3

Análise da Complexidade de Algoritmos Paralelos

Tiarajú Asmuz Diverio *(UFRGS, diverio@inf.ufrgs.br)* [1]
Laira Vieira Toscani *(UFRGS,UNILASALLE, laira@inf.ufrgs.br)* [2]
Paulo A. S. Veloso *(PUCRJ, UFRJ)* [3]

Resumo:

Neste trabalho, apresentam-se os conceitos básicos de complexidade, incluindo noções de complexidade de algoritmos, de medidas de complexidade, de métodos de cálculo de complexidade, e de programação paralela. Esses conceitos são aplicados na análise do desenvolvimento de algoritmos paralelos baseados em memória compartilhada e em troca de mensagens. São considerados exemplos clássicos, como operações com matrizes e classificação de listas. Para ambos os exemplos são considerados a complexidade do problema, uma solução seqüencial de algoritmo onde a complexidade é determinada e versões paralelas para as duas metodologias ou formas de programação. Por fim, apresentam-se algumas questões e considerações sobre análise de complexidade de algoritmos paralelos, seus impactos e suas vantagens.

[1] Doutor em Ciência da Computação e Mestre em Ciência da Computação junto ao PPGC da UFRGS. Licenciado em Matemática pela UFRGS. Professor e Pesquisador do Instituto de Informática da UFRGS. Orientador de Mestrado e Doutorado do PPGC da UFRGS. Editor da Série Livros Didáticos do Instituto de Informática da UFRGS. Áreas de Interesse: Teoria da Computação, Fundamentos da Computação, Processamento de Alto Desempenho e Matemática da Computação.

[2] Doutora em Ciência da Computação pela PUC/RJ. Mestre em Ciência da Computação junto ao Curso de Pós-Graduação em Ciência da Computação da UFRGS. Professora do Instituto de Informática da UFRGS e Chefe do Departamento de Informática Teórica da UFRGS (2000-2001). Atualmente, leciona na UNILASALLE. Áreas de Interesse: Complexidade de Algoritmos e Algoritmos de Aproximação.

[3] Doutor em Ciência da Computação e Mestre em Matemática pela Universidade da Califórnia, Berkeley. Mestre em Engenharia de Sistemas pela COPPE-UFRJ. Professor Titular da Pontifícia Universidade Católica do Rio de Janeiro e da Universidade Federal do Rio de Janeiro. Área de Interesse: Lógica, Fundamentos da Computação, Métodos de Programação e Análise de Algoritmos.

Instituto de Informática e PPGC da UFRGS, Cx. Postal.15064, 91501-970 Porto Alegre RS Brasil

3.1 Introdução

Têm-se constatado a crescente oferta de equipamentos que proporcionam ambientes de programação paralela, como os agregados de computadores, conhecidos como *clusters*. Reúnem-se várias máquinas, como computadores pessoais (PCs), através de uma rede de comunicação de alto desempenho, tendo assim um ambiente de programação paralela ou distribuída. Nesses ambientes, o poder computacional tem aumentado, tanto pelo aumento da velocidade de processamento dos processadores quanto pelo aumento do número de processadores integrados ou agregados ao sistema.

O crescente avanço tecnológico que permite a criação dessas máquinas cada vez mais rápidas, pode, ingenuamente, confundir usuários sobre a importância da complexidade de algoritmos. Entretanto, o que se observa é exatamente o inverso, pois máquinas mais rápidas, possibilitam a resolução de problemas maiores, e é a complexidade de algoritmos que determina o novo tamanho máximo do problema a ser resolvível. O impacto da nova máquina, no caso da resolução de problemas onde os algoritmos são menos eficientes, pode ser muito pequeno, como será ilustrado neste texto.

Apesar dessa crescente oferta de tecnologia, o avanço da área de programação paralela e distribuída, ao nosso ver, tem sido lento. Isso se deve, em parte, a falta de uma cultura de ensino de programação paralela e distribuída. Na grande maioria dos cursos de Bacharelado em Ciência da Computação existentes no estado do Rio Grande do Sul, os conteúdos que servem de base para a formação de pessoal em programação paralela, estão diluídos entre várias disciplinas do curso. São poucos os cursos que têm disciplinas específicas de arquiteturas paralelas, de noções de computação paralela ou de programação paralela. Mas, infelizmente, nenhum curso de graduação tem a disciplina de análise da complexidade de algoritmos paralelos. Nesse minicurso, pretende-se introduzir o tema e mostrar alguns dos benefícios que se pode explorar dessa disciplina.

Sabendo-se que um curso desses necessita certa base, como por exemplo: noção de arquiteturas paralelas, noções de complexidade de algoritmos e certa informação sobre estruturas de dados, será apresentado, inicialmente, uma revisão introdutória desses conteúdos. Uma vez feita essa revisão, pretende-se introduzir duas formas de programação paralela, que são através de memória compartilhada e através de troca de mensagens. Essas duas formas, têm-se evidenciado como metodologias básicas de desenvolvimento de algoritmos em ambientes paralelos e distribuídos.

Foram escolhidas duas aplicações clássicas: operações com matrizes e classificação de listas, (uma numérica e outra não), para demonstrar a análise da complexidade do problema e de algoritmos paralelos. Inicialmente, os problemas são analisados com a finalidade de estabelecer a complexidade do problema, a seguir é ilustrada uma solução seqüencial e determinada sua complexidade, então são desenvolvidos algoritmos paralelos baseados em memória compartilhada e em troca de mensagens. Por fim, são comparadas as duas versões de algoritmos paralelos.

3.1.1 Tipos de processamento

Inicialmente, é necessário caracterizar a forma de se executar tarefas conhecida como processamento. Existem diferenças entre autores na formalização dos tipos de processamento. Por isso, a seguir, será apresentada a terminologia adotada nesse texto. O processamento pode ser *seqüencial, pipeline, vetorial, paralelo e distribuído*.

- ***Processamento seqüencial***. O processador lê um único fluxo de dados por vez e realiza uma operação por unidade de tempo, ou seja, não se executa mais que uma operação por unidade de tempo.
- ***Processamento pipeline***. *Pipeline* é a divisão de uma função genérica em uma seqüência de *k* sub-funções que podem ser implementadas por *k* módulos de *hardware* dedicados e autônomos denominados estágios. Cada estágio é capaz de receber dados do estágio anterior, operá-los e transmitir o resultado para o estágio seguinte. Dessa forma, sucessivas execuções da função podem ser conduzidas pelas sub-funções, operando por superposição (*overlap*), resultando no processamento *pipeline*.
- ***Processamento vetorial***. Substitui-se um laço do programa que usa instruções escalares por um laço que usa instruções vetoriais, levadas a cabo por um *hardware* específico, implementado por *pipeline*. O processamento vetorial difere no fato de que os *pipelines* evoluíram para instruções vetoriais.
- ***Processamento paralelo***. É uma forma eficiente de processar informação a qual enfatiza a exploração de eventos concorrentes na computação do processo. Pode-se caracterizar o *processamento paralelo* quando existe um conjunto de máquinas sendo controladas por um único sistema operacional. A realização de tarefas é levada a cabo por determinação desse sistema que diz quem realiza o que.
- ***Processamento distribuído***. No processamento distribuído há um conjunto de máquinas, cada uma com seu próprio sistema operacional. Existem diversos processadores, diferentes ou não, utilizando uma rede de comunicação. Os processadores trabalham cooperativamente para que os recursos dispersos sejam utilizados na prestação de serviços. Esses diversos processadores devem controlar os seus próprios recursos.

Nesse texto, serão considerados o processamento seqüencial (execução de algoritmos seqüenciais), processamento paralelo no caso da execução de programas com memória compartilhada e o processamento distribuído, que se utiliza da trocas de mensagens para se comunicar.

3.1.2 Análise da algoritmos

Um dos conceitos mais importantes da computação é o de *algoritmo*. A palavra *algoritmo* vem de *Abu al Khwarizmi*, que segundo a história, foi o primeiro a fazer um algoritmo, isso em cerca de 800 D.C. Esse conceito foi formalizado antes de 1930, antes de surgir o primeiro computador. É uma descrição passo a passo de como o problema é solucionado. A descrição deve ser finita e os passos bem definidos, sem ambigüidade.

Por outro lado, dizer que um problema é algoritmicamente solucionável significa, informalmente, que existe um algoritmo tal que, para qualquer entrada, se lhe for dado todo o *tempo* e *espaço* desejado, ele produzirá a resposta correta. Mas ser computável não é suficiente, tem-se que saber o limite do seu custo. Um dos objetivos da análise da complexidade é definir que condições adicionais precisam ser impostas ao problema tal que o custo de sua solução possa ser "*a priori*" limitado e computacionalmente interessante.

