões paralelas subseqüentes. Pode ser utilizada várias vezes no programa para especificar diferentes números de *threads* para diferentes regiões paralelas e deve ser chamada fora da região paralela.

(3.4.1.4.2) `omp_get_num_threads`. Retorna o número de *threads* da região paralela onde esta função foi chamada.

[19] Região que antecede uma região paralela, ou seja, a região que é executada apenas pela *thread* mestre.

(3.4.1.4.3) **`omp_get_thread_num`**. Cada *thread* dentro de uma região paralela é identificada por um *rank* que pode ser descoberto através da função `omp_get_thread_num`. Em uma região com *t threads* os *ranks* variam de 0 a *t* – 1 (o *rank* 0 sempre identifica a *thread* mestre).

(3.4.1.4.4) **`omp_get_num_procs`**. Retorna o número de processadores disponíveis.

3.4.1.5 Diretivas de sincronização

Sincronização é toda e qualquer atividade realizada para (i) impor uma seqüência de execução às *threads* de um programa e/ou (ii) evitar o uso simultâneo, por duas ou mais *threads*, de um recurso compartilhado. O OpenMP oferece diretivas que tornam a sincronização bastante simples.

(3.4.1.5.1) Diretiva *critical*. Especifica uma região, dentro de uma região paralela, que deve ser executada por uma única *thread* por vez (ou seja, é uma diretiva utilizada para prover exclusão mútua na execução de determinadas partes do código):

```
#pragma omp parallel
{
    .
    .
    #pragma omp critical
    {
        código executado por todas as threads, mas por apenas uma delas de cada vez.
    }
}
```

(3.4.1.5.2) Diretiva *barrier*. Esta diretiva sincroniza todas as *threads* de uma região paralela. Cada *thread* ao encontrar a diretiva `#pragma omp barrier` é parada/bloqueada. Quando a última *thread* encontra a barreira, todas as *threads* "prosseguem" (são desbloqueadas).

(3.4.1.5.3) Diretiva *single*. Especifica uma região, dentro de uma região paralela, que deve ser executada por apenas uma *thread*. A primeira *thread* a encontrar a diretiva `single` é a que fará isso.

(3.4.1.5.4) Diretiva *master*. Especifica uma região, dentro de uma região paralela, que deve ser executada apenas pela *thread* mestre.

3.4.1.6 Variáveis de ambiente

O OpenMP provê quatro variáveis de ambiente (LLNL, 2003) que são descritas a seguir.

(3.4.1.6.1) `OMP_NUM_THREADS`. Define o número de *threads* utilizadas nas regiões paralelas (a função `omp_set_num_threads` tem precedência sobre esta variável; portanto, o valor especificado na variável `OMP_NUM_THREADS` só será utilizado se a função não for chamada).

(3.4.1.6.2) `OMP_SCHEDULE`. O valor desta variável especifica como as iterações de um laço são escalonadas nos processadores.

(3.4.1.6.3) `OMP_DYNAMIC`. Habilita ou desabilita o ajuste dinâmico do número de *threads* utilizadas em regiões paralelas.

(3.4.1.6.4) `OMP_NESTED`. Habilita ou desabilita a possibilidade de paralelismo aninhado.

3.4.1.7 O compilador OpenMP utilizado neste trabalho

Neste trabalho foi utilizado o compilador do pacote Sun Workshop 6 update 2.

3.4.2 Message Passing Interface

O MPI (MPIF, 1995; MPIF, 1997), assim como o OpenMP, é apenas um padrão. Este padrão especifica um conjunto de funções que uma biblioteca de passagem de mensagens deve oferecer. Sob responsabilidade do *Message Passing Interface Forum* (http://www.mpi-forum.org), a primeira versão da especificação, MPI versão 1.0, foi publicada em junho de 1994. Em 1995 foi publicada uma revisão, MPI-1.1, e em 1997 a especificação MPI-2, que incluiu correções e novas funcionalidades, como criação dinâmica de processos, E/S paralela e comunicação unilateral. Atualmente, diversas implementações MPI existem, algumas pagas, outras gratuitas.

Utilizando uma biblioteca de funções que implementa o padrão MPI “dentro” de uma linguagem seqüencial como C, C++, Fortran 77 ou Fortran 90, o programador constrói o seu programa paralelo utilizando o modelo de passagem de mensagens.

3.4.2.1 Modelo de execução

Um programa MPI cria um conjunto de processos que executam, ou não, um mesmo código e trocam dados utilizando as rotinas de comunicação MPI – o MPI não especifica o modelo de execução dos processos, que pode ser seqüencial ou *multithread* (MPIF, 1995). Estes processos "existem" dentro dos chamados comunicadores. Dois processos podem trocar mensagens se, e somente se, eles pertencem ao mesmo comunicador. Todos os processos de um programa MPI são automaticamente associados a um comunicador padrão, MPI_COMM_WORLD, mas o programador pode criar outros. Cada processo em um comunicador recebe uma identificação única chamada *rank*[20], que é utilizada para especificar a origem e o destino das mensagens – no caso do processo pertencer a mais de um comunicador, ele recebe uma identificação/*rank* diferente em cada um deles.

3.4.2.2 Comunicação ponto-a-ponto

Comunicação ponto-a-ponto é aquela onde existem apenas dois processos envolvidos, um enviando (*transmissor*) e outro recebendo (*receptor*) uma mensagem. A transmissão da mensagem requer a participação dos dois processos, que devem explicitamente realizar um SEND (transmissor) e um RECEIVE (receptor). O MPI oferece uma grande quantidade de rotinas de comunicação ponto-a-ponto, que diferem em pelo menos dois aspectos: (i) modo de comunicação e (i) bloqueio/não-bloqueio. Para cada modo de comunicação e bloqueio/não-bloqueio existe uma função.

(3.4.2.2.1) Modos de comunicação. O modo de comunicação define o procedimento de transmissão de uma mensagem e o critério que determina quando esta transmissão é considerada completada (NCSA, 2003b). Para envio (SEND) existem quatro modos (ou seja, quatro funções): (i) padrão, (ii) síncrono, (iii) *bufferizado* e (iv) pronto. Para recepção (RECEIVE) existe apenas um. Com qualquer modo envio, quando o SEND completa significa que o *buffer de transmissão* (posições de memória do processo transmissor onde estão os dados que estão sendo transmitidos) pode ser reutilizado com segurança. No seu único modo, quando o RECEIVE completa significa que o *buffer de recepção* (posições de memória do processo receptor onde são armazenados os dados recebidos), já tem os dados e eles estão disponíveis para uso.

[20] Em um comunicador com p processos os *ranks* variam de 0 a $p - 1$

Os quatro modos de envio são descritos a seguir (CENAPAD, 2003; NCSA, 2003b; SOUZA, 1996; SOUZA, 1997):

- **SEND padrão:** no modo padrão, o SEND completa quando uma de duas coisas aconteceu: (i) a mensagem foi copiada para um *buffer* interno do MPI (no transmissor) para ser enviada posteriormente, em *background*, pelo MPI; neste caso, a chamada à função SEND pode até mesmo retornar antes da realização do RECEIVE; ou (ii) o processo receptor realizou o RECEIVE correspondente e começou a receber a mensagem. O MPI não especifica quando utilizar um ou outro (*bufferizado* ou síncrono). Esta é uma decisão das implementações MPI, que pode inclusive variar a cada mensagem. Em geral, o que as implementações fazem é utilizar o primeiro método quando a mensagem cabe no seu *buffer* interno e o segundo em caso contrário. A função utilizada para um SEND do tipo padrão é:

```
int MPI_Send (void *buffer_t, int cont, MPI_Datatype tipo, int rank,
              int tag, MPI_Comm com);
```

O parâmetro `buffer_t` é o endereço do dados que esta sendo enviado; `cont` indica o número de elementos a ser enviado (este parâmetro é útil quando se quer transmitir vários elementos que estão alinhados na memória, de uma só vez, como por exemplo, um vetor); `tipo` corresponde ao tipo do dado (pode ser um dos tipos definidos pelo MPI, conforme Tabela 3.1, ou um tipo de dado definido pelo usuário. Além dos dados, a função requer a especificação do `rank` do processo de destino (receptor), do comunicador (`com`) a ser utilizado e de um `tag`, que é utilizado para distinguir mensagens de um mesmo transmissor.

TABELA 3.1. Tipos de dados MPI. Ao especificar um dos tipos de dados do MPI o usuário não tem que se preocupar com a representação destes dados em diferentes arquiteturas. Qualquer conversão necessária é feita automaticamente pelo MPI.

tipo MPI	correspondente em C
MPI_CHAR	char
MPI_SHORT	short
MPI_INT	int
MPI_LONG	long
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long doublé
MPI_BYTE	(nenhum)
MPI_PACKED	(nenhum)

- **SEND *bufferizado:*** Para um SEND *bufferizado* o programador define explicitamente um *buffer* para onde o MPI deve obrigatoriamente copiar os dados que estão sendo enviados. Após a cópia, o processo transmissor é liberado (ou seja, a chamada SEND retorna). O

real envio da mensagem acontece em *background*, a partir do *buffer* especificado, pelo MPI. No caso do tamanho do *buffer* ser insuficiente para acomodar a mensagem, o resultado é indefinido.

- **SEND síncrono:** uma operação SEND no modo síncrono pode ser iniciada independentemente da operação RECEIVE ter sido executada, mas só será completada quando o processo receptor executar o RECEIVE. Assim, um SEND síncrono além de indicar que o *buffer* de transmissão pode ser reutilizado, indica que o processo destino iniciou (mas não necessariamente terminou) o recebimento da mensagem.

- **SEND pronto:** No modo pronto, a mensagem é enviada imediatamente para a rede (a única espera ocorre durante a cópia do *buffer* de transmissão para a rede), mas o SEND só deve ser executado se o RECEIVE correspondente já tiver sido postado pelo receptor. Isto deve ser garantido pelo programador. A realização do SEND sem um prévio RECEIVE tem resultado indefinido.

As funções utilizadas para os SENDs *bufferizado*, síncrono e pronto são, respectivamente, `MPI_Bsend`, `MPI_Ssend`, e `MPI_Rsend`, e têm os mesmos parâmetros da função SEND padrão (`MPI_Send`).

A função de recepção (RECEIVE) contém basicamente os mesmos parâmetros das funções de envio e um parâmetro adicional:

```
int MPI_Recv (void *buffer_r, int cont, MPI_Datatype tipo, int rank,
                    int tag, MPI_Comm com, MPI_status status);
```

O parâmetro `buffer_r` é o endereço onde os dados recebidos devem ser armazenados; `cont` indica o número de elementos a ser recebido (o número de elementos efetivamente recebido pode ser menor); `tipo` corresponde ao tipo do dado (deve ser o mesmo que o especificado no SEND). O receptor deve especificar ainda o `rank` do processo transmissor, o comunicador (`com`) utilizado e o `tag` (que deve ser o mesmo que o especificado no SEND); `status` contém informações sobre a mensagem que foi recebida, como por exemplo, a quantidade de dados efetivamente recebida.

Em um processo *A* podem chegar mensagens provenientes de quaisquer outros processos, em qualquer ordem. Ao especificar, no RECEIVE, o *rank* do processo de origem e um *tag*, somente a mensagem que combinar com os valores especificados por estes dois parâmetros será recebida. Quaisquer outras mensagens serão ignoradas e armazenadas para um recebimento posterior através de um RECEIVE que combine com os seus parâmetros.

(3.4.2.2.2) Bloqueio (comunicação bloqueante e não-bloqueante). Rotinas de comunicação bloqueante garantem que o transmissor ou receptor ficará bloqueado até que o envio ou recepção da mensagem seja completado, ou seja, o código que sucede a chamada de uma rotina bloqueante só é executado quando ela tiver sido completada. Nas rotinas de comunicação não-bloqueante, por outro lado, ele é executado imediatamente após o inicio da comunicação, o que possibilita a sobreposição de computação e comunicação, podendo resultar em maior desempenho. O MPI oferece para todos os modos de envio e para o único modo de recepção existente, descritos na Seção 3.4.2.2.1, versões bloqueantes (que correspondem exatamente às funções que foram apresentadas) e não bloqueantes (Tabela 3.2).

TABELA 3.2. Rotinas de comunicação ponto-a-ponto bloqueantes e não bloqueantes do MPI

Tipo	Função bloqueante	Função não bloqueante
SEND padrão	MPI_Send	MPI_Isend
SEND síncrono	MPI_Ssend	MPI_Issend
SEND bufferizado	MPI_Bsend	MPI_Ibsend
SEND pronto	MPI_Rsend	MPI_Irsend
RECEIVE	MPI_Recv	MPI_Irecv

Todos as funções de envio não-bloqueantes têm os mesmos parâmetros das correspondentes bloqueantes e um parâmetro adicional `request`. No caso do RECEIVE, `request` vem no lugar do parâmetro `status` (`MPI_Status`). `request` é um ponteiro do tipo `MPI_Request` onde é retornado um *identificador de operação não-bloqueante*. Futuramente o programador pode utilizar tal identificador para verificar se a operação foi completada. Isto pode ser feito com uma das funções:

```
int MPI_Test (MPI_Request *request, int *flag, MPI_Status status);
int MPI_Wait (MPI_Request *request, MPI_Status status);
```

Ambas as funções tem como parâmetro de entrada a variável `request` que identifica um SEND ou RECEIVE não-bloqueante.

A função `MPI_Test`, apenas verifica se operação de comunicação identificada por `request` foi completada, retornando `flag <> 0` em caso afirmativo. Já a função `MPI_Wait` causa o bloqueio do processo até que a operação identificada por `request` seja completada.

3.4.2.3 Comunicação coletiva

Uma comunicação coletiva é aquela que envolve um grupo de processos. O grupo envolvido é especificado através de um comunicador. Todos os processos do comunicador

devem participar e, para isso, todos devem chamar a mesma função. Algumas destas funções são descritas a seguir.

(3.4.2.3.1) *Broadcast*. É uma comunicação do tipo *1-para-todos*, ou seja, os dados em um dos processos do comunicador especificado é enviado para todos os outros processos deste comunicador. O processo que contém os dados é chamado raiz do *broadcast*. O funcionamento do *broadcast* é ilustrado na Figura 3.10(a) e a função MPI utilizada para esta operação é:

```
int MPI_Bcast (void *buffer, int cont, MPI_Datatype tipo, int raiz_b,
               MPI_Comm com);
```

O parâmetro `buffer` é o endereço dos dados (no caso do processo raiz é o endereço dos dados que estão sendo transmitidos; para os outros processos é o endereço onde estes dados devem ser armazenados); `cont` indica o número de elementos a ser enviado/recebido; `tipo` corresponde ao tipo dos dados; `raiz_b` é o *rank* do processo raiz e deve ter o mesmo valor em todos os processos; `com` é o comunicador associado ao grupo de processos envolvido na comunicação.

(3.4.2.3.2) *Scatter* (ou distribuição). Enquanto que no *broadcast* todos os dados no *buffer* de transmissão do processo raiz são enviados para todos os outros, com *scatter* cada processo, incluindo o raiz, recebe uma porção diferente de igual tamanho deste *buffer*. Os dados são distribuídos na ordem de *rank* (o processo de *rank* 0 recebe a porção 0 do *buffer* e assim sucessivamente). O funcionamento desta operação é ilustrado na Figura 3.10(b) e a função utilizada para tal é:

```
int MPI_Scatter (void *sbuffer, int scont, MPI_Datatype stipo,
                 void *rbuffer, int rcont, MPI_Datatype rtipo,
                 int raiz_d, MPI_Comm com);
```

Os parâmetros `sbuffer`, `scont` e `stipo` especificam, respectivamente, o *buffer* de transmissão, a quantidade e o tipo dos dados a serem enviados, tendo validade apenas no processo raiz; `scont` especifica a quantidade de elementos a ser enviada para cada processo (por exemplo, se `sbuffer` é um vetor de 8 elementos e existem 4 processos, `scont` deve ser igual a 2); `rbuffer`, `rcont` e `rtipo` especificam, respectivamente, o *buffer* de recepção, a quantidade e o tipo dos dados a serem recebidos, tendo validade em todos os processos, incluindo o raiz; `raiz_d` especifica o processo raiz da distribuição e deve ter o mesmo valor em todos os processos.

(3.4.2.3.3) *Gather* (ou coleta). Comunicação do tipo *todos-para-1*; os dados no *buffer* de transmissão de cada um dos processos de um comunicador são enviados para um único processo chamado raiz da coleta, que armazena estes dados no seu *buffer* de recepção, em ordem de *rank.* O funcionamento de uma coleta é ilustrado na Figura 3.10(c) e a função utilizada para tal é:

```
int MPI_Gather (void *sbuffer, int scont, MPI_Datatype stipo,
                void *rbuffer, int rcont, MPI_Datatype rtipo,
                int raiz_c, MPI_Comm com);
```

Os parâmetros `sbuffer`, `scont` e `stipo` especificam o endereço, quantidade e tipo dos dados que estão sendo enviados, tendo validade em todos os processos, incluindo o raiz; `rbuffer`, `rcont` e `rtipo` especificam o *buffer* de recepção, quantidade e tipos de elementos a serem recebidos, tendo validade apenas no processo raiz (`rbuffer`, deve ser suficientemente grande para armazenar os dados de todos os processos); `rcont` especifica a quantidade de elementos do tipo `rtipo` a ser recebida de cada processo; `raiz_c` especifica o processo raiz da coleta e deve ter o mesmo valor em todos os processos.

Uma outra função, **`MPI_Allgather`**, é muito parecida com `MPI_Gather`, porém, os dados são coletados por todos os processos, conforme ilustrado na figura 3.10(d).

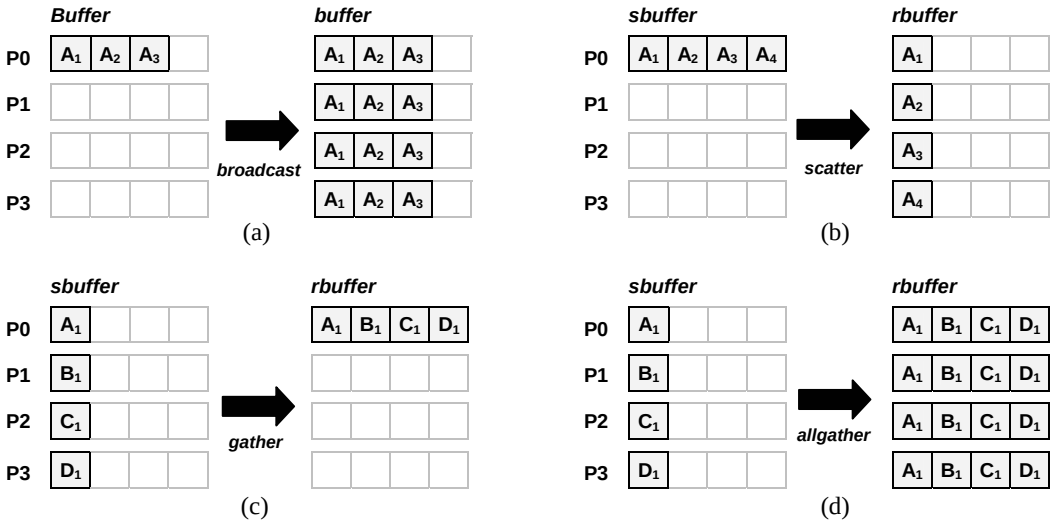


Figura 3.10. Operações de comunicação coletiva do MPI. *Broadcast* (a); *Scatter* (b); *Gatter* (c); *Allgatter* (d).

O MPI tem ainda outras operações de comunicação coletiva, incluindo algumas "variantes" de `MPI_Gather` e `MPI_Scatter` que permitem distribuir/coletar partes de tamanhos diferentes para processos diferentes.

3.4.2.4 A biblioteca MPI utilizada neste trabalho

Neste trabalho foi utilizada a biblioteca MPICH versão 1.2.5, desenvolvida pelo Argonne National Laboratory. Esta biblioteca é uma das implementações do padrão MPI mais utilizadas e é gratuita.

3.5 Análise de desempenho de programas paralelos

Uma vez implementado um programa paralelo é de fundamental importância que se faça uma análise do desempenho deste programa. Uma análise de desempenho consiste na execução de procedimentos, computacionais ou não, com o objetivo de identificar possíveis gargalos de execução e/ou partes do programa que podem ter seu desempenho melhorado (MARCARI JUNIOR, 2002).

3.5.1 Medidas desempenho

Durante uma análise são utilizadas medidas através das quais é possível obter informações sobre o desempenho do programa. A medida mais comum é o *speedup*, que indica o número de vezes que o programa paralelo é mais rápido que o seqüencial (QUINN, 1994). Ele é dado pela fórmula:

$$Speedup = \frac{T_s}{T_p}, \tag{3.1}$$

onde T_s é o tempo de execução do programa seqüencial e T_p é o tempo de execução do programa paralelo.

Na tentativa de melhor traduzir o desempenho do programa, outras medidas são formuladas a partir do *speedup*. Uma destas medidas é a eficiência. A eficiência de um programa paralelo indica o quanto o *speedup* obtido está próximo ou não do ótimo (MARCARI JUNIOR, 2002). Ela é dada pela fórmula:

$$Eficiência = \frac{speedup}{p}, \tag{3.2}$$

onde p é o número de processadores utilizados pelo programa paralelo.

O *speedup* geralmente é um valor entre 0 e p. Na prática, dificilmente o valor máximo é obtido por causa de sobrecargas relacionadas à comunicação e sincronização. Além disso, segundo a chamada *Lei de Amdahl* (FOSTER, 1995; QUINN, 1994), todo programa paralelo apresenta uma fração f seqüencial, que limita o seu *speedup* em:

$$Speedup \text{ máximo} = \frac{1}{f + \frac{(1-f)}{p}} \tag{3.3}$$

Existem casos, no entanto, onde um *speedup* superlinear (FOSTER, 1995; QUINN, 1994), ou seja, um *speedup* > *p*, é obtido. Por exemplo, na maioria dos computadores paralelos cada processador tem alguma memória *cache*. Um programa paralelo utilizando *p* processadores tem, portanto, *p* vezes mais memória *cache* que um programa que utiliza um único processador. Coletivamente o número de *cache hits* (STALLINGS, 2002) pode ser significantemente maior do que em um único processador, quando então o *speedup* pode ser maior que o número de processadores.

3.5.2 Ferramentas para análise de desempenho

Quando um programa não apresenta a aceleração esperada por exemplo, não é possível saber o(s) motivo(s) da mais baixa aceleração com base apenas no *speedup* e/ou eficiência calculados. Para isto existem ferramentas através das quais é possível analisar/entender o comportamento do programa, ou de partes dele.

Segundo Foster (1995), uma análise de desempenho consiste de três etapas: coleta, transformação e visualização dos dados de desempenho, e para cada uma delas existem ferramentas.

3.5.2.1 Coleta

Esta é a etapa onde os dados de desempenho são obtidos. Segundo Foster (1995), existem três técnicas/ferramentas de coleta: *profiling*, contagem e extração de traços de eventos (*event tracing*).

(3.5.2.1.1) ***Profiling***. Este tipo de técnica obtém o tempo gasto na execução de diferentes partes (procedimentos/funções) de um programa via amostragem do contador de programas. Em intervalos fixos de tempo o valor do contador de programas é consultado. Este valor é então utilizado para a construção de um histograma das freqüências de execução, que combinadas com as informações da tabela de símbolos do compilador são utilizadas para estimar o tempo gasto em diferentes partes do programa.

(3.5.2.1.2) Contagem. Neste tipo de técnica são utilizados contadores que registram o número de ocorrências de determinado evento. Por exemplo, um contador pode ser incrementado toda vez que uma função é executada para contar o número total de chamadas àquela função. A inserção de contadores no código do programa pode ser feita pelo compilador, por códigos

presentes nas rotinas das bibliotecas de programação paralela, ou pelo próprio programador. Contadores também podem ser utilizados para registrar o valor do relógio do sistema em diferentes partes do programa. Estes valores podem então ser posteriormente verificados para determinar o tempo gasto na execução de trechos do programa.

(3.5.2.1.3) Extração de traços de eventos. *Event tracing*, como é chamada, é a técnica que revela maiores detalhes sobre a execução de um programa. Ferramentas de análise de desempenho baseadas em *tracing* agem fazendo registros, datados da ocorrência, de eventos significativos ocorridos durante a execução de um programa como, por exemplo, chamadas a procedimentos/funções, envio/recepção de mensagens, etc. O código responsável pela geração dos registros pode ser gerado automaticamente (pelo compilador ou por códigos presentes nas rotinas das bibliotecas de programação) ou pelo programador.

3.5.2.2 Transformação

Durante a fase de coleta os dados de desempenho são usualmente armazenados em arquivos, em formatos na maioria das vezes não adequados à análise/interpretação pelo programador. Por isso é comum estes dados passarem por transformações, com o objetivo de reduzir o volume total de dados ou mesmo extrair outros tipos de dados a partir dos existentes. Por exemplo, o *profiling* de um programa paralelo revela o tempo de execução de cada procedimento/função deste programa. Uma transformação pode então ser feita para revelar tempos médios de execução ou outros dados estatísticos.

3.5.2.3 Visualização

O processo de visualização está diretamente ligado ao de transformação (em geral ambos os processos são executados por uma única ferramenta). Ferramentas de visualização em geral transformam os dados obtidos via *profiling*, contagem ou *event tracing* em dados e imagens mais facilmente entendidas pelo programador, por exemplo, gráficos de barras representando os tempos de execução de cada função em cada processo de um programa paralelo.

3.5.3 As ferramentas de análise de desempenho utilizadas neste trabalho: MPE e Jumpshot.

Neste trabalho, a análise de desempenho dos programas desenvolvidos foi feita com duas ferramentas: MPE (http://www-unix.mcs.anl.gov/mpi/www/) e Jumpshot 3.0

(http://www.mcs.anl.gov/perfvis/software/viewers/). Ambas são gratuitas e bastante conhecidas no meio acadêmico, por isso foram escolhidas.

MPE é uma ferramenta de coleta do tipo *event tracing*. Para utiliza-la basta liga-la ao programa MPI. Assim, quando este programa é executado os eventos são automaticamente registrados em um arquivo com extensão CLOG. São exemplos de eventos registrados automaticamente pela MPE o envio (SEND) e recepção (RECEIVE) de mensagens. A MPE também dispõe de funções que o programador pode utilizar para definir seus próprios eventos (*user-defined events*).

A ferramenta Jumpshot por sua vez analisa/transforma os dados gerados pela MPE, armazenados em um arquivo CLOG, e os apresenta ao usuário na forma de gráficos de barra como o que é mostrado na Figura 3.11.

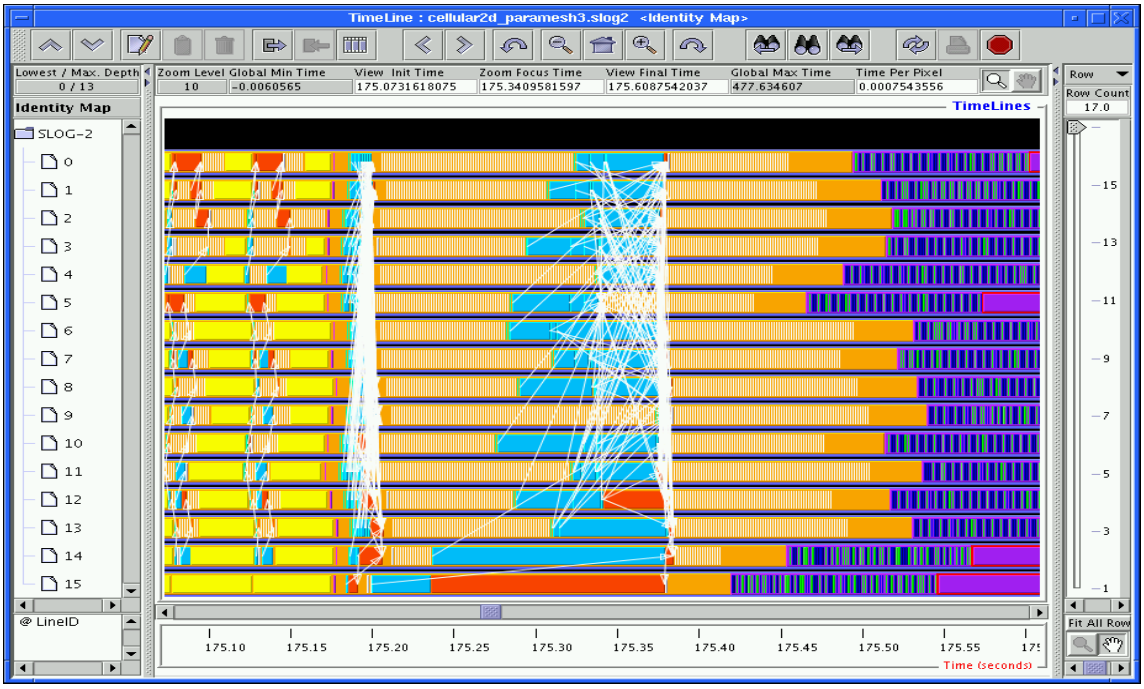


Figura 3.11. Tela da ferramenta Jumpshot.

O gráfico ilustra a execução de um programa MPI. Cada barra representa um processo do programa. Cada cor representa uma parte do programa definida pelo programador. As setas representam a troca de mensagens entre processos.

Capítulo 4

TDE paralela

Este capítulo apresenta os algoritmos de TDE exata paralelizados neste trabalho e as estratégias de paralelização desenvolvidas para estes algoritmos. Primeiro é apresentado o algoritmo de TDE por varredura independente e as estratégias de paralelização deste algoritmo em SMPs e em agregados. Depois é apresentado o algoritmo de TDE por propagação ordenada e a estratégia de paralelização deste algoritmo em agregados. Para a validação dos programas/implementações desenvolvidos foram utilizadas diferentes imagens de diferentes tamanhos. Este capítulo começa apresentando estas imagens.

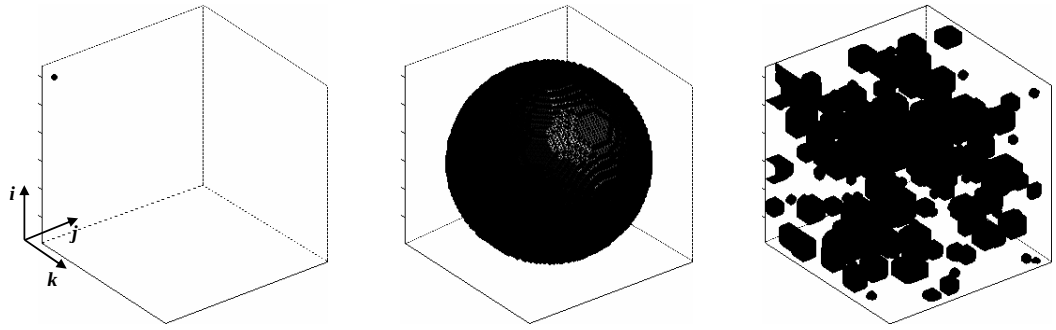
4.1 Imagens utilizadas para a validação das implementações

O tempo de execução da maioria dos algoritmos de transformada de distância euclidiana exata depende do conteúdo da imagem, ou seja, o tempo de execução de um mesmo algoritmo pode variar para imagens do mesmo tamanho se o número e a disposição dos pontos de objeto/fundo nas imagens são diferentes. Por isso, para validar a implementação dos algoritmos seqüenciais e das estratégias de paralelização desenvolvidas neste trabalho foram utilizadas, além de imagens de diferentes tamanhos (500×500×500, 550×550×550, 600×600×600, 650×650×650, 700×700×700, 750×750×750, 800×800×800), imagens com conteúdos diferentes. Estas imagens/conteúdos são descritas a seguir:

- **Um único ponto de fundo na posição (0, 0, 0) (Figura 4.1 (a)).** Este tipo de imagem foi sugerido por Maurer Jr, Qi e Raghavan (2003) porque é onde a TDE produz as maiores distâncias possíveis para um dado tamanho de imagem.

- **Uma esfera (Figura 4.1 (b)).** Este tipo de imagem foi sugerido por Cuisenaire e Macq (1999a) e Saito e Toriwaki (1994) porque o seu diagrama de Voronoi é bastante irregular, caso onde muitos algoritmos de TDE apresentam o seu pior desempenho.

- **Cubos randômicos**[21] **(Figura 4.1 (c)).** Este tipo de imagem foi sugerido por Eggers (1998) porque possui alguma semelhança com imagens reais.



(a) (b) (c)

Figura 4.1. Imagens teste. Imagem do ponto (a); imagem da esfera (b); e imagem dos cubos randômicos (c).

4.2 O algoritmo de TDE por varredura independente de Saito e Toriwaki

O algoritmo de TDE exata de Saito e Toriwaki (1994) é um dos mais conhecidos, por causa da sua eficiência e razoável facilidade de implementação. Tal algoritmo gera o mapa de distâncias de qualquer imagem *k*-D fazendo *k* transformações 1-D, uma para cada direção de coordenada.

4.2.1 O algoritmo seqüencial

Em imagens 3-D primeiro ele calcula a distância de cada ponto até o ponto de fundo mais próximo na sua linha (*transformação 1*). Estes valores são então utilizados para calcular a distância de cada ponto até o ponto de fundo mais próximo no seu plano (*transformação 2*). Finalmente, utilizando estes valores é calculada a distância de cada ponto até o ponto de fundo mais próximo na imagem (*transformação 3*). Formalmente:

Transformação 1. Dada uma imagem binária *F* contendo *L* linhas, *C* colunas e *P* planos, a *transformação 1* gera uma imagem *G* dada por:

$$G(i, j, k) = \min \{(j - y)^2; F(i, y, k) = 0,\ 1 \le y \le C\}, \qquad (4.1)$$

onde $1 \le i \le L$, $1 \le j \le C$ e $1 \le k \le P$.

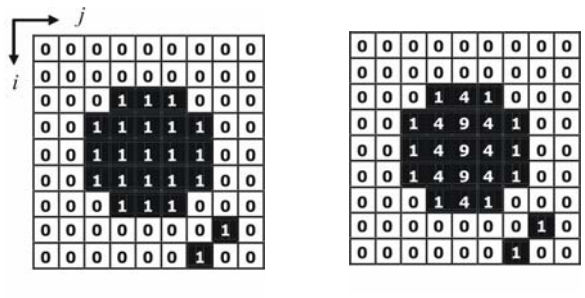
[21] Esta imagem foi gerada escolhendo-se aleatoriamente os centros e os tamanhos dos cubos. Os cubos foram desenhados até que o número de pontos de objeto/cubo ultrapassasse 50% do total de pontos da imagem.

Isto corresponde a calcular a distância de cada ponto (*i*, *j*, *k*) até o ponto de fundo mais próximo na sua linha, ou seja, na linha *i*.