Antes de se definir como realizar a análise de algoritmos, deve-se ter em mente o que esperar da análise. Traub (em [TRA73]) apresenta algumas razões para se estudar análise de algoritmos.

- A seleção de algoritmos eficientes é um objetivo central na Ciência da Computação e é um problema de otimização multidimensional, onde uma dessas dimensões é a complexidade de algoritmos.
- Resultados da análise da complexidade ajudam a determinar a estrutura dos dados.
- Limites inferiores da complexidade do problema nos dão uma hierarquia natural baseada nas dificuldades intrínsecas do problema.
- Complexidade de algoritmos é uma teoria matematicamente interessante e satisfatória.

Neste item, ainda, são enumeradas algumas questões importantes quanto à complexidade paralela, das que foram levantadas pelo professor R. Terada durante a VII Escola de Computação, realizada em São Paulo no ano de 1990 [TER90].

- Quais os fatores que determinam a complexidade de algoritmos paralelos?
- Que tipo de problemas computacionais são ou não são solucionáveis eficientemente em paralelo?
- Os algoritmos paralelos são completamente diferentes dos melhores algoritmos seqüenciais para o mesmo problema?
- Quais são as cotas superiores mínimas e as cotas inferiores máximas para complexidade paralela de tempo e de espaço de problemas básicos como: ordenação e operações sobre matrizes?

O ensaio das respostas dessas perguntas, encontra-se no livro do professor Terada da Escola de Computação. Aqui, nesse texto, foram feitas algumas considerações iniciais para introduzir e motivar o estudo.

3.2 Noções de arquiteturas

Apesar dos *sistemas paralelos* serem constituídos por vários processadores, existem várias maneiras diferentes nas quais o *hardware* pode ser organizado. Deve-se considerar o tipo e número de processadores, o tipo de memória, como estão interconectados, seu esquema correspondente de comunicação, gerenciamento, sincronização e operações de entrada e saída. Através dessas questões, podem-se estabelecer diferentes classificações e caracterizações de diferentes máquinas.

Entre as diferentes classificações de máquinas, nenhuma atingiu tamanha notoriedade quanto a proposta por Flynn [FLY72]. Em 1966, Flynn propôs uma classificação das arquiteturas em quatro categorias de máquinas conforme a multiplicidade do fluxo de dados e de instruções. Por ser bastante simples, essa classificação é a mais aceita pela comunidade científica. Segundo Flynn, o ponto principal do processo de um computador é a execução de um conjunto de instruções sobre um conjunto de dados.

A palavra *fluxo* é empregada para descrever uma seqüência de instruções ou de dados, executada num único processador. Portanto, um fluxo de instruções é uma seqüência de instruções executadas por uma máquina, enquanto que um fluxo de dados é uma seqüência de dados (de entrada, de resultados parciais ou intermediários) utilizados pelo fluxo de instruções. A classificação proposta por Flynn leva em conta a forma pela qual é executada uma instrução em um conjunto de dados, ou seja:

- SISD - Single Instruction stream Single Data stream;
- SIMD - Single Instruction stream Multiple Data stream;
- MISD - Multiple Instruction stream Single Data stream;
- MIMD - Multiple Instruction stream Multiple Data stream.

A categoria SISD tem fluxo de instrução e de dados únicos. São máquinas baseadas nos princípios de Von Neumann. As instruções são executadas seqüencialmente, mas podem ser sobrepostas nos seus estágios de execução. Exemplos são máquinas *pipeline*.

A categoria SIMD tem um único fluxo de instruções com vários fluxos de dados. Correspondem aos processadores matriciais (*array*), paralelos e associativos, também chamados de arranjos paralelos e associativos. Nesse tipo de arquitetura, vários elementos de processamento são supervisionados por uma única unidade de controle que encaminhará a mesma instrução para execução nos elementos sobre seus dados. Nessas arquiteturas, portanto, uma única operação é executada simultaneamente por diversos elementos de processadores sobre dados distintos. A memória pode ser dividida em módulos vinculados a cada elemento de processamento ou então ser de acesso global. Exemplos são máquinas matriciais.

A categoria MISD caracteriza-se por ter múltiplos fluxos de instruções e um único fluxo de dados. Esse tipo de arquitetura não existe na prática, mas corresponderia a uma máquina que possuísse vários elementos de processamento recebendo instruções distintas de várias unidades de controle, que processariam o mesmo fluxo de dados.

A categoria MIMD caracteriza-se por ter múltiplos fluxos de instruções com múltiplos fluxos de dados. Nessa arquitetura, cada unidade de controle possui sua unidade de processamento, executando instruções sobre um conjunto de dados de forma independente. São máquinas capazes de executar simultaneamente diversas instruções, sobre dados distintos. A maioria dos sistemas multiprocessadores são desse tipo. A interação ou comunicação entre os processadores é obtida através da memória global ou através de um sistema de memórias gerenciado por um único sistema operacional.

A classe das máquinas MIMD pode ser subdividida em várias subclasses de acordo com outras características. Por exemplo, pode-se considerar o *tipo de acesso a memória*, se ela é compartilhada ou não. Máquinas com memória compartilhada são usualmente conhecidas como *multiprocessadores*. Máquinas que não têm memória compartilhada são ditas, algumas vezes, *multicomputadores*. A diferença essencial entre elas é que em um *multiprocessador* existe um único espaço de endereçamento virtual o qual é compartilhado por todos processadores. Já nos *multicomputadores*, cada máquina tem sua própria memória.

Outra característica a ser considerada na classificação das máquinas MIMD é o *acoplamento*. Acoplamento diz respeito ao tempo de espera de uma mensagem passada por outro computador e a taxa de transferência. As máquinas podem ser *fortemente acopladas* ou *fracamente acopladas*:

- *fortemente acopladas*. O tempo de espera de uma mensagem enviada por outro computador é pequeno e a taxa de transferência de dados é alta, ou seja o número de *bits* por segundo que podem ser transferidos é grande; geralmente, essas máquinas são úteis em sistemas paralelos;
- *fracamente acoplados*. O tempo de espera para ser lida uma mensagem enviada por outro computador é grande e a taxa de transferência de dados é baixa; geralmente são úteis em sistemas distribuídos.

Em geral, multiprocessadores tendem a ser mais fortemente acoplados do que multicomputadores, uma vez que devem modificar dados em memórias de alta velocidade.

Em máquinas com memória compartilhada, existem vários processadores podendo acessar um único espaço de memória, ou seja, os processadores compartilham a mesma memória. Nesse tipo de máquina, o sistema operacional será executado em apenas um processador, o acesso a dispositivos de entrada e saída pode ser feito por

todos os processadores ou somente por alguns, esse tipo de memória é encontrado em multiprocessadores. Os multiprocessadores têm como ponto positivo a facilidade de programação, pois a comunicação entre eles é feita através de memória compartilhada, e a forma como é implementado esse compartilhamento é transparente para os processadores. Algumas das dificuldades dos multiprocessadores são: a complexidade do hardware, a dificuldade de escalabilidade e a redução de desempenho devido à contenção dos processadores.

Apesar de compartilhar um único espaço de endereçamento, a memória, em um multiprocessador, pode se encontrar centralizada ou distribuída entre os diversos processadores. Quando a memória compartilhada é centralizada, diz-se que é uma máquina de acesso uniforme à memória (*UMA – Uniform Memory Access*). Quando a memória compartilhada é distribuída entre os diversos processadores, diz-se que é uma máquina de acesso não uniforme à memória (*NUMA – Non-Uniform Memory Access*). Em outra organização de memória, o espaço de endereçamento é dividido em linhas de *cache* (com boa capacidade de armazenamento) e as linhas podem migrar entre os diversos processadores, não estando associadas a um processador específico (*COMA – Cache Only Memory Access*).

Existe uma diversidade muito grande de multicomputadores, caracterizados pela topologia, pelos esquemas de chaveamento e pelos algoritmos de roteamento utilizados, o que dificulta a criação de uma taxonomia. Tanenbaum [TAN95] classifica os multicomputadores em Processadores Massivamente Paralelos (*MPPs – Massively Parallel Processors*) e c*lusters* de Estações de Trabalho (*COW – Clusters of Workstations*). Além dos *clusters* de estações de trabalhos citados na taxonomia de Tanembaum, também existem os *clusters* de PCs.

3.3 Complexidade de problemas e algoritmos

O objetivo da análise de algoritmos é: "dado um algoritmo, melhorá-lo, ou escolher o melhor dentre dados algoritmos, segundo os critérios de correção, quantidade de trabalho, quantidade de espaço disponível, simplicidade e otimalidade" [TOS2001]. Complexidade de um algoritmo é o esforço, a quantidade de trabalho despendido em sua execução. As principais medidas de complexidade de algoritmos seqüenciais são tempo e espaço, relacionadas à velocidade e quantidade de memória, respectivamente. A complexidade é determinada com base em operações básicas e no tamanho da entrada. Ambos devem ser apropriados ao algoritmo e ao problema.

Para o cálculo de complexidade, pode-se medir o número de segundos num computador específico ou medir o número de passos de execução num modelo matemático. A complexidade experimental (medir o tempo de execução requerido para a solução de um problema particular) costuma depender de detalhes de implementação, variando de máquina a máquina. No modelo matemático, os passos de execução de um algoritmo são estimados pelo número de operações básicas requeridas pelo algoritmo em função do tamanho da entrada. Esses são dois extremos, onde os resultados podem diferir por mais do que um simples fator de escala. As funções de complexidade são descritas por classes de comportamentos assintóticos polinomiais (*constante, log n, n, n log n, n^2, n^3, ...*) e não polinomiais (*exponencial*: 2^n, 3^n, ..., x^n ou *fatorial*).

3.3.1 Complexidade de algoritmos

A quantidade de trabalho requerido por um algoritmo, na sua execução, não pode ser descrita simplesmente por um número, porque o número de operações básicas efetuadas, em geral, não é o mesmo para qualquer entrada (depende do tamanho da

entrada). Por exemplo, numa inversão de matrizes, o número de operações (de adição e multiplicação) varia com a dimensão da matriz, ou na classificação de uma lista depende do número de elementos da lista e do tipo de entrada, pois mesmo para entradas do mesmo tamanho, o número de operações efetuadas pelo algoritmo pode depender da entrada particular. Para classificar uma lista quase ordenada, pode não ser necessário o mesmo esforço que para classificar uma lista de mesmo tamanho, mas com os elementos em grande desordem. A expressão "quantidade de recurso requerido" também é chamada *complexidade do algoritmo*. A complexidade depende da entrada particular, sendo os principais critérios o pior caso e o caso médio.

- *Complexidade no pior caso*. A complexidade é tomada como a máxima para qualquer entrada de um dado "tamanho".
- *Complexidade Média.* A complexidade leva em conta a probabilidade de ocorrência de cada entrada de um mesmo "tamanho".

Quando se sabe apenas a complexidade média de um algoritmo, não se sabe nada sobre o seu caso específico. Mas se a complexidade do pior caso é conhecida, sabe-se que no máximo o seu caso tem a complexidade do pior caso.

A complexidade assintótica dá o desempenho para entradas de tamanho grande. As principais notações de comportamento assintótico dão limites superior - notação O (cota assintótica superior), inferior - notação Ω (cota assintótica inferior) e exato - notação Θ (limite assintótico exato), a menos de constantes multiplicativas. O comportamento assintótico de um algoritmo é o mais procurado, já que, para um volume grande de dados, a complexidade torna-se mais importante. O algoritmo assintoticamente mais eficiente é melhor para todas as entradas, exceto para entradas relativamente pequenas. Assim, o cálculo da complexidade se concentra em determinar a ordem de magnitude do número de operações básicas na execução do algoritmo.

3.3.2 Complexidade de problemas

O mais importante questionamento sobre um problema é a sua computabilidade, isto é, se ele pode ser resolvido mecanicamente (por um algoritmo). Para identificar e definir questionamentos como esse, surgiram vários formalismos e máquinas abstratas. Entre esses, o mais interessante é a Máquina de Turing. A Máquina de Turing tornou-se a mais utilizada definição de algoritmo e, assim, deu precisão à definição de *computável* - um problema é computável se é resolvível em uma Máquina de Turing.

Quando se acrescenta à computabilidade uma propriedade de eficiência, isto é, além de um algoritmo que resolva o problema, quer-se um algoritmo eficiente (uma definição aceitável para *eficiente* é "cuja complexidade seja polinomial"), tem-se também duas classes de problemas: *P*, a classe de problemas para os quais existe algoritmo (Máquina de Turing) de ordem polinomial, que resolve o problema; *NP*, a classe de problemas para os quais existe algoritmo (Máquina de Turing) de ordem polinomial, que, dada uma entrada para o problema, verifica (faz a certificação) se tem resposta SIM ou NÃO (ou, equivalentemente, têm algoritmo não determinístico de ordem polinomial).

A questão *P versus NP* surgiu em 1971, com o artigo *The complexity of theorem-proving procedures* ([COO71]), e tem sido intensamente estudada. A conjectura que existe é "$P \neq NP$". A partir dessa conjectura, foi criada toda uma teoria, na qual a figura principal é a classe *NP-Completa*.

Neste item, o ponto de referência deixará de ser o algoritmo para ser o problema. O objetivo é estudar as limitações dos problemas com respeito à complexidade dos algoritmos que os resolvem.

O *limite superior de complexidade de um problema* refere-se ao melhor algoritmo conhecido que o resolve. Já o limite inferior de um problema refere-se à melhor complexidade possível. Os problemas têm suas dificuldades, que não permitem o desenvolvimento de algoritmos melhores do que um certo limite de complexidade. Um limite inferior é um resultado teórico que determina não ser possível desenvolver um algoritmo para o problema com complexidade melhor que uma certa função. Essa função então é um *limite inferior de complexidade do problema*.

A técnica mais simples de cálculo de limite inferior é simplesmente contar as entradas e as saídas produzidas, pois o algoritmo precisa, no mínimo, ler a entrada e dar as saídas. Um limite inferior pode ser obtido sem, entretanto, conhecer-se um algoritmo que realmente atinja essa complexidade. Nesse caso, o limite inferior conhecido não é igual ao limite superior conhecido.

Enquanto houver diferença entre os limites de complexidade inferior e superior para um problema, a questão pode progredir nos dois sentidos: um algoritmo melhor que o limite superior pode ser desenvolvido, ou um cálculo mais apertado da complexidade mínima pode ser alcançado, diminuindo a diferença entre esses dois limites. Quando essa diferença desaparece, a complexidade mínima do problema é conhecida. Então, a complexidade mínima do problema é a melhor complexidade que um algoritmo que resolve esse problema pode atingir. O problema do cálculo da complexidade, nesse caso, é dito *fechado*.

Alguns problemas são *bem comportados* e permitem chegar a limites de complexidade, como o problema de classificação por comparações, para o qual é conhecido o limite de complexidade ($n \log n$). Isso significa que não existe algoritmo de classificação, baseado em comparações, de complexidade melhor do que $O(n \log n)$. Já o problema de operações entre matrizes não é fechado, ou seja, seus limites de complexidade inferior e superior não são iguais. Um limite superior para o problema conhecido é $O(n^{2.374})$ atribuído a *Pan* ([PAN78]). Sabe-se, também, que $O(n^2)$ é um limite inferior, logo é impossível resolver o problema com complexidade menor do que essa. Esforços têm sido feitos para reduzir esse intervalo.

Historicamente, a expressão "*algoritmo não eficiente*" é associada à condição de algoritmo de complexidade não polinomial, ou simplesmente algoritmo não polinomial, como é mais utilizado. Essa associação de problema intratável, algoritmo não eficiente e algoritmo não polinomial justifica-se, porque um algoritmo com complexidade, por exemplo, $O(2^n)$ tem um tempo de execução que cresce tão rapidamente com n que, para um n razoavelmente grande, torna-se proibitivo, e o problema que tem esse algoritmo como melhor opção de solução, torna-se intratável para entradas razoavelmente grandes. Os algoritmos podem ser, então, particionados em duas classes: os *razoáveis* (polinomiais) e os não *razoáveis* (exponenciais).

Um problema é dito *tratável* se o limite superior de complexidade é polinomial, isto é, conhece-se um algoritmo razoável que o resolve. Um problema é dito *intratável* se o limite superior de complexidade é exponencial. Há os problemas cuja resposta é tão longa, que se precisa de um tempo exponencial para descrevê-la. Mas, nesse caso, costuma-se dizer que o problema não está bem posto, isto é, não está definido realisticamente.

Os chamados problemas *NP-completos* são problemas que têm a questão da complexidade ou tratabilidade não resolvida, pois não se sabe se existe ou não algoritmo polinomial que o resolva. Essa é uma das questões do século e vale um milhão de dólares.