O algoritmo da *transformação 1* consiste de uma varredura da esquerda para a direita (*forward scan*) mais uma varredura da direita para a esquerda (*backward scan*) em cada linha da imagem. Ele é apresentado a seguir, onde assume-se que todo $G(i, j, k)$ foi inicializado como ∞, ou seja, um número maior que a máxima distância possível na imagem:

```
para k = 1 até P faça {
    para i = 1 até L faça {
        para j = 1 até C faça {   \\ forward scan
            se      F(i, j, k) ≠ 0  então   G(i, j, k) = ( √G(i, j − 1, k)  + 1)²
            senão                           G(i, j, k) = 0
        }
        para j = C - 1 até 1 faça  \\ backward scan
            G(i, j, k) = min { ( √G(i, j + 1, k)  + 1)², G(i, j, k) }
    }
}
```

A Figura 4.2 mostra um plano de uma imagem 3-D (a) e este mesmo plano após a *transformação 1* (b). Note que, ao invés da distância euclidiana propriamente dita, é calculada a distância euclidiana ao quadrado.



(a) (b)

Figura 4.2. *Transformação 1*. O primeiro plano de uma imagem binária 3-D (a) e este plano após a *transformação 1* (b).

Transformação 2. A partir de *G*, a *transformação 2* (Figura 4.3) gera uma imagem *H* dada por:

$$H(i, j, k) = \min \{ G(x, j, k) + (i - x)^2; 1 \le x \le L\} \qquad (4.2)$$

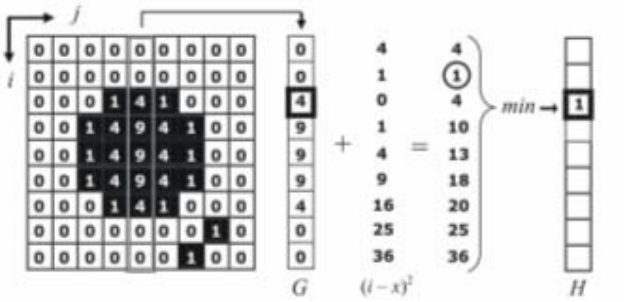


Figura 4.3. ***Transformação 2.*** Para calcular $H(i, j, k)$ some ao valor de cada ponto (x, j, k), ou seja, de cada ponto da coluna onde (i, j, k) está, a distância deste ponto até (i, j, k), isto é, $(i - x)^2$. Então defina $H(i, j, k)$ como sendo o menor destes valores.

O algoritmo da *transformação 2* consiste de uma varredura de cima para baixo mais uma varredura de baixo para cima em cada coluna de G. Durante a varredura para baixo, para cada ponto é feito o seguinte teste:

se ($G(i, j, k) > G(i\text{-}1, j, k) + 1$), faça para os valores de n,
tal que $0 \leq n \leq (G(i, j, k) - G(i\text{-}1, j, k) - 1) / 2$

    se $G(i\text{-}1, j, k) + (n + 1)^2 < G(i+n, j, k)$
        $H(i+n, j, k) = G(i\text{-}1, j, k) + (n + 1)^2$

    senão
        i++

senão
    $H(i, j, k) = G(i, j, k)$

Este teste verifica se o ponto $(i + n, j, k)$, para $0 \leq n \leq (G(i, j, k) - G(i\text{-}1, j, k) - 1) / 2$, está mais próximo do ponto de fundo da linha i - 1 do que do ponto de fundo da sua linha, cuja distância foi calculada durante a *transformação 1*. Se isto é verdade, sua distância até o ponto da linha i - 1 é calculada/armazenada. O valor máximo de n, $(G(i, j, k) - G(i\text{-}1, j, k) - 1) / 2$, é definido sabendo que $G(i\text{-}1, j, k) + (n + 1)^2 > G(i, j, k) + n^2$, para qualquer $n > (G(i, j, k) - G(i\text{-}1, j, k) - 1) / 2$. Note que a *transformação 2* irá demorar mais ou menos tempo dependendo dos valores de n, que por sua vez dependem dos resultados/valores da *transformação 1*, que dependem do conteúdo da imagem.

Após a varredura para baixo, a varredura para cima procede da mesma forma, isto é o mesmo teste é aplicado, porém, no sentido contrário.

Transformação 3. A partir de *H*, a *transformação 3* gera uma imagem *S* (na prática, as imagens *F*, *G*, *H* e *S* podem ser a mesma) dada por:

$$S(i, j, k) = \min \{ H(i, j, z) + (k - z)^2; 1 \leq z \leq P\} \quad (4.3)$$

O algoritmo da *transformação 3* é o mesmo da *transformação 2*. Porém, a imagem é varrida para traz e para frente, ou seja, na direção do eixo *k*.

Como o algoritmo trabalha com a distância euclidiana ao quadrado, a operação 'raiz quadrada' deve ser aplicada sobre cada ponto da imagem após a *transformação 3*.

4.2.2 Estratégia de paralelização em SMPs

Esta seção apresenta a estratégia de paralelização da TDE 3-D de Saito e Toriwaki em SMPs. Tal estratégia é ilustrada na Figura 4.4. No texto e na figura ela foi dividida em estágios, para facilitar a sua apresentação.

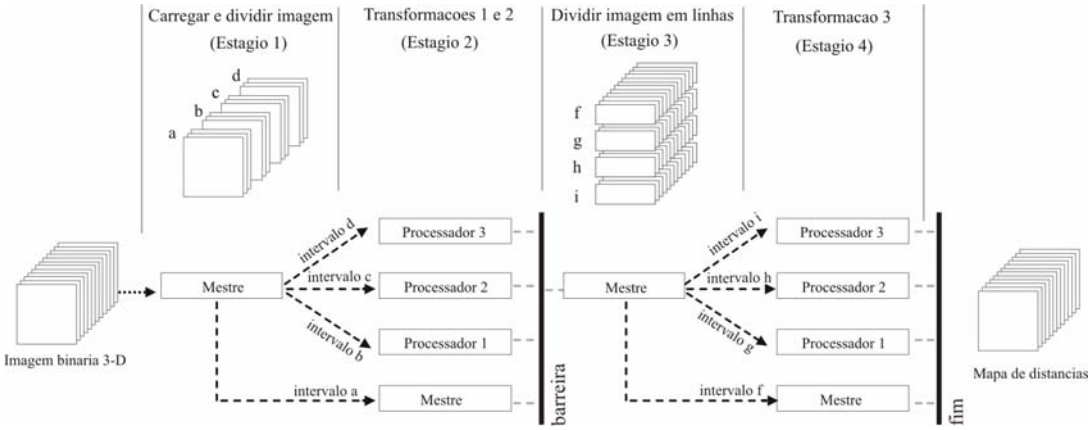


Figura 4.4. Estratégia de paralelização do algoritmo de Saito e Toriwaki em SMPs. Na figura, uma linha serrilhada ligando dois processadores indica uma comunicação via memória compartilhada.

Em um primeiro estágio um processador, que aqui será chamado de mestre, carrega a imagem 3-D do disco para a memória e "divide" esta imagem em partes, uma para cada processador, cada uma contendo um mesmo número de planos consecutivos da imagem. Como os processadores compartilham memória, a divisão propriamente dita da imagem não é necessária. Ao invés disso o mestre calcula o intervalo de planos de cada processador. Ele então passa esta informação para cada processador, que executa a *transformação 1* e em seguida a *transformação 2* sobre a sua parte da imagem. Depois de executar a segunda

transformação os processadores encontram uma barreira[22]. Esta barreira faz cada processador esperar os demais terminarem a segunda transformação, isto porque cada processador precisa dos resultados da segunda transformação de todos os outros para iniciar a *transformação 3*. Além disso, após a segunda transformação a imagem precisa ser "re-dividida", porque na terceira transformação os processadores varrem a imagem na direção do eixo *k*. A re-divisão é feita pelo processador mestre. Ele re-divide a imagem em partes contendo um mesmo número de linhas consecutivas da imagem. Assim como no primeiro estágio, a re-divisão propriamente dita da imagem não é necessária. Ao invés disso o mestre calcula o intervalo de linhas de cada processador. Ele então passa esta informação para os demais processadores que executam a *transformação 3* sobre a sua parte da imagem. Depois da terceira transformação cada processador aplica a operação 'raiz quadrada' sobre os pontos na sua parte da imagem e termina.

4.2.2.1 Detalhes de implementação

Esta estratégia de paralelização foi implementada no SMP descrito na Seção 3.2.2.3. A implementação foi feita com a linguagem C e OpenMP (compilador C do pacote Sun Workshop 6 update 2). A implementação/programa começa com uma única *thread*, chamada *thread* principal/mestre. Primeiro esta *thread* carrega a imagem do disco para a memória. Após carregar a imagem ela encontra a diretiva OpenMP `#pragma omp parallel` que indica o início de uma região paralela. Então, a *thread* principal cria outras três *threads* e "divide" a imagem entre elas. Cada *thread* executa a *transformação 1* e, em seguida, a *transformação 2* sobre a sua parte da imagem. Depois da segunda transformação as *threads* encontram a diretiva `#pragma omp barrier`, ou seja, uma barreira, que faz cada *thread* esperar pelas demais. Depois da barreira a *thread* principal "re-divide" a imagem. Todas as *threads* então executam a *transformação 3* sobre a sua nova parte da imagem. Depois da terceira transformação cada *thread* aplica a operação 'raiz quadrada' sobre os pontos na sua parte da imagem e termina.

4.2.2.2 Balanceamento de carga

Por *balanceamento de carga* entende-se a situação onde os processadores, trabalhando na resolução de um problema, executam aproximadamente a mesma quantidade de trabalho. Por *não-balanceamento* entende-se a situação onde a quantidade de trabalho de cada

[22] Uma barreira é uma primitiva de sincronização. Quando um processador atinge a barreira ele é bloqueado até que todos os outros a atinjam.

processador difere da quantidade de trabalho dos demais. Quando isso acontece os processadores com menos quantidade de trabalho ficam ociosos, enquanto que os processadores com mais quantidade de trabalho ficam sobrecarregados, o que resulta na perda de desempenho. Neste trabalho foram utilizadas as ferramentas MPE e Jumpshot para verificar o balanceamento de carga (ou o não-balanceamento de carga) nos programas desenvolvidos. A Figura 4.5 mostra um diagrama gerado por estas ferramentas[23]. Este diagrama foi gerado durante o processamento da imagem dos cubos randômicos 700×700×700 no SMP.

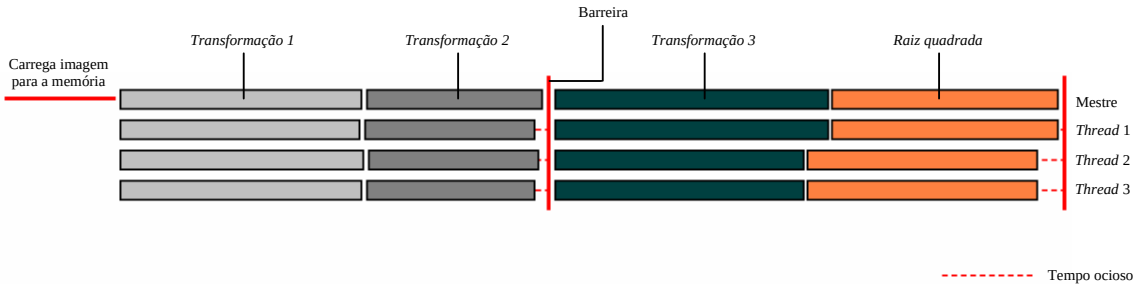


Figura 4.5. **Diagrama de execução da imagem dos cubos randômicos 700×700×700 no SMP.**

Na figura está claro o balanceamento de carga. Note que cada *thread* começa e termina cada uma das três transformações quase ao mesmo. Tal balanceamento foi possível porque: (i) a imagem foi dividida em partes de igual tamanho; (ii) o conteúdo das partes eram parecidos e, por isso, as *transformações 1*, *2* e *3* demoraram o mesmo tempo em cada *thread*; e (iii) o sistema operacional, responsável pela execução das *threads*, deu oportunidades iguais de execução para todas elas.

A Figura 4.6 mostra o diagrama gerado durante o processamento de uma outra imagem, a imagem da esfera 700×700×700.

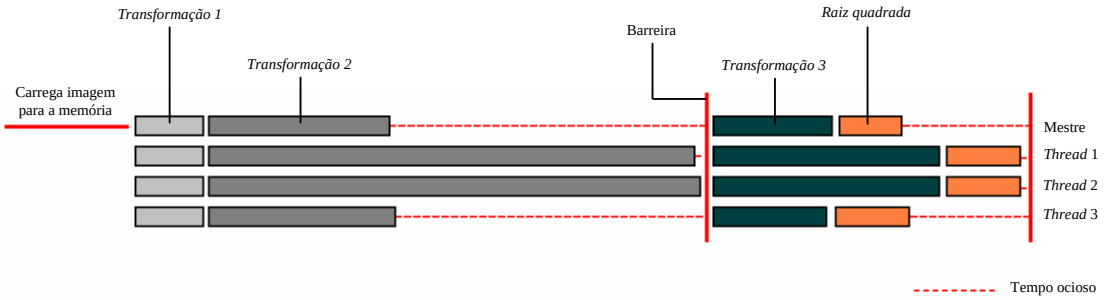


Figura 4.6. Diagrama de execução da imagem da esfera 700×700×700 no SMP.

[23] Neste trabalho todos os diagramas gerados pela ferramenta Jumpshot foram adaptados/redesenhados para melhorar a sua apresentação.

Nesta imagem está claro um não-balanceamento de carga nas *transformações 2* e *3*. Como foi discutido na Seção 4.2.1, os algoritmos destas transformações demoram mais ou menos tempo dependendo do conteúdo da imagem. No caso da imagem da esfera o conteúdo das partes atribuídas às *threads 1* e *2* fez o algoritmo da *transformação 2* demorar mais que o dobro do tempo das outras *threads*. O mesmo aconteceu na *transformação 3*.

Para resolver este problema de não-balanceamento, durante a "divisão" da imagem a *thread* mestre, ao invés de dividir a imagem em partes de igual tamanho, poderia dividir a imagem de forma que a *thread* mestre e a *thread 3* ficassem com partes maiores e as *threads 1* e *2*, conseqüentemente, com partes menores, aproximando o tempo de execução de todas elas. Tal solução, no entanto, poderia não funcionar em imagens com outros conteúdos. De fato, não existe uma solução de balanceamento que funcione para todas imagens.

4.2.2.3 Resultados

Esta seção apresenta os resultados obtidos com a implementação paralela do algoritmo de Saito e Toriwaki no SMP. A Tabela 4.1 traz os tempos médios de execução da implementação do algoritmo seqüencial de Saito e Toriwaki e os tempos médios de execução da implementação paralela deste algoritmo no SMP. Para calcular as médias cada implementação foi executada cinco vezes. Na tabela, os valores entre parênteses representam o desvio padrão.

Tabela 4.1. Tempos de execução, em segundos, do algoritmo seqüencial de Saito e Toriwaki e da implementação paralela deste algoritmo no SMP.

Tamanho da imagem	Imagem do ponto		Imagem da esfera		Imagem dos cubos randômicos	
	seqüencial	Paralelo	seqüencial	paralelo	seqüencial	paralelo
500×500×500	100 (0)	20 (0)	165 (0)	50 (1)	104 (0)	23 (0)
550×550×550	149 (0)	30 (0)	238 (0)	74 (0)	146 (0)	32 (0)
600×600×600	190 (0)	38 (1)	316 (0)	98 (1)	188 (0)	42 (0)
650×650×650	236 (1)	48 (1)	411 (0)	129 (1)	236 (0)	52 (0)
700×700×700	304 (1)	61 (0)	536 (0)	168 (1)	300 (0)	66 (0)
750×750×750	382 (0)	77 (0)	686 (0)	218 (1)	373 (0)	82 (0)
800×800×800	549 (2)	115 (1)	928 (1)	299 (0)	500 (2)	112 (1)

A partir dos tempos de execução dos programas seqüencial e paralelo foram calculados os *speedups*, que são mostrados no gráfico da Figura 4.7. Note que os maiores

speedups foram aqueles obtidos com a imagem do ponto, sendo que os *speedups* obtidos com a imagem dos cubos ficaram muito próximos destes. Em ambas as imagens os *speedups* foram maiores que quatro, caracterizando-se como *speedups* superlineares[24]. O autor deste trabalho acredita que tais *speedups* foram obtidos por causa da grande quantidade de memória *cache* em cada processador.

Os menores *speedups* foram os obtidos com a imagem da esfera por causa do não-balanceamento de carga que ocorre nas *transformações 2* e *3*. Mesmo assim estes *speedups* sempre foram maiores que três, o que significa uma *eficiência* maior que 75%, um resultado bastante positivo.

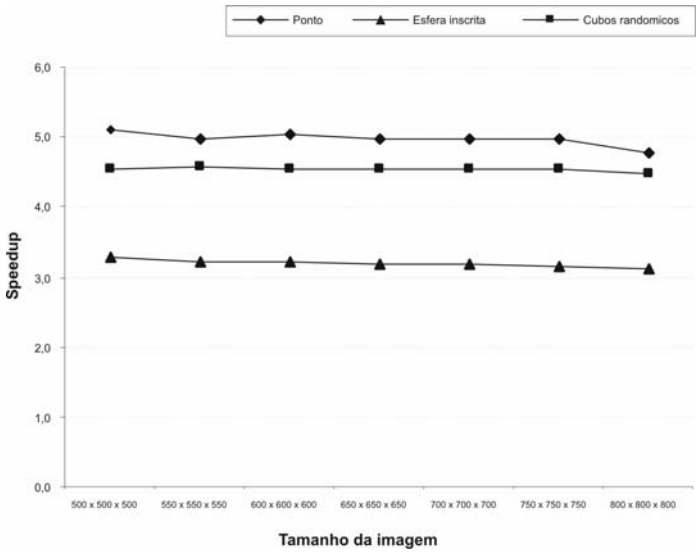


Figura 4.7. *Speedups* obtidos com a implementação paralela do algoritmo de Saito e Toriwaki no SMP.