O fato de um problema *NP-completo* qualquer ter um algoritmo polinomial que o resolva implicar que todos os outros (milhares de problemas *NP-completos*) também

possuam algoritmo polinomial que os resolva e nenhum algoritmo polinomial para eles ter sido desenvolvido até o momento é uma evidência de que esses algoritmos não existem. Portanto, os problemas *NP-completos* seriam intratáveis. Essas circunstâncias tornam o conhecimento dos conceitos de *NP-Completude* e os procedimentos de reconhecimento da pertinência de um problema nessa classe um requisito fundamental na formação de um profissional da área da Computação.

Os problemas estão classificados segundo sua complexidade, localizados em várias classes possíveis. As principais classes, considerando algoritmos seqüenciais são *P, NP, NP-completa*. Considerando algoritmos paralelos, existem as classes NC^k e *NC*.

3.3.2.1 Classes *P, NP, NP-completa*

Um *algoritmo é polinomial* (de complexidade de tempo polinomial) se é O(*p(n)*), onde p é um polinômio, e *n* representa o tamanho da entrada. Se um algoritmo não é polinomial, diz-se que é *exponencial* (mesmo que sua complexidade não esteja na forma usualmente chamada como função exponencial).

Um *problema de decisão* é dito *P* se pode ser resolvido deterministicamente em tempo polinomial, e *NP* se pode ser resolvido não deterministicamente em tempo polinomial. Outra forma de se definir a classe *NP* é como a classe de problemas que podem ser certificados em tempo polinomial, isto é, problemas que possuem algoritmo determinístico tal que, dada uma entrada *x* e uma candidata a solução *y*, é verificável em tempo polinomial se *y* é solução para a entrada *x*.

Assim, intuitivamente, *P* é a classe dos problemas que podem ser resolvidos eficientemente, e *NP* é a classe dos problemas que podem ser certificados eficientemente. Os problemas de *NP* possuem algoritmos determinísticos de ordem exponencial.

Se $\Pi \in NP$, então existe um polinômio p tal que Π pode ser resolvido por algoritmo determinístico de complexidade $O(2^{p(n)})$. (Em [TOS2001, GAR69] encontra-se a demonstração). Esse resultado permite concluir que qualquer problema *NP-completo* tem solução paralela de ordem polinomial com $2^{p(n)}$ processadores.

É comum, no processo de solução de um problema Π, reduzi-lo a outro problema Π', o que significa: dada uma instância de Π, construir uma instância de Π', tal que a solução da instância de Π determine a solução da instância de Π' correspondente. Se a redução é computável em tempo polinomial é chamada de *redução polinomial* ($\propto$). Um problema Π é dito *NP-completo* se $\Pi \in NP$ e, para qualquer outro $\Pi' \in$ NP, $\Pi' \propto \Pi$. Assim, identificam-se como *NP-completos* os problemas mais difíceis entre os problemas de *NP*.

3.3.2.2 Classes NC^k e *NC*

Para caracterizar problemas mais promissores de possuírem boas soluções paralelas é necessário usar um modelo de máquina paralela. Existem muitos modelos de máquinas paralelas, desde os mais simples aos mais sofisticados. *Bovet* e *Crescenzi* [BOV93], a fim de fazer uma avaliação dos vários modelos, definiram uma condição que chamaram *Tese da Computação Paralela*, que dado um modelo ***M***, ele satisfaz a *Tese* se existem polinômios *p* e *q* tais que: P-TIME[*f(n)*] $\subseteq$ DSPACE[*p(f(n))*] e DSPACE[*g(n)*] $\subseteq$ P-TIME[*q(g(n))*], onde P-TIME[f(n)] é a classe das linguagens decididas no modelo ***M*** num tempo O(*f(n)*) e DSPACE(*g(n)*) como a classe das linguagens decididas deterministicamente no modelo ***M,*** usando um espaço O(*g(n)*).

Muitos modelos não satisfazem a *Tese da Computação Paralela,* entretanto, segundo [BOV93], a satisfação da *tese* é uma garantia de que o modelo de máquina paralela considerado é *razoável*. Dois modelos que satisfazem essa tese são: Circuitos

de funções booleanas e *PRAM*. Nos Circuitos de funções booleanas, a complexidade (c) é medida pelo número de portas SIZE(c) e pelo comprimento do maior caminho de uma porta de entrada a uma porta de saída, DEPTH(c). Como cada porta tem no máximo duas saídas, $SIZE(c) \leq 2^{DEPTH(c)}$. Para uma função booleana *f*, $SIZE_f$ é definida por:

$SIZE_f = \min\{q / c \text{ que computa f tal que } SIZE(c) = q\}$.

Isto é o mínimo entre o número de portas de todos os circuitos que computam *f*. Uma família de circuitos $C = \{ c_n / n > 1\}$ é dita uniformemente construída se para todo *n* o circuito c_n pode facilmente ser determinado. A noção de "*fácil*" afeta a complexidade da classe, pois se a condição de *uniformidade é muito fraca*, a teoria torna-se trivial, pois o poder de computação fica na construção do circuito ao invés de no circuito. O contrário, condição de *uniformidade forte demais* estabelece uma relação impossível entre a família de circuitos e os computadores paralelos reais. A noção de "*fácil*" que parece adequada é "circuitos projetados em um espaço de trabalho constante".

Dadas essas definições, pode-se agora definir, a partir da complexidade das famílias uniformes de circuitos, a classe de problemas *NC*.

$$NC = \bigcup_{k \geq 1} NC^k$$

onde NC^k é a classe das linguagens L para as quais existe uma família uniforme de circuitos $C = \{ c_n / n > 1\}$ e uma constante *h* tal que:

- C decide L;
- $SIZE_c(n)$ é $O(n^h)$ ($SIZE_c(n) = SIZE(c_n)$)
- $DEPTH_c(n)$ é $O(log^k(n))$ ($DEPTH_c(n) = DEPTH(c_n)$)

As classes *NC's* foram definidas inicialmente por *Pippner* em [PIP79], como a classe das linguagens decididas por Máquina de Turing determinística em tempo polinomial, tal que o número de vezes que o cabeçote troca de direção é logarítmico. Portanto, $NC \subseteq P$ e conjectura-se que $P \neq NC$.

Da mesma forma define-se FNC^k e *FNC* como as classes de funções computáveis satisfazendo: $SIZE_f(n)$ é $O(n^h)$ e $DEPTH_f(n)$ é $O(log^k(n))$, isto é, complexidade logaritmicamente polinomial em profundidade e polinomial no tamanho para famílias de circuito uniforme.

Exemplos de problemas em *NC* são: classificação está em NC^1, tem $O(log\ n)$ com n^2 processadores; produto de matrizes quadradas está em NC^1, tem $O(log\ n)$ com n^3 processadores; menor caminho num grafo está em NC^2, tem $O(log^2 n)$ com n^2 processadores; potenciação de matrizes quadradas está em NC^2, tem $O(log^2 n)$ com $(n^3/2)$ processadores e a inversão de matrizes está entre NC^1 e NC^2.

3.3.3 Medidas de complexidade de algoritmos paralelos

As principais medidas de complexidade sobre os quais o desempenho de algoritmos seqüenciais é medido são o tempo e o espaço. Seguindo uma analogia a esses recursos para avaliação do desempenho de algoritmos paralelos, pode-se verificar que o tempo permanece como o principal recurso no processamento paralelo, só que agora ele depende não somente da complexidade das operações computacionais, mas também da complexidade do gerenciamento das operações criadas pela comunicação, sincronização e restrições de acesso aos dados. O desempenho medido quanto ao espaço depende da quantidade de memória de que o algoritmo necessita.

Em computadores paralelos, esse fator memória é de menor importância, pois o tamanho do *hardware*, que é o número de elementos de uma máquina paralela que são

ativados durante a computação, representa o recurso mais importante nas considerações sobre o desempenho do programa.

Quando um novo algoritmo está sendo projetado, é usual avaliá-lo segundo alguns critérios como: tempo de execução, número de processadores usados e custo. Juntamente com essas medidas padrão, um número de outras medidas de tecnologia são algumas vezes utilizadas, quando se sabe que um algoritmo será executado em um computador com uma tecnologia particular.

3.3.3.1 Tempo de execução

O tempo de execução de um algoritmo seqüencial é estimado pelo número de operações básicas requeridas pelo algoritmo como uma função do tamanho da entrada. A operação básica e seu custo são expressos por uma função do tamanho da palavra dos dados envolvidos: esses dependerão do problema específico a ser resolvido e do modelo de computação utilizado. Em geral, operações básicas são tomadas como sendo: unidade de tempo de um acesso à memória para leitura ou escrita; operações aritméticas e lógicas elementares (adição, subtração, comparação, multiplicação de dois números e o cálculo das operações lógicas e e ou entre duas palavras).

Uma vez que velocidade de computação aparece como sendo a principal razão, a principal questão na construção de computadores paralelos, a medida mais importante na avaliação de algoritmos paralelos é, portanto, o tempo de execução. Ele é definido como o tempo absorvido por um algoritmo para resolver um problema em um computador paralelo, isto é, o tempo desde o momento que o algoritmo inicia até o momento que ele termina. Se os vários processadores não iniciam e nem terminam juntos, então o tempo de execução é igual ao intervalo de tempo entre o momento do primeiro processo iniciar a computação até que o último processo termine a computação. *Kung* (em [KUN76]) definiu o tempo absorvido por um programa paralelo, como o intervalo de tempo do processo em um programa que termina por último, onde o intervalo de tempo do processo é calculado como a soma de três quantidades:

- *tempo de processamento básico*, o qual é a soma dos tempos absorvidos pelos seus estágios;
- *tempo bloqueado*, o qual é o tempo total que o programa é bloqueado até o final do estágio devido a espera de dados na sincronização do algoritmo ou de espera para entrada numa seção crítica de um algoritmo assíncrono;
- *tempo de execução do gerenciamento da sincronização*, a qual é feita através de operações primitivas de sincronização e de implementação e acesso a regiões críticas.

Em processamento paralelo, o objetivo do menor tempo de execução não é, necessariamente, sinônimo de realizar o menor número de operações aritméticas como em processamento seqüencial, pois pode haver um número maior de operações, as quais podem ser executadas simultaneamente. Em alguns casos, uma medida útil de desempenho para processamento paralelo pode ser o parâmetro, o qual é inversamente proporcional ao tempo de processamento, isto é, o tempo durante o qual unidades do computador (aritméticas e lógicas) são utilizadas por um programa, estão engajadas durante a execução do programa.

Antes de implementar um algoritmo (seqüencial ou paralelo) em um computador, é bom fazer uma análise teórica do tempo necessário para resolver o problema computacional em questão. Isso é geralmente feito pela contagem do número de operações básicas ou passos executados pelo algoritmo no caso pessimista. Isso leva a uma expressão que descreve o número de passos como uma função do tamanho da entrada.

O tempo de execução de um algoritmo paralelo é, geralmente, obtido pela contagem de dois tipos de passos: *passos computacionais* e *passos de envio*. Um passo computacional consiste de uma operação aritmética ou lógica realizada sobre um dado por um processador. Por outro lado, em um passo de envio, um dado vai de um processador para outro através da memória compartilhada ou através da rede de comunicação. O tempo de execução é também uma função do número de processadores. Geralmente, passos computacionais e passos de envio não requerem, necessariamente, o mesmo número de unidades de tempo. Um passo de envio depende da distância entre os processadores.

Dado um problema computacional para o qual um novo algoritmo seqüencial tenha sido projetado, é comum entre projetistas de algoritmos questionar qual o algoritmo é o mais rápido possível para problemas e, caso não o seja, como compará-lo com os demais algoritmos já existentes para o problema.

Sabe-se, por exemplo, que ao computar o produto de duas matrizes *nxn,* a matriz resultante tem n^2 elementos; para isso, muitos passos são necessários para se produzir o resultado, não se sabe a complexidade mínima. Já o problema de classificação tem complexidade mínima conhecida O(*n log n*).

A mesma idéia geral de cotas se aplica a algoritmos paralelos, mas se tem que adicionar dois fatores: o modelo de computação paralela e o número de processadores envolvidos.

Na avaliação de algoritmos paralelos para um dado problema, é bastante natural ter-se uma comparação em termos do melhor algoritmo seqüencial disponível. Então, uma boa indicação da qualidade de um algoritmo paralelo é o *speedup*. Ele foi definido como o quociente do tempo de execução do mais rápido algoritmo seqüencial para o problema, no pior caso, pelo tempo do algoritmo paralelo, também no pior caso.

Suponha que se conheça um algoritmo seqüencial para dado um problema que seja o mais rápido possível. Idealisticamente, espera-se ter o melhor *speedup*, quando se resolve tal problema utilizando processadores operando em paralelo. Na prática, tal *speedup* não é obtido, pois não é sempre possível decompor o problema em *p* processos, onde cada um requeira *1/p* do tempo necessário por um processador resolver o problema original e, na maioria dos casos, a estrutura do computador paralelo utilizada para resolver o problema, geralmente impõe restrições que faz com que o tempo de execução desejado seja inalcançável.

Outra questão é diferenciar a medida teórica e a medida prática de tempo de processamento. A medida teórica conta o número de passos para a resolução do problema. A medida prática leva em conta o número de processadores, a divisão do problema em tarefas que podem ser executadas em paralelo, o tempo de espera dos processadores e o tempo de gerenciamento do sincronismo dos processos.

3.3.3.2 Número de processadores

O número de processadores disponíveis é limitado, portanto é preciso usá-lo eficientemente. Na prática, tem-se um número fixo de processadores disponíveis e o objetivo é desenvolver um algoritmo paralelo que minimize o tempo de execução. Ainda é desejável um algoritmo que se adapte ao número de processadores disponíveis no momento e que sua eficiência não sofra uma diminuição significativa com uma perda não significativa no número de processadores.

O tamanho do *hardware*, o número de elementos da máquina paralela os quais estão ativos durante o processamento, representa o principal recurso a ser tomado nas considerações de desempenho do programa. Exemplos são: em processadores

matriciais, o número de processadores envolvidos na computação; em máquina vetorial, a soma dos tamanhos dos vetores.

3.3.3.3 Complexidade de comunicação

Uma simples troca de dados entre processadores é, em geral, mais demorada que a execução de uma operação. A distância entre os processadores que se comunicam é um fator considerável no tempo de comunicação e, portanto, é importante minimizar a comunicação entre processadores e colocá-los no espaço de forma eficiente.

3.3.3.4 Custo

O custo de um algoritmo paralelo é definido como o produto de duas quantidades, ou seja, o produto do tempo de execução paralelo pelo número de processadores utilizados. Isso significa que o custo é igual ao número de passos executados coletivamente por todos processadores na solução de um problema no pior caso. Essa definição assume que todos processadores executam o mesmo número de passos.

Assume-se que o limite inferior é conhecido como o número de operações seqüenciais requeridas, no pior caso, para resolver o problema. Se o custo de um algoritmo paralelo para o problema atingir o limite inferior com um fator de uma constante multiplicativa, então o algoritmo é dito ser de custo ótimo. Isso é porque qualquer algoritmo paralelo pode ser simulado por um algoritmo seqüencial. Se o número total de passos executados durante uma simulação seqüencial for igual ao limite inferior, então isso significa que, em termos de custo, esse algoritmo paralelo não pode proporcionar, com sua execução, o menor número de passos possíveis. Isso pode ser possível se para reduzir o tempo de execução do custo ótimo do algoritmo paralelo forem utilizados mais processadores.

Um algoritmo paralelo não tem custo ótimo se existe um algoritmo seqüencial que possui tempo de execução menor que o custo do algoritmo paralelo.

Uma implementação paralela de um algoritmo para resolver um problema de tamanho n é dito consistente se o número de operações elementares requeridas por essa implementação for da mesma ordem de magnitude da função que expressa a quantidade necessária para sua implementação em um computador seqüencial.

3.4 Programação paralela

O paralelismo pode ser *implícito* ou *explícito*. O *paralelismo implícito* é tido como aquele que é inerente ao próprio programa. Esse tipo de paralelismo, geralmente, é identificado pelo sistema operacional. Portanto, cabe ao sistema operacional fazer a análise de dependências, visando identificar partes ou pedaços do programa que podem ser processados ao mesmo tempo, ou seja, podem ser paralelizados. A independência é dada pela necessidade de acesso exclusivo aos dados.

O *paralelismo explícito* pode aparecer em três formas:

- *Paralelismo através de dados* (*data paralellism*) - muito comum em arquiteturas matriciais, onde uma única instrução é executada por vários elementos processadores sobre diversos fluxos de dados;
- *Paralelismo por troca de mensagens* (*message passing*) - são encontrados em arquiteturas de multiprocessadores, onde vários processadores estão envolvidos na execução do programa, sendo necessário que se comuniquem. Nesses casos a maior parte do processamento é local, mas há uma pequena parte do processamento que consiste em troca de informação;

- *Paralelismo através de variáveis comuns* - também encontrado em arquiteturas de multiprocessadores, onde a comunicação se faz através de variáveis comuns (memória global), as quais substituem mecanismos de comunicação, onde os processos trocam de informação através de acessos à memória, o que proporciona uma comunicação rápida.