4.2.3 Estratégia de paralelização em agregados

Esta seção apresenta a estratégia de paralelização da TDE 3-D de Saito e Toriwaki em agregados. Tal estratégia é ilustrada na Figura 4.8. A figura ilustra a estratégia de paralelização em um agregado com quatro nós. No texto e na figura ela foi dividida em estágios, para facilitar a sua apresentação.

[24] Quando o *speedup* é maior que o número de processadores ele é chamado de superlinear. Em um SMP com quatro processadores, um *speedup* maior que quatro é chamado de superlinear.

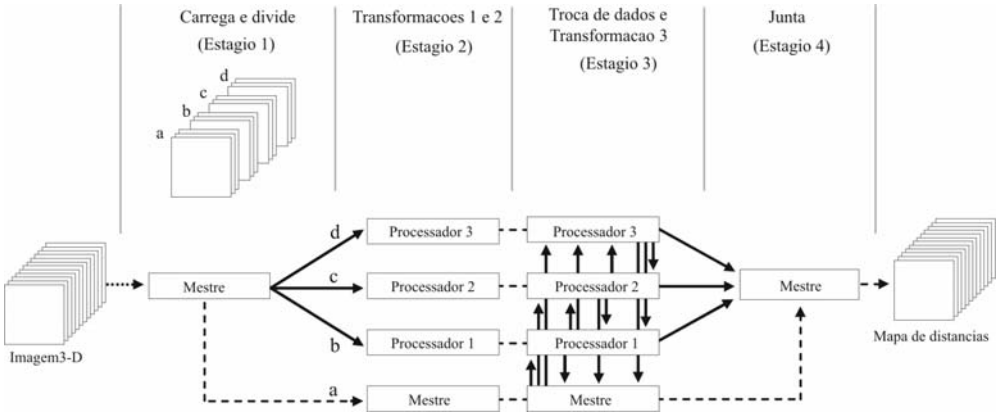


Figura 4.8. Estratégia de paralelização do algoritmo de Saito e Toriwaki em agregados. Na figura, uma linha ligando dois processadores representa uma comunicação via troca de mensagens. Uma linha serrilhada representa uma comunicação via memória compartilhada.

No primeiro estágio o processador mestre (processador do nó onde a imagem está armazenada) carrega a imagem 3-D do disco para a memória e divide esta imagem em partes, uma para cada processador, cada uma contendo um mesmo número de planos consecutivos da imagem. Destas partes, uma permanece no nó mestre. Cada uma das outras é enviada para um processador diferente. Após receber a sua parte da imagem, cada processador executa a *transformação 1* e, em seguida, a *transformação 2*, sobre ela. No próximo estágio os processadores executam a *transformação 3*, mas antes disso é necessário "juntar" os planos. Como foi descrito na Seção 4.2.1, durante a *transformação 3* a imagem é varrida na direção do eixo *k*. Porém, no primeiro estágio do algoritmo paralelo os planos da imagem são distribuídos entre os processadores, que não compartilham memória. Por isso, antes da *transformação 3*, é necessário "juntar" os planos da imagem. Depois é necessário re-dividir a imagem em partes onde a terceira transformação será executada. A imagem pode ser re-dividida de diferentes formas. Neste trabalho optou-se por re-dividi-la em partes contendo um mesmo número consecutivo de linhas, como mostra a Figura 4.9.

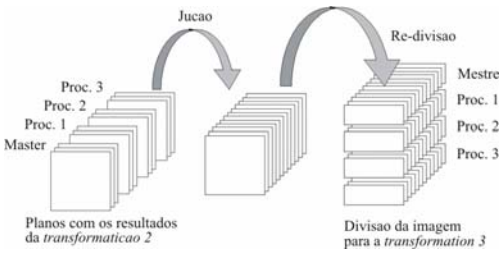


Figura 4.9. Junção e re-divisão dos dados para a *transformação 3*.

A junção dos planos e a re-divisão da imagem em linhas pode ser implementada de diferentes formas, por exemplo, (i) cada processador pode enviar para o mestre os planos com os resultados da *transformação 2*; o mestre então "junta" todos os planos, re-divide a imagem e envia uma parte para cada processador; ou (ii) cada processador pode enviar diretamente para cada um dos outros as linhas que aquele processador usa na *transformação 3*. Neste trabalho optou-se por esta segunda forma porque a quantidade de dados que tem que ser transmitida através da rede é menor. Tal forma (esquema de troca de dados) é ilustrada na Figura 4.10. Note que cada processador envia uma mensagem para cada um dos outros. Apesar das mensagens poderem ser enviadas em qualquer ordem, enviá-las de uma forma que não existam dois ou mais processadores enviando uma mensagem para um mesmo processador destino ao mesmo tempo pode resultar em algum paralelismo durante as comunicações.

Cada processador após receber os dados (as linhas) de todos os outros, executa a *transformação 3* sobre eles. Como o algoritmo de Saito e Toriwaki trabalha com a distância euclidiana ao quadrado, após a *transformação 3* os processadores aplicam a operação 'raiz quadrada' sobre cada ponto na sua parte da imagem. Então, cada processador envia a sua parte para o mestre, que "junta" as partes de todos os processadores para construir o mapa de distâncias euclidianas final.

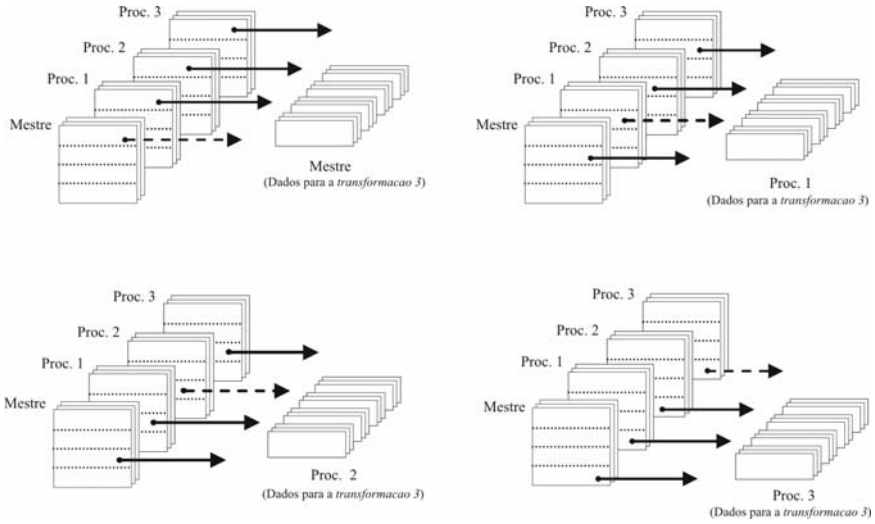
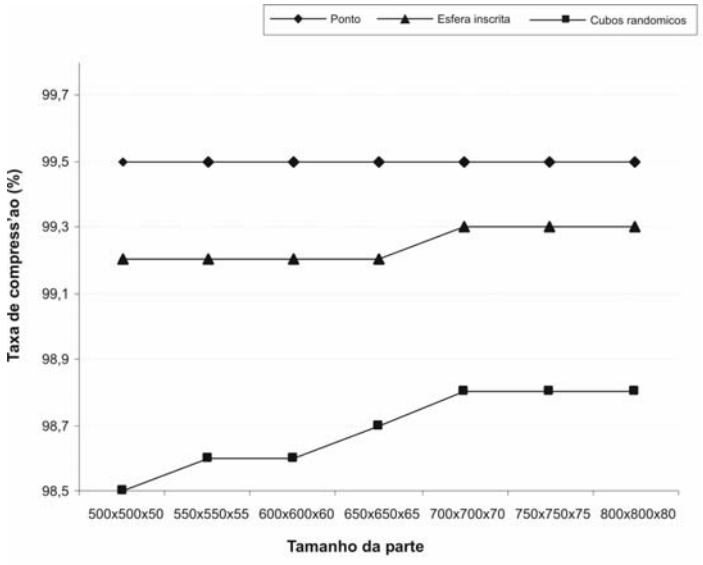


Figura 4.10. Troca de dados para junção e re-divisão da imagem para a *transformação 3*. Na figura, uma seta representa uma comunicação via troca de mensagens. Uma seta serrilhada representa uma comunicação via memória compartilhada.

4.2.3.1 Detalhes de implementação

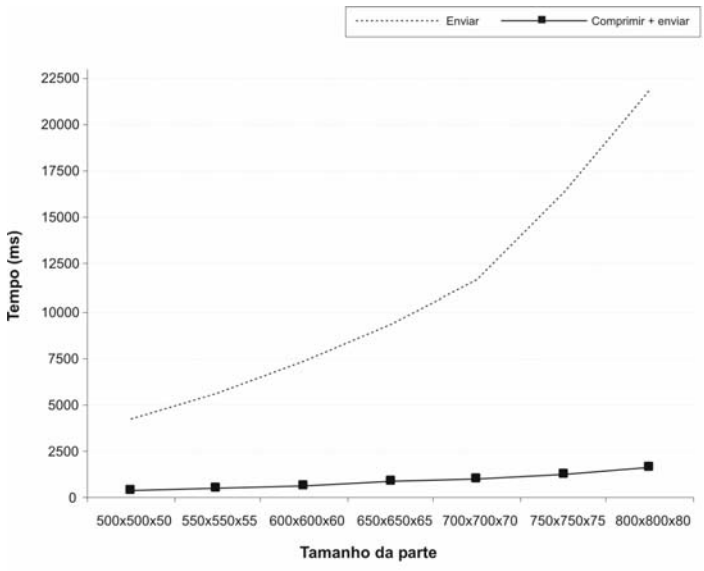
Esta estratégia de paralelização foi implementada no agregado descrito na Seção 3.2.3.2 usando a linguagem C e a biblioteca MPICH, e compilada com o compilador *gcc* versão 3.3.5. Os processos do programa paralelo foram distribuídos entre os dez nós do agregado de acordo com um esquema *Roudin-Robin* usando uma lista com o número IP dos computadores que fazem parte do agregado, sendo que foi "colocado" um processo por nó.

No estágio 1 o processo mestre (processo do nó onde a imagem está armazenada) divide a imagem em partes e comprime cada uma das partes antes de enviá-las para os outros. A compressão é usada para reduzir o tempo gasto com comunicação (troca de mensagens), uma vez que a rede do agregado é uma rede com baixa largura de banda. A compressão é feita com a biblioteca/ferramenta Zlib (http://www.zlib.org). Como imagens binárias são compostas apenas de 0s e 1s, elas apresentam altos graus de redundância, que favorecem a compressão. O gráfico na Figura 4.11 (a) apresenta a taxa média de compressão obtida quando comprimindo as partes de cada uma das imagens teste. Note que com todas as imagens as taxas de compressão foram bastante altas. Já o tempo médio de compressão foi bem pequeno. O gráfico em (b) mostra o tempo necessário para transmitir uma parte da imagem dos cubos randômicos sem e *com compressão*[25]. Esta foi a imagem com as piores taxas de compressão, assim como a com os maiores tempos de compressão. Mesmo assim, comprimir e enviar uma parte desta imagem foi significantemente mais rápido que enviá-la sem comprimir.



(a)

[25] Por tempo *com compressão* entende-se o tempo para comprimir mais o tempo para enviar os dados comprimidos através da rede.



(b)

Figura 4.11. Compressão das partes das imagens teste no estágio 1. Em (a) têm-se a taxa de compressão de uma parte de cada uma das imagens teste. Em (b) têm-se os tempos para enviar uma parte da imagem dos cubos randômicos sem e com compressão. Em ambos os gráficos são mostradas partes com tamanhos 500×500×50, 600×600×60, etc., porque as imagens teste tinham tamanhos 500×500×500, 600×600×600, etc., e o agregado utilizado neste trabalho dez nós.

As partes comprimidas são enviadas do mestre para os outros processadores com a função MPI *SEND pronto* (`MPI_RSend`). Tal função, apresentada na Seção 3.4.2.2.1, requer que o RECEIVE já tenha sido postado pelo escravo antes da realização do SEND pelo mestre, mas tem desempenho superior ao das outras, porque protocolos mais simples são utilizados internamente pelo MPI quando ele sabe que o RECEIVE já foi feito pelo processador de destino.

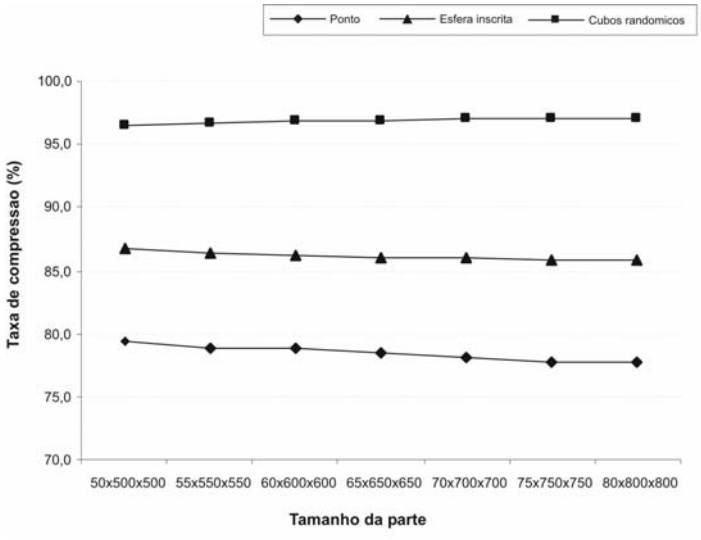
Cada nó do agregado utilizado neste trabalho tem 1.5 GB de memória RAM. Porém, uma imagem 750×750×750, por exemplo, requer 1.57 GB[26]. Portanto, durante o estágio 1, se o processador mestre carregar toda a imagem para a memória o sistema operacional deste nó iniciará sucessivas operações de leitura e escrita em disco (*swapping*), para manter na memória as partes que estão em uso e em disco as partes que não estão. Neste caso o tempo de execução do programa paralelo aumentará drasticamente. Para evitar a realização de *swapping* neste trabalho a imagem é carregada sob demanda, ou seja, a parte que é enviada do processador mestre para o escravo 1 só é carregada para a memória do mestre no momento do

[26] Considerando que cada ponto tem 32 bits uma imagem 750×750×750 tem aproximadamente 1.57 GB.

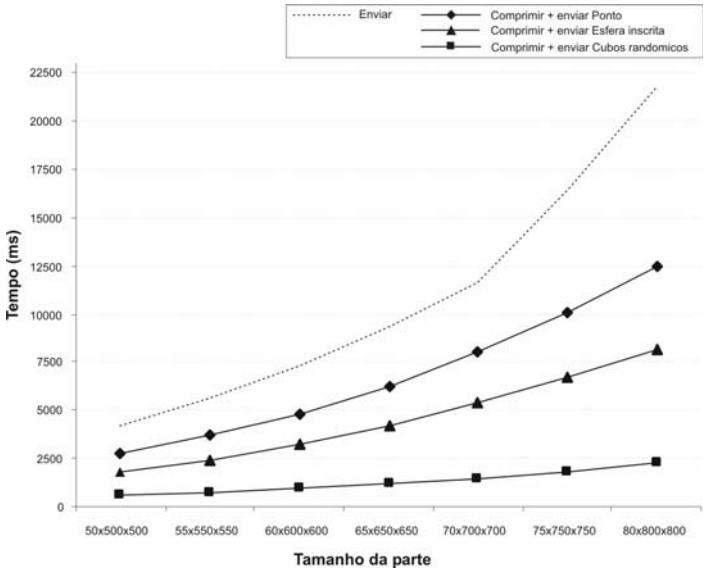
envio, sendo então comprimida, enviada e imediatamente removida da memória. Só então a parte do processador 2 é carregada, comprimida, enviada e removida, e assim sucessivamente.

No estágio 3 do algoritmo paralelo, antes de cada processador enviar para cada um dos outros as linhas que eles utilizam na terceira transformação também é utilizada compressão. As linhas comprimidas são enviadas com a função MPI *Send padrão não-bloqueante* (`MPI_ISend`). Uma função não-bloqueante é utilizada porque um processador pode postar um SEND com as linhas de um processador e, enquanto esta comunicação ocorre em *background*, comprimir as linhas do próximo processador.

Ainda no estágio 3, após executar a *transformação 3* e aplicar a operação 'raiz quadrada' sobre todos os pontos da sua parte da imagem, cada processador comprime esta parte para enviá-la ao mestre. A taxa média de compressão obtida para as partes das imagens teste são ilustradas no gráfico da Figura 4.12 (a). Note que, diferente do que ocorreu no primeiro estágio, neste as piores taxas de compressão são as da imagem do ponto, porque, diferente do primeiro estágio, onde as partes contêm apenas 0s e 1s, no terceiro elas contêm as distâncias, e a imagem dos pontos é onde a imagem com o menor número de distâncias repetidas, porque ela tem um único ponto de fundo. Apesar disso, comprimir e enviar esta imagem ainda foi mais rápido que enviá-la sem comprimir, como mostra o gráfico da Figura 4.12 (b).