3.4.1 Programação paralela com memória compartilhada

O modelo de memória compartilhada, baseado em variáveis comuns, é uma extensão natural do modelo seqüencial, diferenciando-se por ter vários processadores que têm acesso a uma única unidade de memória, compartilhada por todos. Formalmente, o modelo consiste de um número de processadores, cada um deles com sua própria memória local, que pode executar seu programa. Toda comunicação é feita pela troca de dados através da memória global compartilhada. Cada processador é unicamente identificado por um índice, chamado de *número* ou *identificador do processador*, anotado por ID, o qual é disponibilizado localmente. A fig. 3.1 fornece uma visão geral desse modelo com *p* processadores. Os processadores são identificados pelos índices *1, 2,..., p.*

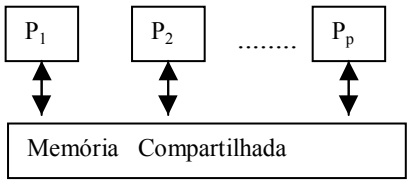


Figura 3.1: Modelo de memória compartilhada.

Existem dois modos básicos de operação no modelo de memória compartilhada, que são síncrono e assíncrono. No primeiro, chamado de *síncrono*, todos os processadores operam sincronizadamente sob o controle de um relógio comum. A nomenclatura padrão para esse modelo de memória sincronizada é *PRAM* (*Parallel Random-Access Machine*).

No segundo, chamado de *assíncrono*, cada processador opera sob um relógio separado. No modo de operação assíncrono, fica a cargo do programador a definição de pontos de sincronização, quando necessário. Se um dado necessita ser acessado por um processador, o programador se encarrega de garantir que os valores corretos serão obtidos. Assim, o valor da variável compartilhada é determinado dinamicamente durante a execução dos programas dos diferentes processadores.

Uma vez que cada processador pode executar seu próprio programa, esse modelo de memória compartilhada é do tipo MIMD. Cada processador pode executar uma instrução, ou operar com dados diferentes dos demais, ou operar junto com outro processador durante qualquer unidade de tempo.

Para se ter o algoritmo, o tamanho dos dados transferidos entre a memória compartilhada e as memórias locais dos diferentes processadores representa a quantidade de comunicação requerida pelo algoritmo. Deve-se ter comandos que permitam mover um bloco de dados da memória global para a memória local e que escrevam um dado local na variável compartilhada da memória global. Deve-se ter, ainda, comandos que permitam a criação de regiões de acesso exclusivo, as regiões críticas.

3.4.2 Programação paralela por troca de mensagens

Programação paralela por troca de mensagens ocorre em máquinas do tipo multicomputador, onde cada nodo consiste de um processador, de memória *RAM* e de dispositivos de entrada e saída, ou seja, cada nodo é um computador independente.

Nas máquinas com memória distribuída, cada máquina possui sua memória local, não sendo acessível por outras máquinas. Como cada computador só pode acessar sua memória, a troca de informações é efetuada através da rede de interconexão, a qual tem um papel de grande importância quando se pretende ter um bom desempenho.

Uma rede pode ser vista como um grafo descrito pelo conjunto dos nodos e arcos, ou seja, o grafo G = (N, E), onde cada nodo $i \in N$ representa um processador e cada arco $(i, j) \in E$ representa um *link* de comunicação de mão-dupla entre os processadores *i* e *j,* como representado na fig. 3.2. É assumido que cada processador tenha sua própria memória local e que não exista memória compartilhada. As operações podem ser síncronas ou assíncronas.

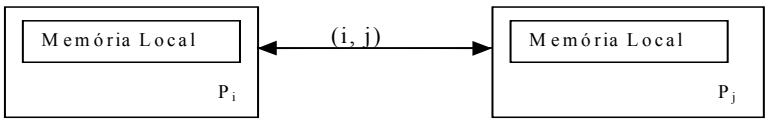


Figura 3.2: Rede por Grafo.

Na descrição de algoritmos paralelos por troca de mensagem, são necessárias instruções que descrevam a comunicação entre processadores. Essas instruções implementam o envio e o recebimento de mensagens.

- *Envio*. Quando um processador P executa a instrução de envio, ele está enviando uma cópia de dados para o processador P_i. O processador continua a execução de seu programa local após o envio;
- *Recebimento.* Quando um processador P executa uma instrução de recebimento, ele suspende a execução de seu programa até que a informação seja recebida do processador P_j. Uma vez recebida, ela é armazenada e a execução do programa prossegue.

Nesse método os processadores não necessitam ser adjacentes. Podem existir *roteadores*, que se encarregam de levar cada mensagem de sua fonte ao seu destino.

O modelo de redes incorpora a topologia de interconexão entre os processadores em si mesmo. Existem vários parâmetros usados para avaliar a topologia de uma rede G. Entre eles estão: diâmetro (distância máxima entre qualquer par de nodos), grau máximo de um nodo em G e a conectividade dos nodos e arcos na rede. Exemplos de topologias de intercomunicação, são:

- *array linear* de processadores consiste de *p* processadores P_1, P_2, ..., P_p, conectados em um *array* linear, isso é, o processador Pi está conectado ao processador P_{i-1} e ao processador P_{i+1}, se eles existirem;
- *array circular ou anel* é uma *array* linear de processadores onde o primeiro e o último estão conectados, ou seja, P_1 e P_p estão diretamente conectados;
- *malha bidimensional* é uma versão bidimensional de processadores interconectados por *arrays* lineares. Ela consiste de $p=m^2$ processadores, organizados em uma grade *mxm* de processadores, de tal forma que o processador $P_{i,j}$ é conectado aos processadores $P_{i\pm1,j}$ e $P_{i,j\pm1}$, quando eles existem;
- *malha completamente interconectada (estrela)*, onde cada processador possui ligações a todos os outros processadores. Nessa topologia os vértices

representam processadores e as arestas representam a comunicação entre os pares de processadores, são necessários $p \times (p-1)$ ligações;

- *hipercubo* tem uma estrutura recursiva. Pode-se estender um cubo *d*-dimensional para um cubo *(d+1)*-dimensional, através da conexão de processadores correspondentes de dois cubos *d*-dimensionais.

As topologias de estrela e anel descrevem as topologias das redes *Myrinet* e *SCI* do cluster do Instituto de Informática da UFRGS. São redes de tecnologias independentes: a primeira é uma rede interconectada com 6 nodos e a segunda é uma rede em anel com 4 nodos.

A comunicação dessas máquinas através da rede é baseada no uso de bibliotecas de troca de mensagens, como por exemplo: MPI e DECK. O MPI (*Message Passing Interface)* é um modelo de comunicação através de troca de mensagens, largamente utilizado para exploração do paralelismo em multicomputadores. Esse modelo é portável para qualquer arquitetura, tendo aproximadamente 125 primitivas para programação ([CNA2001]), entre elas destacam-se:

- *primitivas de administração* - responsáveis por criar grupos, informar aos processos suas identificações e controlar o número de processos existentes no grupo em que estão inseridos;
- *primitivas de comunicação ponto a ponto* - responsáveis por envio e recebimento de mensagens entre dois processos;
- *primitivas de comunicação em grupos*- responsáveis pela comunicação entre vários processos em um grupo, exemplo é o *broadcast*, envio de uma mensagem a todos os processos do grupo.

As primitivas de comunicação no MPI podem ser ou não *bloqueantes* (aguardam que a transferência seja completada), com ou sem *bufferização* (uso de buffer para a transferência da mensagem), o que torna o desenvolvimento de aplicação flexível. Existem implementações do MPI que são de domínio público (exemplo: MPICH).

O DECK (*Distributed Execution and Communication Kernel)* é uma biblioteca desenvolvida pelo GPPD da UFRGS e tem como vantagens sobre o MPI, a característica de poder ter várias *threads* enviando e recebendo mensagens. As primitivas do comunicação do DECK são *bloqueantes* e *bufferizadas*. O tamanho do *buffer* é aproximadamente de 1 Megabyte, o que pode implicar no uso de uma política de troca de mensagens para evitar a ocorrência de problemas. Existem primitivas de administração, primitivas de comunicação ponto a ponto e em grupos.

3.5 Exemplo do problema de classificação

A classificação é definida como sendo a tarefa de ordenar uma coleção de elementos desordenados em uma ordem monotônica crescente ou decrescente. Especificamente, o problema de classificação pode consistir em rearranjar uma seqüência de *n* elementos ($S = < a_1, a_2, ..., a_n >$) em uma ordem arbitrária (crescente ou decrescente), ordenar um arquivo ou uma base de dados. Na descrição de muitos algoritmos de classificação existentes na literatura se utiliza o conceito de chaves. Elas são empregadas para acessar e ordenar bancos de dados. Nesse contexto ordenar um banco de dados é produzir um arquivo de chaves ordenados segundo algum critério, isso é mais econômico do que gerar novo banco de dados a cada ordenação. A multiplicidade de soluções é, em parte, motivada pela ocorrência desse problema no processamento de informação, em problemas comerciais e científicos e em processamento de listas. Exemplos disso são listas telefônicas, dicionários e acessos a banco de dados.

Considere o problema de classificar uma lista de *n* números inteiros. Inicialmente, considere a solução por comparação e troca, onde a entrada do algoritmo consiste de *n* elementos (números inteiros). A solução consiste nesses elementos em ordem crescente. Para tanto, são comparados dois elementos, se estiverem fora de ordem eles são trocados entre si. O *Algoritmo 3.5.1* apresenta essa solução. É fácil perceber que o número de comparações é O(*n(n-1)/2)* e que no pior caso, quando a lista estiver ordenada de forma decrescente, será necessário exatamente esse número de trocas. O tamanho da instância é o número de elementos *n*. Tem-se, então, que a complexidade no pior caso e caso médio é O(n^2).

ALGORITMO 3.5.1 - Comparação e Troca.

```
Entrada:  O tamanho da lista e a lista de n elementos
Saída: A lista classificada em ordem crescente.
inicio
      leia (n);
      leia (lista(1:n))
      para j:= n-1 até 1 faça
            para i := 1 até j faça
               se lista(i) > lista (i+1)
                  então
                    inicio
                     Aux:= lista(i+1); lista(i+1):=lista(i);  lista(i):= Aux;
                    fim
fim
```

Exemplo 3.5.1: Seja *n* = *8*, e a lista a ser ordenada L = < 2, 8, 3, 7, 9, 1, 6, 5 >

j	i	$lista_i > lista_{i+1}$	Lista_ordenada
j=7	i=1	f	< 2, 8, 3, 7, 9, 1, 6, 5 >
j=7	i=2	v	< 2, **3**, **8**, 7, 9, 1, 6, 5 >
j=7	i=3	v	< 2, 3, **7, 8**, 9, 1, 6, 5 >
j=7	i=4	f	< 2, 3, 7, 8, 9, 1, 6, 5 >
j=7	i=5	v	< 2, 8, 3, 7, **1, 9**, 6, 5 >
j=7	i=6	v	< 2, 8, 3, 7, 1, **6, 9**, 5 >
j=7	i=7	v	< 2, 8, 3, 7, 1, 6, **5, 9** >
j=6	i=1	f	< 2, 8, 3, 7, 1, 6, 5, 9 >
j=6	i=2	v	< 2, **3, 8**, 7, 1, 6, 5, 9 >
j=6	i=3	v	< 2, 3, **7, 8,** 1, 6, 5, 9 >
j=6	i=4	v	< 2, 3, 7, **1, 8**, 6, 5, 9 >
j=6	i=5	v	< 2, 3, 7, 1, **6, 8**, 5, 9 >
j=6	i=6	v	< 2, 3, 7, 1, 6, **5, 8**, 9 >
j=5	i=1,2	f	< 2, 3, 7, 1, 6, 5, 8, 9 >
j=5	i=3	v	< 2, 3, **1, 7**, 6, 5, 8, 9 >
j=5	i=4	v	< 2, 3, 1, **6, 7**, 5, 8, 9 >
j=5	i=5	v	< 2, 3, 1, 6, **5, 7**, 8, 9 >
j=4	i=1	f	< 2, 3, 1, 6, 5, 7, 8, 9 >
j=4	i=2	v	< 2, **1, 3**, 6, 5, 7, 8, 9 >
j=4	i=3	f	< 2, 1, 3, 6, 5, 7, 8, 9 >
j=4	i=4	v	< 2, 1, 3, **5, 6**, 7, 8, 9 >
j=3	i=1	v	< **1**, **2**, 3, 5, 6, 7, 8, 9 >
j=3	i=2,3	f	< 1, 2, 3, 5, 6, 7, 8, 9 >
j=2	i=1,2	f	< 1, 2, 3, 5, 6, 7, 8, 9 >
j=1	i=1	f	< 1, 2, 3, 5, 6, 7, 8, 9 >

Figura 3.3 Computação do *Algoritmo 3.5.1*

O *Algoritmo 3.5.2* apresenta o *Rank Sort.* A idéia geral desse algoritmo é de comparar cada elemento da lista a ser classificada com todos os outros, fazendo a contagem de todos os elementos que são menores que o elemento que está sendo calculado. A complexidade de tempo desse algoritmo seqüencial também é O(n^2).

ALGORITMO 3.5.2 – Rank Sort Seqüencial

```
Entrada: O tamanho n da lista e seus elementos
Saída: A lista classificada em ordem crescente (lista_ordenada)
inicio
   para i = 1 até n faca
     inicio
        contador := 0;
        para j:= 1 até n faça
              se (lista[i] ≥ lista[j]) então contador:= contador + 1;
         lista_ordenada[contador] = lista[i];
      fim
fim
```

O algoritmo envolve três tarefas principais:

- *Contagem*: os elementos são particionados em subconjuntos e para cada elemento o número de elementos menores que ele é determinado (em uma versão simples do algoritmo é assumido que todos os elementos são diferentes);
- *Rank*: a posição final do elemento na seqüência ordenada é determinado pela soma das comparações verdadeiras obtida na fase de contagem;
- *Reorganização dos Dados*: cada elemento é colocado na sua posição final determinada pelo seu *rank*.

Exemplo 3.5.2: Sejam $n = 4$ e a lista L = < 4, 2, 3, 1 >. A computação do *Algoritmo 3.5.2*, é apresentada pela fig. 3.4. para uma entrada com quatro elementos. A seguir será apresentada a computação desse algoritmo para a lista com oito elementos, apresentada no *Exemplo 3.5.1*.

```
i = 1
        j = 1          lista[1]=4 ≥ lista[1]=4 → contador = 1
        j = 2          lista[1]=4 ≥ lista[2]=2 → contador = 2
        j = 3          lista[1]=4 ≥ lista[3]=3 → contador = 3
        j = 4          lista[1]=4 ≥ lista[4]=1 → contador = 4
lista_ordenada[contador] = lista[1];
i = 2
        j = 1          lista[2]=2 ≥ lista[1]=4 → contador = 0
        j = 2          lista[2]=2 ≥ lista[2]=2 → contador = 1
        j = 3          lista[2]=2 ≥ lista[3]=3 → contador = 1
        j = 4          lista[2]=2 ≥ lista[4]=1 → contador = 2
lista_ordenada[contador] = lista[2];
i = 3
        j = 1          lista[3]=3 ≥ lista[1]=4 → contador = 0
        j = 2          lista[3]=3 ≥ lista[2]=2 → contador = 1
        j = 3          lista[3]=3 ≥ lista[3]=3 → contador = 2
        j = 4          lista[3]=3 ≥ lista[4]=1 → contador = 3
lista_ordenada[contador] = lista[3];
i = 4
        j = 1          lista[4]=1 ≥ lista[1]=4 → contador = 0
        j = 2          lista[4]=1 ≥ lista[2]=2 → contador = 0
        j = 3          lista[4]=1 ≥ lista[3]=3 → contador = 0
        j = 4          lista[4]=1 ≥ lista[4]=1 → contador = 1
lista_ordenada[contador] = lista[4];
lista_ordenada = < 1, 2, 3, 4 >.
```

Figura 3.4 Computação do *Algoritmo 3.5.2*

Exemplo 3.5.3 Sejam $n = 8$ e a lista A= < 2, 8, 3, 7, 9, 1, 6, 5 >. A computação do *Algoritmo 3.5.2* para essa lista é apresentada a seguir:

- *Contagem*: comparação de um dado elemento (lado esquerdo) com cada um dos outros elementos:

i = 1	2:2	2:8	2:3	2:7	2:9	2:1	2:6	2:5
i = 2	8:2	8:8	8:3	8:7	8:9	8:1	8:6	8:5
i = 3	3:2	3:8	3:3	3:7	3:9	3:1	3:6	3:5
i = 4	7:2	7:8	7:3	7:7	7:9	7:1	7:6	7:5
i = 5	9:2	9:8	9:3	9:7	9:9	9:1	9:6	9:5
i = 6	1:2	1:8	1:3	1:7	1:9	1:1	1:6	1:5
i = 7	6:2	6:8	6:3	6:7	6:9	6:1	6:6	6:5
i = 8	5:2	5:8	5:3	5:7	5:9	5:1	5:6	5:5