(a)



(b)

Figura 4.12. Compressão das partes das imagens teste após a *transformação 3*. Em (a) têm-se a taxa de compressão de uma parte de cada uma das imagens teste. Em (b) têm-se os tempos para enviar uma parte de cada uma das imagens teste sem e com compressão. Em ambos os gráficos são mostradas partes com tamanhos 500×500×50, 600×600×60, etc., porque as imagens teste tinham tamanhos 500×500×500, 600×600×600, etc., e o agregado utilizado dez nós.

4.2.3.2 Balanceamento de carga

Para verificar o balanceamento (ou não-balanceamento de carga) da implementação desenvolvida foram utilizadas as ferramentas MPE e Jumpshot. A Figura 4.13 mostra um diagrama gerado por estas ferramentas durante o processamento da imagem da esfera 800×800×800 no agregado.

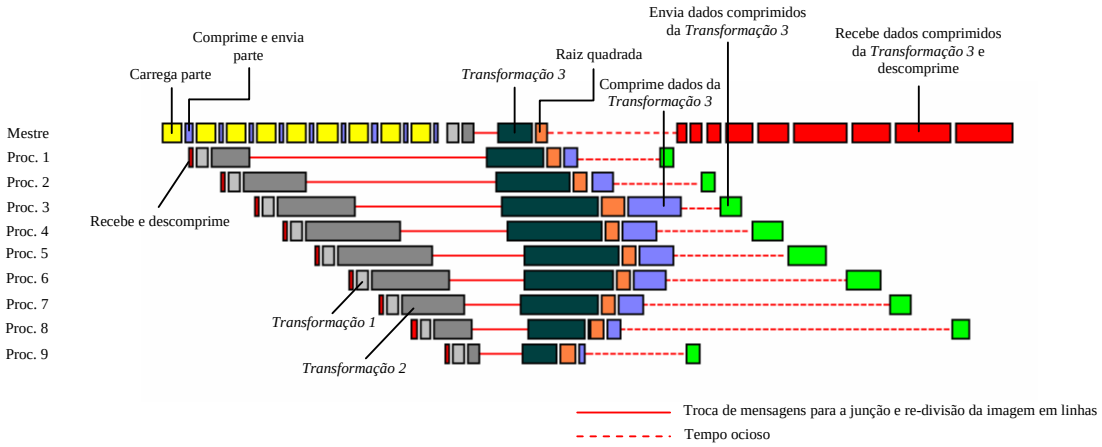


Figura 4.13. Diagrama de execução da imagem da esfera 800×800×800 no agregado.

Note que o processador mestre carrega, comprime e envia uma parte da imagem para o processador 1, que por sua vez faz a descompressão desta parte para então executar as *transformações 1* e *2* sobre ela. Enquanto isso o processador mestre carrega, comprime e envia uma outra parte da imagem para o processador 2, que faz a descompressão desta e executa as *transformações 1* e *2* sobre ela, e assim sucessivamente. Note que o tempo gasto na compressão e o envio de cada parte da imagem resulta em um não-balanceamento de carga. Por isso, quando o processador 8, por exemplo, recebe a sua parte da imagem, os processadores 1, 2, 3 e 4 já terminaram as *transformações 1* e *2* nas suas partes. Um não-balanceamento também ocorre durante a *transformação 2*. O diagrama da Figura 4.13 mostra que a *transformação 2* demorou menos tempo em alguns processadores (nos processadores 1 e 9, por exemplo) e mais tempo em outros (nos processadores 4 e 5, por exemplo). Como o tempo de execução da *transformação 2* em cada processo depende do conteúdo da parte da imagem deste processo é difícil resolver este problema de não-balanceamento. De fato seria necessário uma análise do conteúdo da imagem. De acordo com esta análise mais ou menos planos seriam incluídos em cada parte da imagem. Porém, uma análise deste tipo é demasiadamente complexa e demorada, e executá-la levaria mais tempo que o tempo perdido em função do não balanceamento.

4.2.3.3 Resultados

Esta seção apresenta os resultados obtidos com a implementação paralela do algoritmo de Saito e Toriwaki no agregado. A Tabela 4.2 traz os tempos médios de execução da implementação do algoritmo seqüencial de Saito e Toriwaki executando em um nó do agregado e os tempos médios de execução da implementação paralela executando em dez nós. Para calcular as médias cada implementação foi executada cinco vezes. Na tabela, os valores entre parênteses representam o desvio padrão. Note que o tempo de execução do programa seqüencial aumenta drasticamente nas imagens com 750×750×750 ou mais pontos, porque imagens deste tamanho não couberam inteiramente na memória de um único nó que então começou a fazer *swapping*.

A partir dos tempos de execução dos programas seqüencial e paralelo foram calculados os *speedups*, que são mostrados no gráfico da Figura 4.14. Note que o *speedup* aumenta bastante nas imagens com 750×750×750 ou mais pontos, porque imagens deste tamanho não couberam inteiramente na memória de um único nó, que começou a fazer

swapping, elevando drasticamente o tempo de execução do programa seqüencial. Como no programa paralelo a imagem é distribuída entre os nós, isso não aconteceu.

Tabela 4.2. Tempos de execução, em segundos, do algoritmo seqüencial de Saito e Toriwaki e da implementação paralela deste algoritmo no agregado.

Tamanho da imagem	Imagem com um ponto de fundo		Imagem da esfera		Imagem dos cubos	
	seqüencial	paralelo	Seqüencial	paralelo	seqüencial	paralelo
500×500×500	36 (0)	31 (4)	53 (0)	29 (1)	33 (0)	26 (1)
550×550×550	48 (1)	43 (0)	74 (0)	38 (1)	44 (0)	35 (1)
600×600×600	64 (0)	56 (0)	100 (0)	51 (2)	58 (1)	47 (2)
650×650×650	80 (0)	72 (1)	133 (1)	69 (5)	73 (0)	60 (1)
700×700×700	103 (3)	92 (0)	172 (1)	82 (1)	91 (0)	75 (1)
750×750×750	535 (28)	120 (0)	598 (42)	110 (1)	486 (13)	106 (1)
800×800×800	974 (20)	175 (4)	1044 (32)	161 (5)	863 (41)	131 (1)

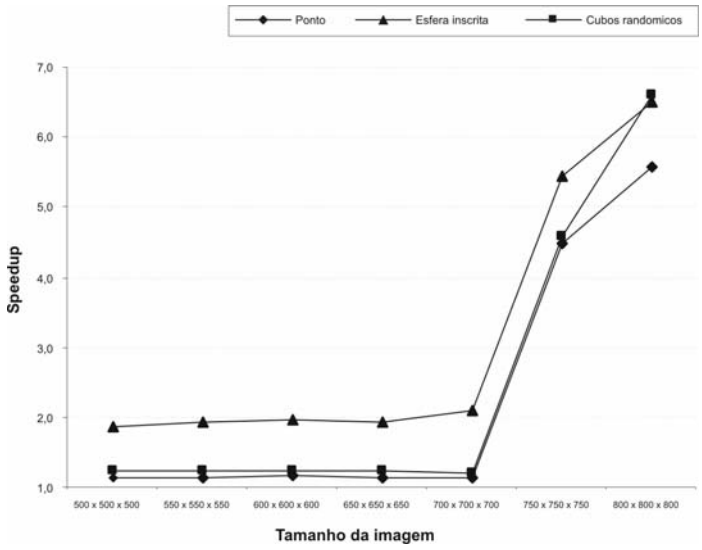


Figura 4.14. *Speedups* obtidos com a implementação paralela do algoritmo de Saito e Toriwaki no agregado.

Como um agregado de computadores é uma arquitetura que pode ser facilmente expandida, uma característica desejável nos programas desenvolvidos para este tipo de arquitetura é a *escalabilidade*. Por escalibidade entende-se a capacidade de um programa em executar usando um número maior de processos/processadores. Por isso, a implementação paralela do algoritmo de Saito e Toriwaki foi executada com diferentes números de processos/processadores. Os resultados (tempos de execução) destas execuções são mostrados

no gráfico da Figura 4.15. Como pode ser visto, os tempos de execução diminuem à medida que o número de processadores aumenta. Portanto, a estratégia de paralelização desenvolvida pode ser considerada escalável.

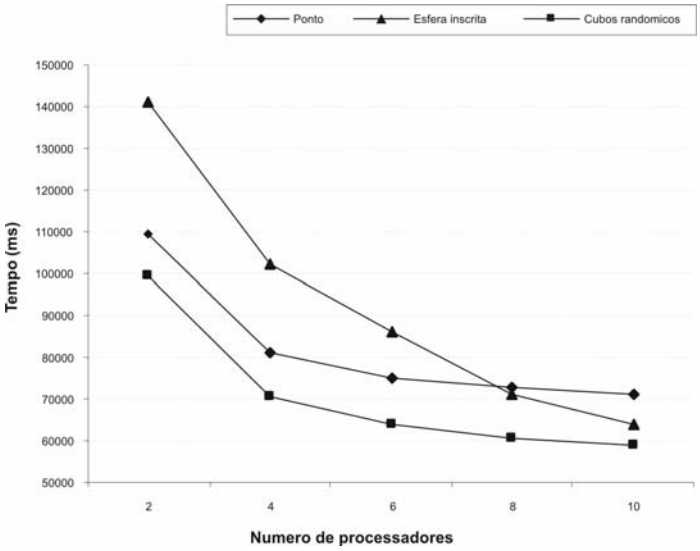


Figura 4.15. Tempos de execução da implementação paralela no agregado usando 2, 4, 6, 8 e 10 processadores.

4.3 O algoritmo de TDE por propagação ordenada de Eggers

Eggers (1998) propôs dois algoritmos para transformada de distância euclidiana exata de imagens 2-D, o propagação suficiente d_1 (*cityblock*) e o propagação suficiente d_∞ (*chessboard*). Este último foi descrito na Seção 2.5.3. Ele é chamado de propagação d_∞ porque consiste de uma série de iterações que, começando da borda do objeto, calculam e transmitem/propagam o valor de distância de cada ponto para os seus vizinhos, sendo que os pontos que transmitem o seu valor na iteração $i + 1$ são aqueles que estão a uma distância *chessboard* i de um ponto de fundo qualquer. Segundo Eggers, o propagação suficiente d_∞ apresenta melhor desempenho médio que o d_1, e pode ser estendido para imagens 3-D. Tal extensão no entanto não é trivial, nem é descrita no seu artigo. A extensão deste algoritmo foi feita pelo autor deste trabalho, e é descrita a seguir.

4.3.1 O algoritmo seqüencial

O algoritmo consiste de uma série de iterações que, começando da borda do objeto, calculam e propagam a informação de distância armazenada em cada ponto para os seus vizinhos. Na iteração *i* um conjunto de pontos, chamado *conjunto de contorno*, propaga informação para os seus vizinhos. Os vizinhos cujas distâncias são alteradas/atualizadas formam o conjunto de contorno da próxima iteração, $i + 1$. Todo ponto p no conjunto de contorno tem associado a ele um índice k, que indica a direção, ou seja, o vizinho, $N_k(p)$, para o qual ele deve transmitir informação. Os vizinhos de p são numerados como na Figura 4.16.

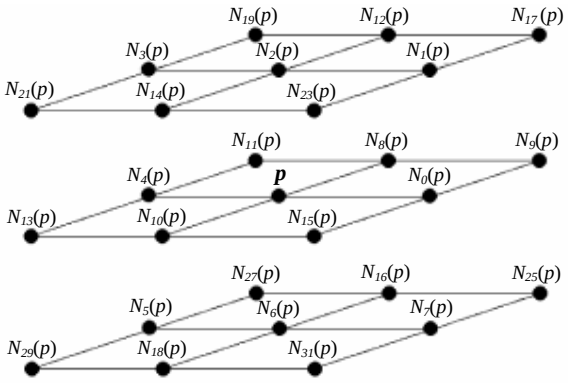


Figura 4.16. **Vizinhos de um ponto *p*.** Neste texto, os pontos a uma distância euclidiana 1 (ou seja, os pontos $N_0(p)$, $N_2(p)$, $N_4(p)$, $N_6(p)$, $N_8(p)$ e $N_{10}(p)$) são chamados vizinhos diretos de p; os pontos a uma distância euclidiana $\sqrt{2}$ ($N_1(p)$, $N_3(p)$, $N_5(p)$, $N_7(p)$, $N_9(p)$, $N_{11}(p)$, $N_{13}(p)$, $N_{15}(p)$, $N_{12}(p)$, $N_{14}(p)$, $N_{16}(p)$ e $N_{18}(p)$) são chamados vizinhos indiretos de p; e os pontos a uma distância euclidiana $\sqrt{3}$ ($N_{17}(p)$, $N_{19}(p)$, $N_{21}(p)$, $N_{23}(p)$, $N_{25}(p)$, $N_{27}(p)$, $N_{29}(p)$ e $N_{31}(p)$) são chamados vizinhos extremos de p.

Diferente dos algoritmos tradicionais de propagação ordenada que transmitem a informação de um ponto para todos os seus vizinhos, este algoritmo só transmite para aqueles que realmente é necessário, de forma que a informação de um ponto p "chegue" até um ponto q através dos pontos que estão no caminho de custo mínimo entre eles. No caso de uma imagem com um único ponto de fundo, esta técnica de propagação, que Eggers chamou de *propagação suficiente*, garante que cada ponto será visitado uma única vez. Os pontos com índice de direção k = 0, 2, 4, 6, 8 e 10, por exemplo, transmitem seu valor para um vizinho apenas, $N_k(p)$. Os pontos com índice k = 1, 3, 5 e 7 transmitem para três, $N_{k-1}(p)$, $N_k(p)$ e $N_{k+1}(p)$. Os pontos com índice k = 9, 11, 13 e 15, assim como os pontos com índice de direção k = 12, 14, 16 e 18, também transmitem para três, como mostra a Figura 4.17. Finalmente, os

pontos com índice k = 17, 19, 21, 23, 25, 27, 29 e 31 transmitem o seu valor para sete vizinhos, como mostra a Figura 4.18.

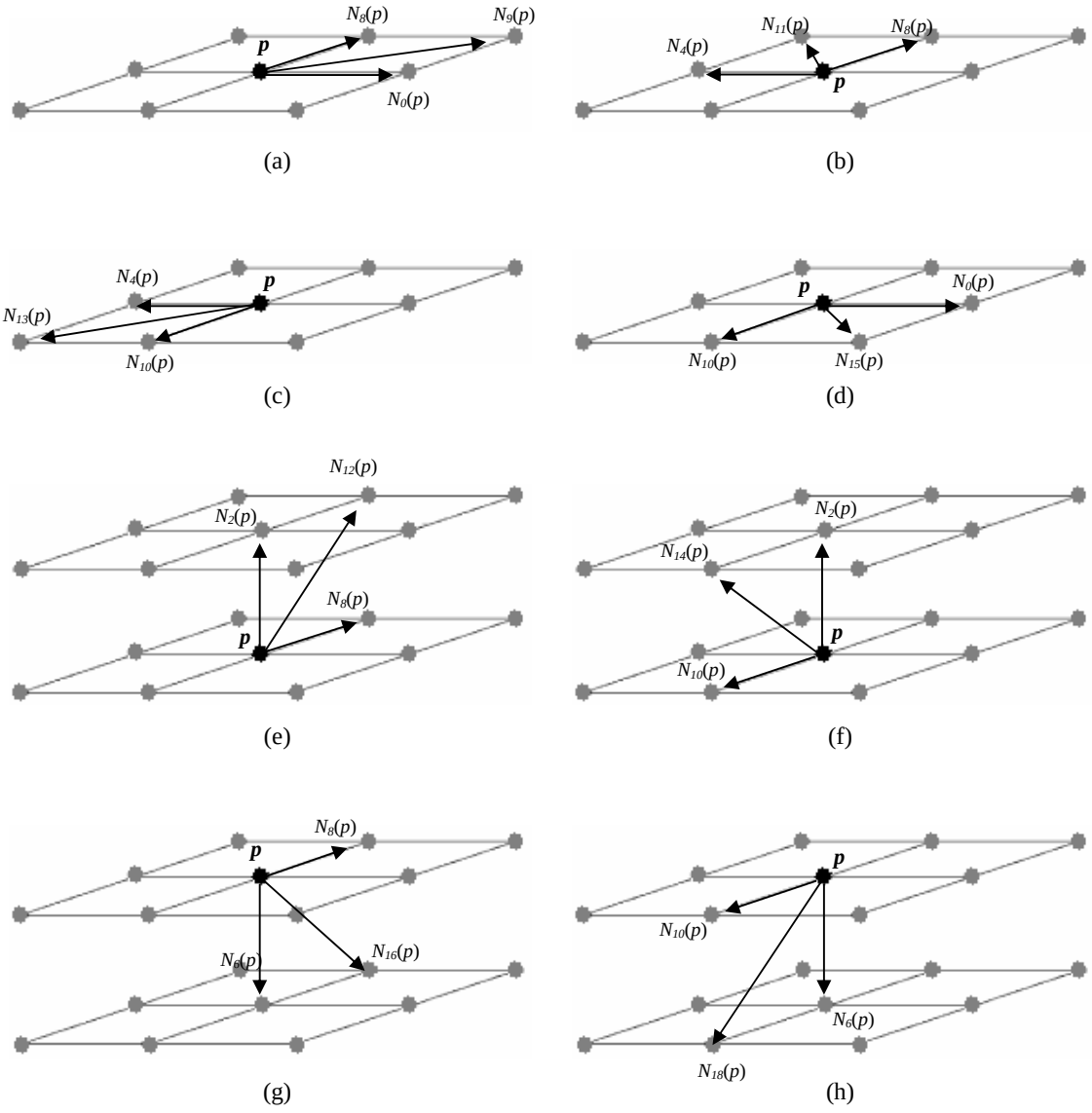


Figura 4.17. Propagação suficiente: pontos que transmitem informação para três vizinhos. Pontos para os quais um ponto p, com índice de direção k = 9 (a), 11 (b), 13 (c), 15 (d), 12 (e), 14 (f), 16 (g) e 18 (h), transmite informação.

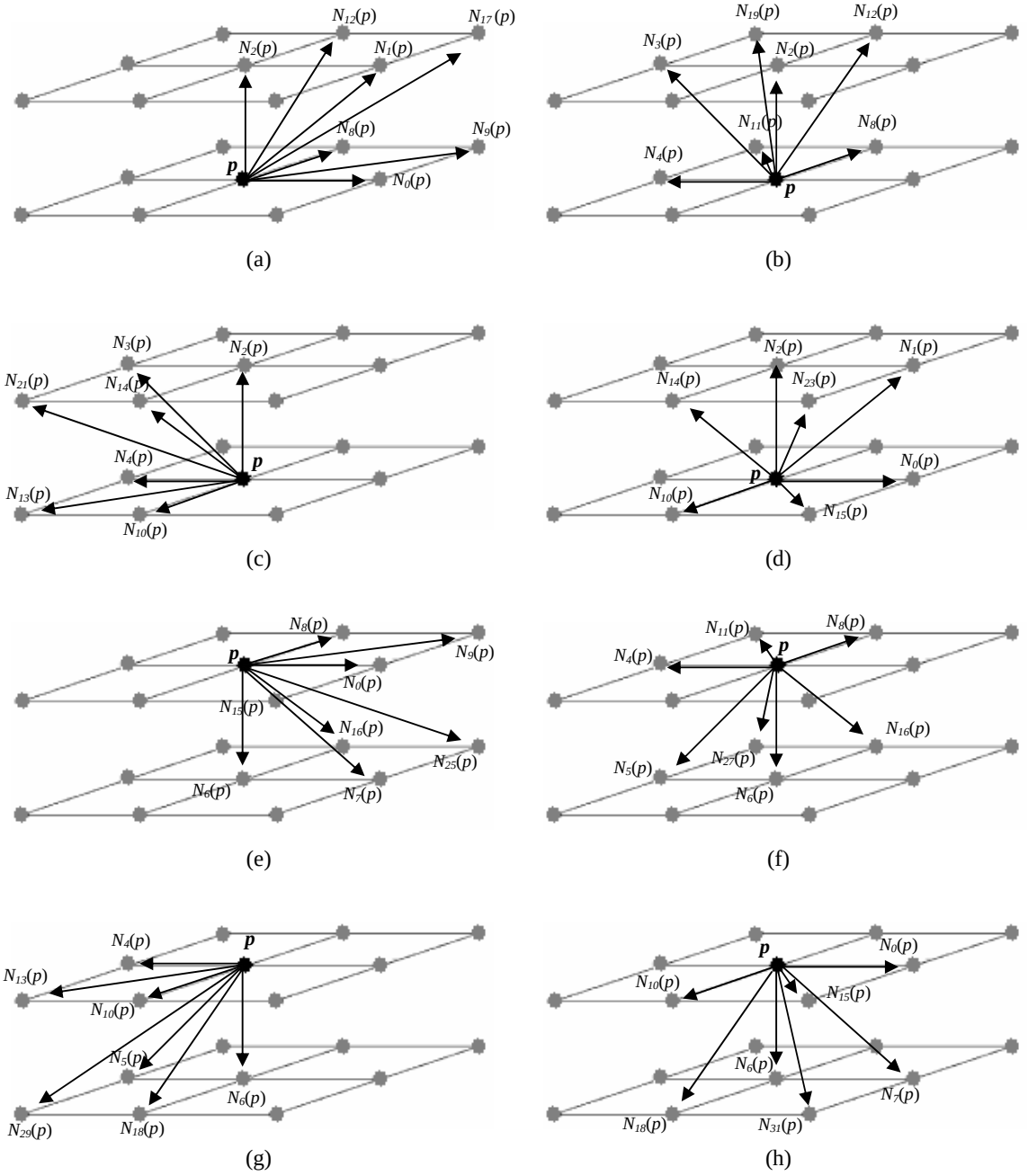


Figura 4.18. Propagação suficiente: pontos que transmitem informação para sete vizinhos. Pontos para os quais um ponto *p*, com índice de direção *k* = 17 (a), 19 (b), 21 (c), 23 (d), 25 (e), 27 (f), 29 (g) e 31 (h), transmite informação.

Na prática, o algoritmo utiliza duas listas encadeadas, *Lista1* e *Lista2*. A primeira armazena os pontos de contorno da iteração corrente e a segunda os pontos de contorno da próxima iteração. Nas listas, além das coordenadas de cada ponto do conjunto de contorno,

são armazenados o seu índice de direção e a sua distância. Durante a propagação o valor de distância armazenado na lista, e não o valor armazenado na imagem, é propagado para os seus vizinhos, porque a distância na imagem pode ter sido alterada depois do ponto ter sido inserido na lista. O algoritmo é descrito a seguir:

1. Inicialize todo ponto de objeto com o valor de distância ∞;
2. Inicialize o *conjunto de contorno* (ou seja, a *Lista1*) fazendo, para cada ponto de fundo *p*:
   - Para k = 0, 2, 4, 6, 8 e 10, se $N_k(p)$ é um ponto de objeto armazene em $N_k(p)$ o valor de distância 1 e insira $N_k(p)$ em *Lista1* com índice de direção igual a *k;*
   - Para k = 1, 3, 5, 7, 9, 11, 12, 13, 14, 15, 16 e 18, se $N_k(p)$ é um ponto de objeto armazene em $N_k(p)$ o valor de distância 2 e insira $N_k(p)$ em *Lista1* com índice de direção igual a *k;*
   - Para k = 17, 19, 21, 23, 25, 27, 29 e 31, se $N_k(p)$ é um ponto de objeto armazene em $N_k(p)$ o valor de distância 3 e insira $N_k(p)$ em *Lista1* com índice de direção *k;*
3. Enquanto *Lista1* $\neq \varnothing$, faça:
   - iteração = iteração + 1;
   - addir = 2 * iteração - 1; adind = 2 * addir; adext = 3 * addir;

   a. Para cada ponto *p* em *Lista1* com índice k = 0, 2, 4, 6, 8 ou 10, faça :
      i. Se $N_k(p) >$ p + addir então
         - Atualize $N_k(p)$ com o valor de distância *p* + addir, e insira $N_k(p)$ em *Lista2* com índice de direção igual a *k*.

   b. Para cada ponto *p* em *Lista1* com índice de direção k = 1, 3, 5 ou 7, faça:
      i. Se $N_k(p) > p$ + adind então
         - Atualize $N_k(p)$ com o valor de distância *p* + adind, e insira $N_k(p)$ em *Lista2* com índice de direção igual a *k*.
      ii. Se $N_{k+1}(p) > p$ + addir então
         - Atualize $N_{k+1}(p)$ com o valor de distância *p* + addir, e insira $N_{k+1}(p)$ em *Lista2* com índice de direção igual a $k + 1$.
      iii. Se $N_{k-1}(p) > p$ + addir então
         - Atualize $N_{k-1}(p)$ com o valor de distância *p* + addir, e insira $N_{k-1}(p)$ em *Lista2* com índice de direção igual a $k - 1$.

   c. Para os demais pontos em *Lista1* faça a propagação como nas figuras 4.13 e 4.14.

   d. Faça *Lista1 = Lista2*; *Lista2* = ∅;

É no passo 3 do algoritmo que acontece a propagação. A propagação de valores dos pontos com índice de direção k = 0, 2, 4, 6, 8 e 10 é mostrada no passo 3*a*, e a dos pontos com índice k = 1, 3, 5 e 7, no passo 3*b*. A propagação de valores dos pontos com os demais índices de direção, passo 3*c*, foi omitida, mas acontece exatamente da mesma forma. Os vizinhos para os quais estes pontos transmitem o seu valor são mostrados nas figuras 4.17 e 4.18. Note que o valor propagado de um ponto p para os seus vizinhos diretos é p + addir; o valor propagado para os seus vizinhos indiretos é p + adind; e para os vizinhos extremos é p + adext. Um ponto p com índice de direção k = 17, por exemplo, transmite o valor p + addir para os seus vizinhos $N_0(p)$, $N_2(p)$ e $N_8(p)$; transmite o valor p + adind para $N_1(p)$, $N_9(p)$ e $N_{12}(p)$; e o valor p + adext para o seu vizinho $N_{17}(p)$. Observe que o os valores addir, adind e adext são calculados com base no número da iteração.

Como o algoritmo utiliza a distância euclidiana ao quadrado, após a sua execução a operação 'raiz quadrada' deve ser aplicada sobre todos os pontos.

4.3.2 Estratégia de paralelização em agregados

Esta seção apresenta a estratégia adotada neste trabalho para a paralelização da TDE 3-D, descrita na seção anterior, em agregados de computadores. Tal estratégia é ilustrada na Figura 4.19. A figura ilustra a estratégia de paralelização em um agregado com quatro nós/processadores.

Primeiro o processador mestre (processador do nó onde a imagem está armazenada) carrega a imagem 3-D do disco para a memória e divide esta imagem em partes, uma para cada processador, cada uma contendo um mesmo número de planos consecutivos da imagem. Destas, uma permanece no nó mestre. Cada uma das outras é enviada para um processador diferente. Após receber a sua parte da imagem, cada processador inicia a execução do algoritmo 3-D de TDE por propagação ordenada, descrito na seção anterior, sobre ela. Este algoritmo consiste de uma série de iterações. Na primeira iteração, *iter* = 1, o *conjunto de contorno* de cada processador contêm apenas os pontos de fundo do pedaço da imagem deste processador, portanto, na primeira iteração estes pontos propagam o seu valor de distância para os seus vizinhos. Os vizinhos cujas distâncias são alteradas/atualizadas são inseridos no conjunto de contorno da próxima iteração. Depois da primeira iteração, inicia-se a segunda, *iter* = 2, e assim sucessivamente. A propagação do valor de cada ponto é feita respeitando as regras descritas na seção anterior. Por exemplo, um ponto com índice de direção k = 0, 2, 4, 6, 8 ou 10 transmite o seu valor para apenas um vizinho.

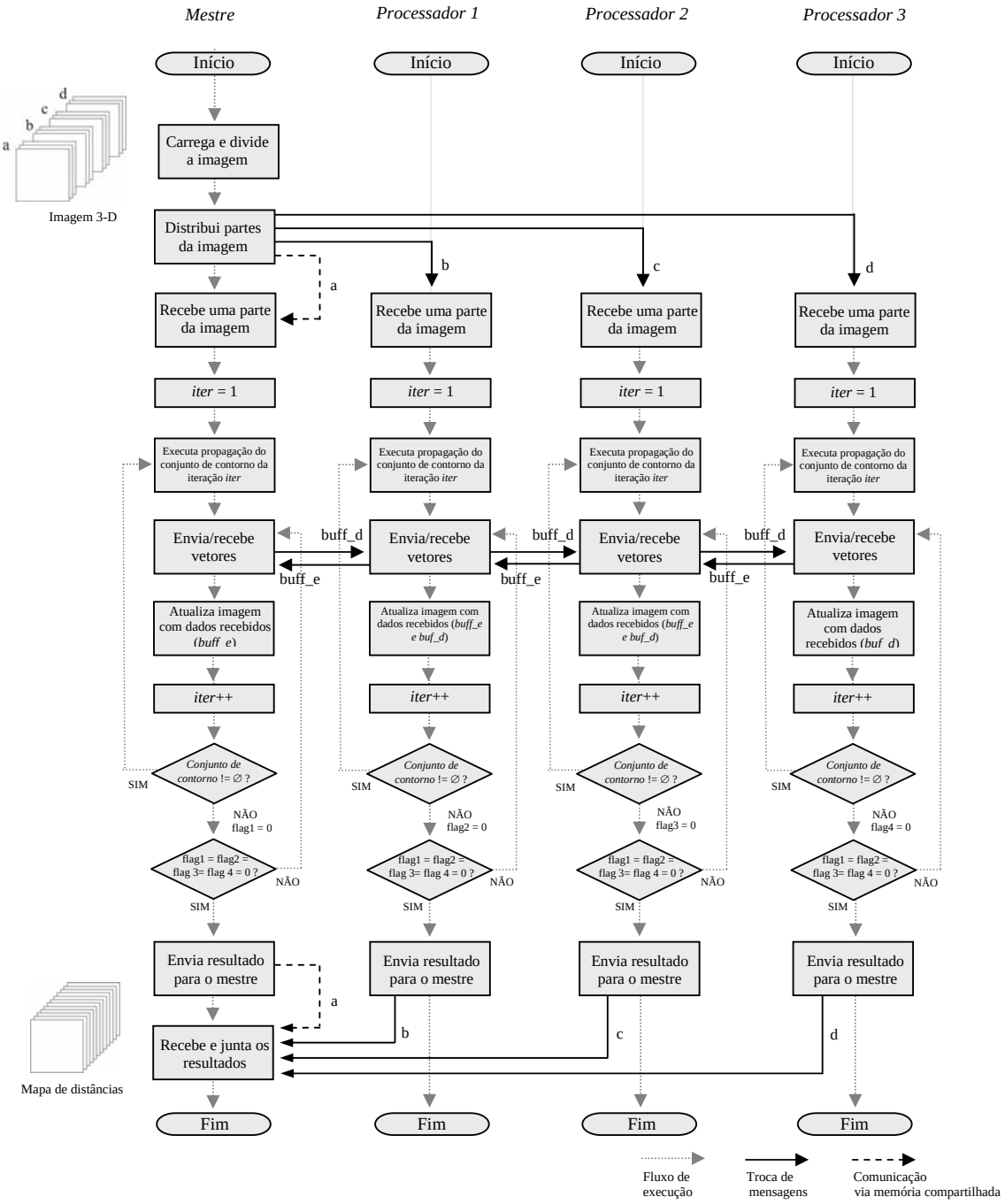


Figura 4.19. Estratégia de paralelização do algoritmo 3-D de TDE por propagação ordenada em agregados.

Note no entanto que, como a imagem foi dividida/distribuída entre os processadores/nós do agregado pode ser que um ponto do conjunto de contorno de um processador na iteração *i* tente propagar o seu valor para outro que não está no pedaço da imagem deste processador. Para resolver este problema em cada processador são utilizados dois vetores auxiliares, *buff_e* e *buff_d*. Toda vez que um processador tenta propagar o valor

de um ponto para outro que está na parte da imagem do processador à sua esquerda[27], as coordenadas deste ponto, o índice de direção e a distância propagada são armazenadas em *buff_e*. Da mesma forma, quando um processador tenta propagar o valor de um ponto para outro que está na parte da imagem do processador à sua direita, as coordenadas, o índice de direção e a distância são armazenadas em *buff_d*. Ao término da iteração *i* cada processador envia os vetores *buff_e* e *buff_d* para o processador à sua esquerda e para o processador à sua direita, respectivamente. O processador que recebe *buff_e* e/ou *buff_d* verifica se o valor de distância de cada ponto contido nos vetores é menor que o valor de distância deste ponto no seu pedaço da imagem. Em caso afirmativo o valor de distância na imagem é substituído pelo do vetor e o ponto do vetor é inserido no conjunto de contorno da iteração $i + 1$ deste processador. Após verificar todos os pontos nos vetores cada processador inicia a próxima iteração. Quando o conjunto de contorno da próxima iteração de todos os processadores for vazio o algoritmo de TDE termina. Por isso cada processador depois de executar cada iteração informa aos demais se o seu conjunto de contorno está vazio. Quando isto é verdade cada processador executa a operação 'raiz quadrada' sobre os pontos na sua parte da imagem e então envia esta parte para o processador mestre, que junta as partes de todos os processadores para construir o mapa de distâncias euclidianas final.

4.3.2.1 Detalhes de implementação no agregado utilizado neste trabalho

Esta estratégia de paralelização foi implementada no agregado descrito na Seção 3.2.3.2 usando a linguagem C e a biblioteca MPICH, e compilada com o compilador *gcc* versão 3.3.5. Os processos do programa paralelo foram distribuídos entre os nove[28] nós do agregado de acordo com um esquema *Roudin-Robin* usando uma lista com o número IP dos computadores que fazem parte do agregado, sendo que foi "colocado" um processo por nó.

Inicialmente o processo mestre (processo do nó onde a imagem está armazenada) divide a imagem em partes e comprime cada uma das partes para enviá-las para os outros, porque a rede do agregado é uma rede com baixa largura de banda. Como imagens binárias são compostas apenas de 0s e 1s, elas apresentam altos graus de redundância, que favorecem a compressão. Foi por isso que comprimir e enviar as partes das imagens teste foi significativamente mais rápido que enviá-las sem comprimir. As (i) taxas de compressão, o (ii) tempo de compressão e o (iii) tempo de envio das partes destas imagens foram

[27] Em um agregado com quatro nós/processadores, estes nós são numerados de 1 a 4. Os nós 1 e 3, por exemplo, são, respectivamente, os nós à esquerda e à direita do nó 2.

[28] Diferente da paralelização do algoritmo de Saito e Toriwaki que foi executada com dez nós, esta foi executada com apenas nove, porque um dos nós havia sido removido do agregado.

descritos/discutidos na Seção 4.2.3.1, onde foi apresentada a estratégia de paralelização do algoritmo de Saito e Toriwaki. Neste e naquele algoritmo a imagem é inicialmente dividida da mesma forma, portanto os resultados das compressões foram os mesmos e não serão repetidos aqui.

Assim como na implementação paralela do algoritmo de Saito e Toriwaki, aqui as partes comprimidas são enviadas do mestre para os outros processadores com a função MPI *SEND pronto* e também, como naquela implementação, a imagem é carregada sob demanda, ou seja, a parte que é enviada do processador mestre para o escravo 1 só é carregada para a memória do mestre no momento do envio, sendo então comprimida, enviada e imediatamente removida da memória. Só então a parte do processador 2 é carregada, comprimida, enviada e removida, e assim sucessivamente. Tal estratégia evita a realização de *swapping* no nó mestre.

Após receber a sua parte da imagem, cada processador inicia a primeira iteração, *iter* = 1, sobre ela. Quando o valor de um ponto no pedaço da imagem de um processador é propagado para outro que está no mesmo pedaço da imagem, caso o valor propagado seja menor que o valor corrente do vizinho, o valor do vizinho na imagem é atualizado e ele é inserido no conjunto de contorno da próxima iteração deste processador. Quando o valor de um ponto é propagado para outro que está no pedaço da imagem do processador da esquerda, ele é inserido em *buff_e*. Quando o valor de um ponto é propagado para outro que está no pedaço do processador da direita, ele é inserido em *buff_d*. Ao término da iteração *buff_e* e *buff_d* são enviados para o processador da esquerda e para o processador da direita, respectivamente. O tamanho médio destes vetores, quando processando as imagens teste 500×500×500, são listados na Tabela 4.3.

Tabela 4.3. Tamanho médio, em MB, dos vetores *buff_e* e *buff_d*.

	Imagem com um ponto de fundo	**Imagem da esfera**	**Imagem dos cubos**
Tamanho Médio	0,914 (0,803)	0,377 (1,393)	1,013 (2,240)
Tamanho Máximo	2,276	17,924	13,945
% mensagens maiores que 1 MB	37,5	1,4	21,1

Note que em média o tamanho de *buff_e* e/ou *buff_d* para as imagens do ponto, da esfera e dos cubos randômicos é, respectivamente, 0,914MB, 0,377MB e 1,013MB. Note também que o valor do desvio padrão, apresentado entre parênteses, é pequeno. No entanto, alguns vetores muito grandes apareceram em algumas iterações. Na imagem da esfera, por

exemplo, um vetor de tamanho 17,924MB apareceu. Por isso, optou-se por comprimir os vetores antes de enviá-los. As taxas médias de compressão obtidas quando comprimindo os vetores *buff_e* e *buff_d* são ilustradas no gráfico da Figura 4.20. Note que a menor taxa de compressão foi aproximadamente 60%.

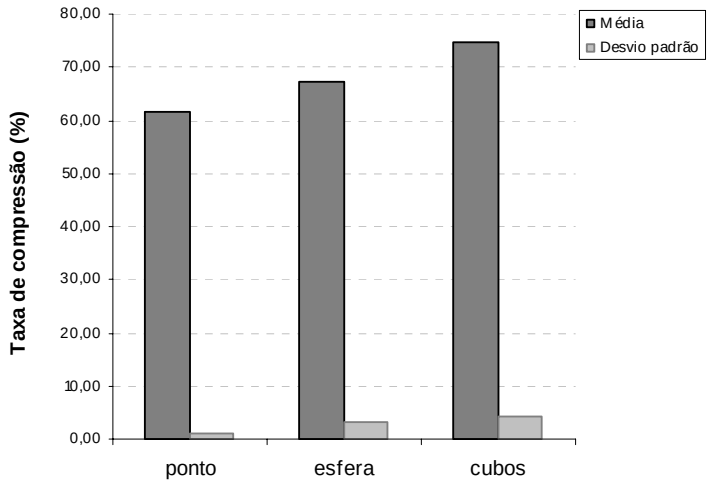


Figura 4.20. Taxa média de compressão dos vetores *buff_e* e *buff_d*.

Apesar das altas taxas de compressão, notou-se que a compressão dos vetores pouco contribuiu para o aumento do desempenho do programa quando processando as imagens teste, uma vez que estes vetores eram, na grande maioria das vezes, pequenos. Na imagem da esfera, por exemplo, apenas 1,4% dos vetores eram maiores que 1MB.

O processador que recebe *buff_e* e/ou *buff_d* faz a descompressão dos mesmos. Então ele verifica se o valor de cada ponto contido em *buff_e* e em *buff_d* é menor que o valor de distância deste ponto no seu pedaço da imagem. Em caso afirmativo o valor de distância na imagem é substituído pelo do vetor e o ponto do vetor é inserido no conjunto de contorno da próxima iteração deste processador.

Note que ao término de cada iteração ocorre uma troca de dados, isto é, a troca dos vetores *buff_e* e *buff_d* entre os processadores. Assim, um processador só inicia a iteração $i + 1$, após receber os vetores *buff_e* e *buff_d* da iteração i dos processadores à sua esquerda e à sua direita, respectivamente. Portanto, todos os processadores executam a iteração i, fazem uma troca de dados, depois executam a iteração $i + 1$, fazem uma nova troca de dados, depois executam a iteração $i + 2$, e assim sucessivamente. No entanto, o tempo que um processador leva para executar uma iteração pode ser bastante diferente dos demais, porque o conteúdo da sua parte da imagem e, portanto, do seu conjunto de contorno, pode ser bem diferente do dos

outros processadores. O número total de iterações, por sua vez, depende do conteúdo da imagem. Para as imagens do ponto, da esfera e dos cubos com 500×500×500 pontos, por exemplo, o número de iterações foi, respectivamente 500, 249 e 56. Apesar do grande número de iterações na imagem do ponto em apenas 8 mensagens os vetores *buff_e* e/ou *buff_d* não estavam vazios. Já na imagem dos cubos 500×500×500 o número de mensagens efetivamente trocadas, ou seja, o número de mensagens onde os vetores *buff_e* e/ou *buff_d* não estavam vazios foi de 464.

Terminada a última iteração cada processador executa a operação 'raiz quadrada' sobre os pontos do seu pedaço da imagem. Então cada processador comprime o seu pedaço da imagem para enviá-lo ao mestre. As taxas médias de compressão obtidas são ilustradas no gráfico da Figura 4.21.

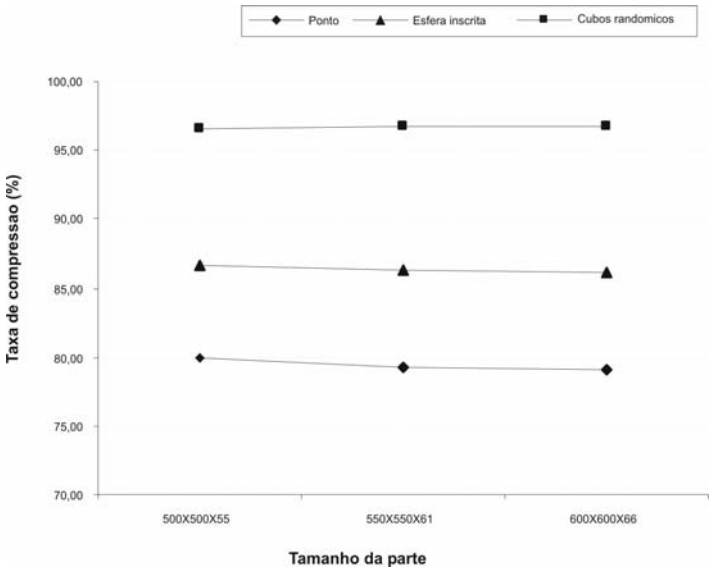


Figura 4.21. Taxa média de compressão das partes das imagens teste. No gráfico são mostradas partes com tamanhos 500×500×55, 550×550×61, 600×600×66 etc., porque as imagens teste tinham tamanhos 500×500×500, 550×550×550, 600×600×600, etc., e o agregado nove nós.

Assim como aconteceu no algoritmo de Saito e Toriwaki, as melhores taxas de compressão foram a da imagem dos cubos, da esfera e do ponto, nesta ordem.

4.3.2.2 Balanceamento de carga

Para verificar o balanceamento (ou não-balanceamento de carga) da implementação desenvolvida foram utilizadas as ferramentas MPE e Jumpshot. A Figura 4.22 mostra um

diagrama gerado por estas ferramentas durante o processamento da imagem dos cubos randômicos 500×500×500 no agregado.

Note que no inicio do programa ocorre um pequeno desbalanceamento de carga em função do tempo inevitavelmente perdido com o carregamento, divisão, compressão e distribuição da imagem entre os processadores. Da primeira até a última iteração, no entanto, todos os processadores estão sempre ocupados. Na figura, cada iteração é representada por um bloco, sendo que foram omitidas algumas iterações/blocos para fins de apresentação. Entre cada bloco existe um espaço que representa o tempo gasto com o envio/recepção dos vetores *buff_e* e *buff_d*.

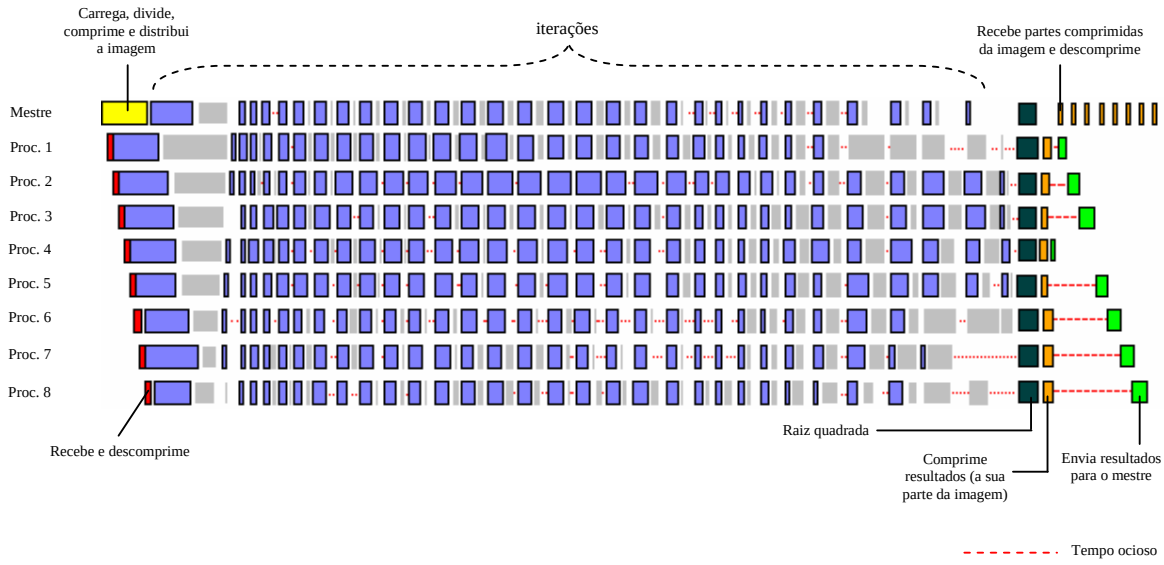


Figura 4.22. Diagrama de execução da imagem dos cubos randômicos 500×500×500 no agregado.

4.3.2.3 Resultados

Esta seção apresenta os resultados obtidos com a implementação paralela do algoritmo de TDE por propagação ordenada no agregado. A Tabela 4.4 traz os tempos médios de execução da implementação do algoritmo seqüencial executando em um nó do agregado, e os tempos médios de execução da implementação paralela executando em nove nós. Para calcular as médias cada implementação foi executada cinco vezes. Na tabela, os valores entre parênteses representam o desvio padrão.

A partir dos tempos de execução dos programas seqüencial e paralelo foram calculados os *speedups*, que são mostrados no gráfico da Figura 4.23.

Tabela 4.4. Tempos de execução, em segundos, do algoritmo seqüencial de TDE por propagação ordenada e da implementação paralela deste algoritmo no agregado.

Tamanho da imagem	Imagem com um ponto de fundo		Imagem da esfera		Imagem dos cubos	
	seqüencial	paralelo	seqüencial	paralelo	seqüencial	paralelo
500×500×500	54 (0)	42 (0)	662 (1)	314 (0)	543 (1)	175 (2)
550×550×550	73 (0)	61 (0)	982 (1)	493 (3)	842 (1)	360 (3)
600×600×600	97 (0)	78 (1)	1632 (5)	789 (14)	1462 (4)	660 (6)

Diferente do que foi feito com a implementação paralela do algoritmo de Saito e Toriwaki, para validar a implementação paralela do algoritmo desta seção foram utilizadas imagens com até 600×600×600 pontos, porque em imagens maiores o tempo de execução do algoritmo seqüencial e da sua implementação paralela começaram a se distanciar significativamente dos tempos do algoritmo seqüencial e paralelo de Saito e Toriwaki, tornando-se inviáveis.

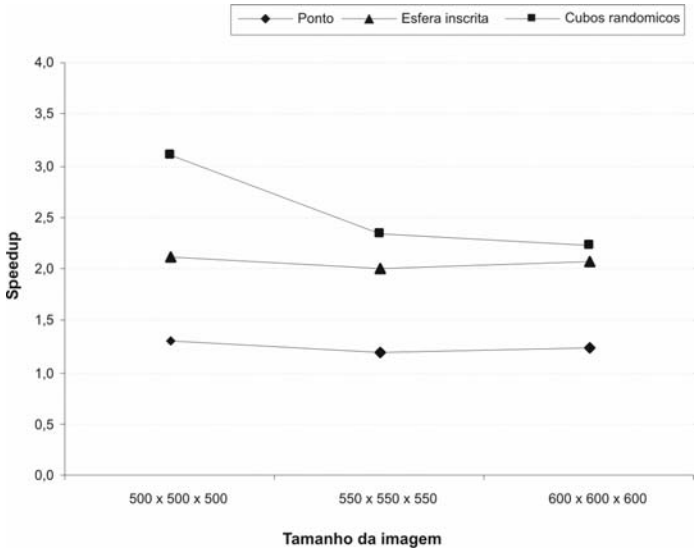


Figura 4.23. *Speedups* obtidos com a implementação paralela do algoritmo de TDE por propagação ordenada no agregado.

4.3.3 Considerações finais

Uma estratégia de paralelização diferente da descrita nesta seção foi testada pelo autor deste trabalho. Nesta estratégia ao invés dos processadores executarem uma iteração de cada vez, e ao término de cada iteração fazerem o envio/recepção dos vetores *buff_e* e *buff_d*, eles executam todas as iterações sobre o seu pedaço da imagem, para só então fazerem o

envio/recepção dos vetores. Como o envio/recepção ocorre depois de todas as iterações, os pontos nos vetores armazenam além da distância propagada, a iteração onde a propagação ocorreu. O processador que recebe estas informações continua a propagação destes pontos no seu pedaço da imagem. Para a propagação destes pontos várias iterações são executadas. Ao término destas iterações ele envia/recebe novamente os vetores *buff_e* e *buff_d*. Isto se repete até que *buff_e* e *buff_d* estejam vazios em todos os processadores.

Esta estratégia de paralelização foi implementada no agregado, mas com todas as imagens apresentou desempenho bem inferior à estratégia descrita anteriormente.

Capítulo 5

Conclusões

5.1 Conclusões

Neste trabalho foram desenvolvidas e implementadas estratégias de paralelização para SMPs e para agregados para o algoritmo 3-D de TDE de Saito e Toriwaki. A estratégia para SMPs foi implementada com a ferramenta de programação paralela OpenMP em um SMP com quatro processadores, onde foram obtidos *speedups* excelentes. A estratégia desenvolvida para agregados foi implementada com a ferramenta MPICH no agregado do grupo de visão cibernética do IFSC-USP. Para imagens com 700×700×700 ou menos pontos os *speedups* ficaram próximos de 2. Estes *speedups* só foram possíveis porque foi utilizada compressão, uma vez que a rede utilizada do agregado, uma *Fast Ethernet*, é uma rede com baixa largura de banda. Para imagens maiores que 700×700×700, no entanto, os *speedups* foram excelentes, ficando entre 5 e 7. Tais *speedups* aconteceram porque o programa seqüencial ao processar estas imagens teve seu tempo de execução aumentado significativamente, por causa da realização de *swapping*, enquanto que no paralelo o *swapping* não aconteceu.

O algoritmo 2-D de TDE por propagação ordenada de Eggers foi estendido para 3-D. Para este algoritmo 3-D foi desenvolvida uma estratégia de paralelização que foi implementada no agregado do grupo de visão cibernética do IFSC-USP. Os *speedups* obtidos utilizando nove nós do agregado foram melhores do que aqueles obtidos com a implementação paralela do algoritmo de Saito e Toriwaki em dez nós. Porém, os tempos de execução da implementação paralela de Saito e Toriwaki foram menores que os tempos de execução da implementação paralela do algoritmo de Eggers, porque a versão seqüencial deste último foi muito mais lenta que a de Saito e Toriwaki.

No agregado, tanto para a implementação paralela do algoritmo de Saito e Toriwaki quanto para a de Eggers a rede foi um gargalo importante que impediu a obtenção de

melhores resultados. Para a implementação paralela do algoritmo de Saito e Toriwaki o maior gargalo foi a baixa largura de banda da rede, uma vez que poucas mensagens com grandes quantidades de dados são trocadas entre os processadores. Para amenizar este problema foi usada, com sucesso, compressão. Entretanto, a compressão/descompressão, e o tempo gasto com ela, poderia ser evitada se uma rede com grande largura de banda fosse utilizada. No caso da implementação do algoritmo de Eggers, o gargalo foi também a alta latência da rede, porque neste algoritmo muitas mensagens pequenas são trocadas entre os processadores a cada iteração.

5.2 Contribuições

As principais contribuições deste trabalho são:

- **Levantamento bibliográfico** atualizado sobre algoritmos de TDE; classificação/categorização destes algoritmos; e descrição detalhada de pelo menos um algoritmo de cada classe.
- **Descrição inédita do algoritmo de Saito e Toriwaki** onde são elucidadas algumas passagens “obscuras” do artigo original.
- **Extensão do algoritmo de Eggers do 2-D para o 3-D**. No seu artigo Eggers apenas comenta a possibilidade de extensão, mas não mostra como fazer isso.
- **Desenvolvimento, implementação e discussão de estratégias de paralelização para os algoritmos 3-D de Saito e Toriwaki em SMPs e em agregados e para a extensão do algoritmo de Eggers em agregados.**
- **Redução do tempo de execução das TDE de Saito e Toriwaki e Eggers.**

Outras contribuições:

- **Validação dos algoritmos.** Fabbri (2004) verificou experimentalmente que os algoritmos 2-D de Saito e Toriwaki e de Eggers são exatos. Por meio dos experimentos realizados no presente trabalho verificou-se que as versões 3-D destes algoritmos também são.
- **Fornecimento de algoritmos eficientes de TDE**. Os algoritmos desenvolvidos neste trabalho já estão sendo disponibilizados aos alunos orientados pelo professor

Odemir. Alguns dos alunos que utilizavam a TDE seqüencial de Costa (2000), agora estão utilizando a de Saito e Toriwaki, que é significativamente mais rápida que aquela. A TDE de Saito e Toriwaki também foi incluída em um software de processamento de imagens desenvolvido pelo professor Odemir e pelos seus orientados.

- **Descrição das ferramentas de programação paralela OpenMP e MPI.**

5.3 Publicações

Um trabalho que valida e compara os principais algoritmos de TDE exata foi desenvolvido por Ricardo Fabbri, em colaboração com o autor deste trabalho. Tal trabalho resultou em um artigo, na forma de *survey*, onde o autor desta dissertação é o segundo autor. Tal artigo está sendo finalizado e será submetido para uma revista internacional. Um artigo com os resultados da paralelização da TDE de Saito e Toriwaki no agregado foi submetido para uma revista internacional. Um artigo com os resultados da paralelização da extensão da TDE de Eggers será, após a defesa desta dissertação, escrito e submetido para outra revista internacional.

Além do artigo submetido e dos que serão submetidos, outros foram publicados:

TORELLI, J. C.; BRUNO, O. M. Uma ferramenta para a realização de medidas de desempenho em programas paralelos. **Scientia,** v.15, n.1, p.1-9, 2004.

TORELLI, J. C.; BRUNO, O. M. Programação paralela em SMPs com OpenMP e POSIX Threads: Um estudo comparativo. In: CONGRESSO BRASILEIRO DE COMPUTAÇÃO, 4., 2004, Itajaí. **Anais ...** Itajaí, SC: Univali, 2004. p.486-491.

TORELLI, J. C.; BRUNO, O. M. Implementação paralela da transformada de distância euclidiana exata. In: WORKSHOP DE TESES E DISSERTAÇÕES, 9., 2004, São Carlos. **Anais ...** São Carlos: ICMC-USP, 2004. p.1-1.

TORELLI, J. C.; BRUNO, O. M. Implementação paralela da transformada exata da distância. In: CONGRESSO DE PÓS-GRADUAÇÃO DA UNIVERSIDADE FEDERAL DE SÃO CARLOS, 2., 2003, São Carlos. **Anais ...** São Carlos: UFSCar, 2003. p.1-1.

5.4 Desenvolvimentos futuros

Diversas atividades podem agora ser desenvolvidas:

- **Incorporar mais imagens teste,** porque é interessante testar o desempenho das implementações seqüenciais e paralelas com outras imagens, e com imagens reais.

- **Estudar o desempenho das implementações paralelas em agregados com redes com alta largura de banda e baixa latência.** No processamento paralelo de imagens 3-D a comunicação entre processos é um gargalo, por causa da grande quantidade de dados manipulados. Como a rede do agregado utilizado neste trabalho era uma *Fast Ethernet*, utilizou-se compressão para reduzir o tempo gasto com comunicação. Resta agora estudar/executar a implementação paralela dos algoritmos de TDE em agregados com redes com alta largura de banda e baixa latência, por exemplo, *Myrinet* e *Infiniband*. Acredita-se que nestas redes a compressão não é necessária e que melhores resultados podem ser obtidos.

- **Paralelizar os algoritmos de TDE em outras arquiteturas**, por exemplo, no computador MIMD com processadores vetoriais disponível no CCE-USP.

- **Paralelizar outros algoritmos de TDE**, especialmente algoritmos de TDE por varredura, pois nenhum algoritmo desta classe foi paralelizado neste trabalho.

Além dos trabalhos que envolvem a paralelização de algoritmos de TDE, outros podem ser desenvolvidos:

- **Avaliação dos principais algoritmos 3-D de TDE.** Fabbri (2004), em colaboração com o autor deste trabalho, desenvolveu um trabalho de comparação de algoritmos 2-D de TDE. Tal trabalho teve como resultado a sua dissertação e um artigo na forma de *survey* que está sendo finalizado e será submetido a uma revista internacional. Dos cinco algoritmos estudados por Fabbri, a versão 3-D de dois deles foram detalhadamente estudadas e implementadas pelo autor deste trabalho. Futuramente, os outros três poderiam ser implementados em 3-D, e uma *survey* de algoritmos 3-D de TDE poderia ser publicada.

- **Mostrar formalmente que os algoritmos de Shih e Wu não são exatos.** Em seu trabalho Fabbri (2004) verificou que o algoritmo 2-D de Shih e Wu (2004a) não é exato. O autor desta dissertação estudou/implementou um algoritmo 3-D de TDE do mesmo autor, publicado em um outro artigo (SHIH; WU, 2004b), e verificou que ele também não é exato.

Referências Bibliográficas

ALMASI, G. S.; GOTTLIEB, A. **Highly parallel computing**. 2ed. Redwood City: Benjamin/Cummings, 1994.

CENAPAD - CENTRO NACIONAL DE PROCESSAMENTO DE ALTO DESEMPENHO. **Introdução ao MPI**. Disponível em: <ftp://ftp.cenapad.unicamp.br/cursos/MPI/>. Acesso em: 28 junho 2003.

CHEN, L.; CHUANG, H. Y. H. An efficient algorithm for complete Euclidean distance transform on mesh-connected SIMD. **Parallel Computing**, v.21, n.5, p.841-852, 1995.

COSTA, L. F. Robust skeletonization through exact Euclidean distance transform and its application to Neuromorphometry. **Real-time imaging**, v.6, p.415-431, 2000.

COSTA, L. F.; BIANCHI, A. G. C. A dimensão da dimensão fractal. **Ciência Hoje**, v.31, n.183, p.40-47, 2002.

COSTA, L. F.; CESAR JUNIOR, R. M. **Shape analysis and classification**: theory and practice. Boca Raton, FL: CRC Press, 2001.

CUISENAIRE, O. **Distance transformations**: fast algorithms and applications to medical image processing. 1999. 213f. Tese (Doutorado) - Laboratoire de telecommunications et teledetection, Université catholique de Louvain, Louvain-la-Neuve, 1999.

CUISENAIRE, O.; MACQ, B. Fast Euclidean distance transformations by propagation using multiple neighborhoods. **Computer Vision and Image understanding**, v.76, n.2, p.163-172, 1999.

_____. Fast and exact signed Euclidean distance transformation with linear complexity. In: INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH AND SIGNAL PROCESSING, 24., 1999, Phoenix. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 1999. p.3293-3296.

CULLER, D. E.; SINGH, J. P.; GUPTA, A. **Parallel computer architecture**: a hardware/software approach. San Francisco: Morgan Kaufmann Publishers, 1999.

DANIELSSON, P. E. Euclidean distance mapping. **Computer Graphics and Image Processing**, v.14, p.227-248, 1980.

DIJKSTRA, E. W. A note on two problems in connection with graphs. **Numerische Math**, v.1, p.269-271, 1959.

DUNCAN, R. A survey of parallel computer architectures. **IEEE Computer**, v.23, n.2, p.5-16, 1990.

EGGERS, H. Two fast Euclidean distance transformations in Z^2 Based on Sufficient Propagation. **Computer Vision and Image Understanding**, v.69, n.1, p.106-116, 1998.

FABBRI, R. **Comparação e desenvolvimento de algoritmos de transformada de distância euclidiana e aplicações**. 2004. 73f. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 2004.

FALVO, M; BRUNO, O. M. **Um algoritmo para otimizar a geração do Gabarito da Transformada da Distância Euclidiana Exata**. São Carlos: ICMC-USP, 2003. (Relatório Técnico, 218).

FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE Transactions on Computers**, v.C-21, n.9, p.948-960, 1972.

FOSTER, I. **Designing and building parallel programs**: concepts and tools for parallel software engineering. Reading, Mass.: Addison-Wesley, 1995.

GE, Y.; FITZPATRICK, J. On the generation of skeletons from discrete Euclidean distance maps. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v.18, n.11, p.1055-1066, 1996.

HENNESSY, J. L; PATTERSON, D. A. **Computer architecture**: a quantitative approach. 3ed. San Francisco: Morgan Kaufmann Publishers, 2003.

HIRATA, T. A unified linear-time algorithm for computing distance maps. Inf. Process Lett., v.58, n.3, p.129-133, 1996.

HWANG, K. **Advanced computer architecture**: parallelism, scalability, programmability. New York: Mcgraw-Hill, 1993.

HUANG, C.; MITCHELL, O. A Euclidean distance transform using grayscale morphology decomposition. **IEEE Transactions on Pattern Analysis and Machine Inteligence**, v.16 p.443-448, 1994.

KNUTH, D. E. **The art of computer programming:** sorting and searching. 2ed. Reading, Mass.: Addison-Wesley, 1998.

KOLOUNTZAKIS, M. N.; KUTULAKOS, K. N. Fast computation of the Euclidean distance maps for binary images. **Inf. Process. Lett.**, v.43, n.4, p.181-184, 1992

LLNL - LAURENCE LIVEMORE NATIONAL LABORATORY. **OpenMP**. Disponível em: <http://www.llnl.gov/computing/tutorials/openMP/>. Acesso em: 25 setembro 2003.

LEE, Y.-H et al. Parallel computation of exact Euclidean distance transform. **Parallel Computing**, v.22, n.2, p.311-325, 1996.

LEE Y.-H., HORNG, S.-J., KAO T.-W.; CHEN, Y.-J. Parallel computation of the Euclidean distance transform on the mesh of trees and the hypercube computer. **Computer Vision and Image Understanding**, v.68, n.1, p.109-119, 1997.

LOTUFO, R. A.; FALCÃO, A. A.; ZAMPIROLLI, F. A. Fast Euclidean distance transform using a graph-search algorithm. In: BRAZILIAN SYMPOSIUM ON COMPUTER GRAPHICS AND IMAGE PROCESSING, 13., 2000, Gramado. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2000. p.269-275.

LOTUFO, R. A.; ZAMPIROLLI, F. A. Fast multi-dimensional parallel Euclidean distance transform based on mathematical morphology. In: BRAZILIAN SYMPOSIUM ON COMPUTER GRAPHICS AND IMAGE PROCESSING, 14., 2001, Florianópolis. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2001. p.100-105.

MARCARI JUNIOR, E. **Classificação e comparação de ferramentas para análise de desempenho de sistemas paralelos**. 2002. 113f. Dissertação (Mestrado em Engenharia Elétrica) - Faculdade de Engenharia, Universidade Estadual Paulista, Ilha Solteira, 2002.

MAURER JR, C. R.; QI, R.; RAGHAVAN, V. A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v.25, n.2, p.265-270, 2003.

MPIF - MESSAGE PASSING INTERFACE FORUM. **MPI**: A Message-Passing Interface Standard. 1995. Disponível em: <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>. Acesso em: 14 Outubro 2003.

_____. **MPI-2**: Extensions to the Message-Passing Interface. 1997. Disponível em: <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>. Acesso em: 14 Outubro 2003.

NCSA - NATIONAL CENTER FOR SUPERCOMPUTING APPLICATON AT THE UNIVERSITY OF ILLINOIS. **Introduction to OpenMP**. Disponível em: <http://webct.ncsa.uiuc.edu:8900/public/OPENMP/index.html >. Acesso em: 17 agosto 2003.

_____. **Introduction to MPI**. Disponível em: <http://webct.ncsa.uiuc.edu:8900/public/MPI/index.html >. Acesso em: 17 agosto 2003.

PAGLIERONI, D. W. Distance transforms: properties and machine vision applications. **CVGIP: Graphical Models and Image Processing**, v.54, n.1, p.56-74, 1992.

PARKER, R. **Algorithms for image processing and computer vision**. New York: John Wiley, 1997.

PASIN, M.; KREUTZ, D. L. Arquitetura e administração de aglomerados. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, 3., 2003, Santa Maria. **Anais...** Santa Maria: SBC, 2003. p.3-34.

PATTERSON, D. A.; HENNESSY, J. L.; GOLDBERG, D. **Computer architecture**: a quantitative approach. 2ed. San Francisco: Morgan Kaufmann Publishers, 1996.

PAVLIDIS, T. **Algorithms for Graphics and Image Processing**. Berlin: Springer, 1982

QUINN, M. J. **Parallel computing**: theory and practice. 2ed. New York: McGraw-Hill, 1994.

ROSENFELD, A.; PFALTZ, J. L. Sequential operations in digital picture processing. **Journal of the ACM**, v.13, n.4, p.471-494, 1966.

_____. Distance functions on digital pictures. **Pattern Recognition**, v.1, n.1, p.33-61, 1968

RUSS, J. C. **The image processing handbook**. 4ed. Boca Raton: CRC Press, 2002.

SAITO, T.; TORIWAKI, J. I. New algorithms for Euclidean distance transformation of an n-dimensional digitized picture with applications. **Pattern Recognition**, v.27, n.11, p.1551-1565, 1994.

SHI, F. Y.; MITCHELL, O. A mathematical morphology approach to Euclidean distance transformation. **IEEE Transactions on Pattern Analysis and Machine Inteligence**, v.1 p.197-204, 1992**.**

SHIH, F. Y.; PU, C. C. A Skeletonization Algorithm by Maxima Tracking on Euclidean distance transform. **Pattern Recognition**, v.28, n.3, p.331-341, 1995.

SHIH, F. Y.; WU, Y.-T. A Fast Euclidean distance transformation in two scans using a 3×3 neighborhood. **Computer Vision and Image Understanding**, v.93, n.2, p.195-205, 2004.

_____. Three-dimensional Euclidean distance transformation and its application to shortest path planning. **Pattern Recognition**, v.37, n.1, p. 79-92, 2004.

SOUZA, M. A. **Avaliação das rotinas de comunicação ponto-a-ponto do MPI.** 1996. 152f. Dissertação (Mestrado em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, 1996.

SOUZA, P. S. L. **MPI:** um padrão para ambientes de passagem de mensagens. 1997. 12f. Monografia de qualificação (Doutorado em Física) - Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, 1997.

STALLINGS, W. **Arquitetura e organização de computadores**: projeto para o desempenho. 5ed. São Paulo: Prentice Hall, 2002.

TAKALA, J. H.; VIITANEN, J. O. Distance transform algorithm for Bit-Serial SIMD architectures. **Computer Vision and Image Understanding**, v.74, n.2, p.150-161, 1999.

TANENBAUM, A. S. **Modern operating systems**. 2ed. Upper Saddle River, N.J.: Prentice Hall, 2001.

TANENBAUM, A. S.; GOODMAN, J. R. **Structured computer organization**. 4ed. Upper Saddle River, N.J.: Prentice Hall, 1999.

TRAVIS, A. J.; HIRST, D. J.; CHESSON, A. Automatic classification of plant cells according to tissue type using anatomical features obtained by the distance transform. **Annals of Botany**, v.78, p.325-331, 1996.

TRICOT, C. **Curves and Fractal Dimension**. New York: Springer-Verlag, 1995.

VICENT, L.; SOILLE, P. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v.13, n.6, p.583-598, 1991.

YAMADA, H. Complete Euclidean distance transformation by parallel operation. In: INTERNATIONAL CONFERENCE ON PATTERN RECOGNITION, 7., 1984, Montreal. **Proceedings …** [S.l.: s.n.], [1984]. p.69-71.

YE, Q. Z. The signed euclidean distance transform and its applications. In: INTERNATIONAL CONFERENCE ON PATTERN RECOGNITION, 9., 1988, Roma. **Proceedings …** [S.l.: s.n.], [1988]. p.495-499.

ZAMPIROLLI, F. A. **Transformada de distância por morfologia matemática**. 2003. 133f. Tese (Doutorado em Engenharia Elétrica) - Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, Campinas, 2003.

ZEILINSKY, A. A mobile robot navigation exploration algorithm. **IEEE Transactions of Robotics and Automation**, v.8, n.6, p.707-717, 1992.