- *Rank*: determina a posição final de cada elemento (armazenado no contador):

1	0	0	0	0	1	0	0	2
1	1	1	1	0	1	1	1	7
1	0	1	0	0	1	0	0	3
1	0	1	1	0	1	1	1	5
1	1	1	1	1	1	1	1	8
0	0	0	0	0	1	0	0	1
1	0	1	0	0	1	1	1	5
1	0	1	0	0	1	0	1	4

- *Reorganização dos dados*: o *rank* determina a posição do elemento na lista ordenada

A(2)	=	2
A(7)	=	8
A(3)	=	3
A(5)	=	7
A(8)	=	9
A(1)	=	1
A(5)	=	6
A(4)	=	5

- Lista ordenada A = < 1, 2, 3, 5, 6, 7, 8, 9 >

Considere, agora, o algoritmo *mergesort* (*Algoritmo 3.5.3*). Para se classificar uma lista A = < a_1, a_2, ..., a_n> por intercalação, divide-se a lista A em duas sublistas L_1 e L_2, tendo L_1 aproximadamente a metade dos elementos e L_2 a outra metade, ou seja, aproximadamente com o mesmo número de elementos (princípio da equiparação). L_1 = < a_1, a_2, ..., a_k> e L_2 = < a_{k+1},..., a_n>, onde *k* é o menor inteiro que contém a metade da lista. Em seguida, as listas L_1 e L_2 são classificadas separadamente. Essa classificação pode ser obtida, reaplicando o algoritmo *mergesort* nas duas listas e, assim sucessivamente, até que se tenham listas suficientemente pequenas, para que sejam ordenadas diretamente. Obtém-se, portanto, duas sublistas ordenadas L'_1 e L'_2. Essas soluções parciais devem, então, ser intercaladas para se obter a solução final, que pode ser feito, intercalando-se os elementos das duas sublistas ordenadas. Como resultado, obtém-se uma lista resultante ordenada. A intercalação pode ser feita com *n-1* comparações.

ALGORITMO 3.5.3 - Mergesort.

```
Entrada:  O tamanho da lista A e a própria lista A de n elementos
Saída: lista_ordenada - lista classificada em ordem crescente.
inicio
  leia (n)
  leia (A(1:n))
  se n = 1
     então  lista_ordenada := A(1)
     senão inicio
                k := Menor_inteiro{n/2}
                L'1 := mergesort (k, A(1), A(2), ..., A(k))
                L'2 := mergesort (n-k, A(k+1),..., A(n))
                lista_ordenada := intercale(L'1, L'2)
           fim
  escreva (lista_ordenada)
fim
```

No *Algoritmo 3.5.3* a função *menor_inteiro* calcula o maior inteiro menor que o quociente n/2, ela é utilizada para garantir o principio da equiparação; a subrotina *intercale* produz junta duas listas em uma, intercalando os elementos das duas. Observa-se que o algoritmo é recursivo, pois existem duas chamadas do próprio algoritmo (linhas 5 e 6). A lista classificada é disponibilizada em *lista_ordenada*.

Exemplo 3.5.4 Sejam $n = 8$ e a lista A = < 2, 8, 3, 7, 9, 1, 6, 5 >. A computação do *Algoritmo 3.5.3*, para essa lista é dada pela fig. 3.5.

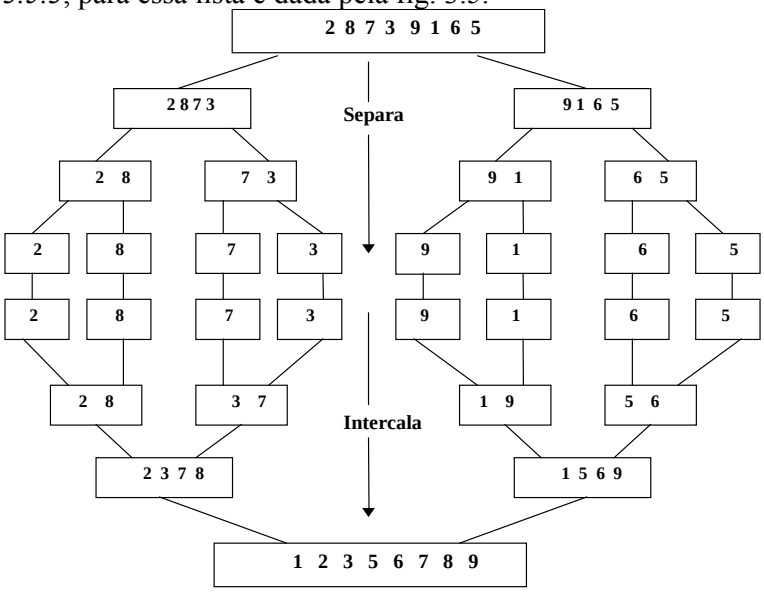


Figura 3.5 Computação do *Algoritmo 3.5.3*

Já o algoritmo *quicksort*, baseado em concatenação, divide a lista em duas sublistas de forma, que após suas classificações não seja necessário intercalá-las, pois se elege um elemento pivô para particionar as listas de forma que uma contenha os valores menores e outra os valores maiores que o pivô. O pior caso é a escolha do pivô sempre como o menor valor (ou maior valor), pois uma lista fica vazia e a outra com todos os valores. Nesse caso a complexidade no pior caso é O (n^2), mas no caso médio a complexidade é O $(n \log n)$.

Existem várias outras formas de se realizar uma classificação, mas os algoritmos mais conhecidos de classificação de listas são o *mergesort* e o *quicksort*.

3.5.1 Algoritmo baseado em memória compartilhada

Existem várias formas de se programar em memória compartilhada. Uma dessas maneiras seria subdividir o programa em tarefas, os quais podem ser executados simultaneamente. A comunicação entre elas se dá através de acessos a variáveis globais. Para garantir a integridade das informações pode ser necessário a definição de regiões de acesso exclusivo. Porém, a maneira mais utilizada, atualmente, para programação paralela através de memória compartilhada são as *threads*, também conhecidas como processos leves. Elas podem ser cópias do programa seqüencial, executadas independentemente em diferentes processadores.

Sabe-se que, o limite da complexidade de tempo da classificação por comparações de uma lista de n elementos, com um único processador, é O($n \log n$), tanto para o pior caso quanto para o caso médio de entradas [TOS2001]. O uso de processadores paralelos, naturalmente, cria a expectativa de reduzir o tempo requerido para realizar tal tarefa. Em caso extremo, quando o número de processadores é assumido ser suficiente para que cada elemento seja simultaneamente comparado com

todos os outros, todas as comparações requeridas podem ser feitas em uma unidade de tempo. Isso simplifica substancialmente a complexidade de qualquer algoritmo baseado em comparação.

A idéia do *Algoritmo 3.5.4* é simples. O comando faça-paralelo baseia-se na idéia de que qualquer *i* dentro do intervalo dado (no caso $0 \leq i \leq n-1$) pode ser calculado de forma independente. Sendo assim, uma solução seria usar uma *thread* para calcular a ordem de cada elemento da lista a ser classificada. A leitura dos elementos na lista de entrada deve ser feita usando algum tipo de mecanismo de sincronização, como por exemplo, um semáforo, para que se evite que mais de uma *thread* calcule a posição de um mesmo elemento. Porém, a escrita do elemento em sua posição correta não exige nenhum tipo de cuidado, pois além da ordenação ser feita em uma nova lista, considera-se que não se tenha elementos repetidos. Dessa forma, cada *thread* faz a escrita em uma posição diferente, não ocorrendo conflitos.

ALGORITMO 3.5.4 – Rank Sort Memória Compartilhada

```
Entrada:  O tamanho n da lista e seus elementos
Saída: A lista classificada em ordem crescente (lista_ordenada)
inicio
 para  1 ≤ i ≤ n  faça-paralelo
                    inicio
                      contador = 0
                       para j = 0 até n-1 faça
                           se lista[i] > lista[j] então contador:= contador +1
                      lista_ordenada[contador]:= lista[i];
                    fim
fim.
```

Exemplo 3.5.5: Sejam $n = 4$ e a lista L = < 4, 2, 3, 1 >. A computação paralela do *Algoritmo 3.5.4*, é apresentada pela fig. 3.6. Foram utilizados quatro processadores, o que possibilitou o uso de quatro *threads*, onde cada uma calculou o *rank* de um dos quatro elementos da lista.

```
THREAD 1
        j = 0    lista[0] > lista[0] → contador = 1
        j = 1    lista[0] > lista[1] → contador = 1
        j = 2    lista[0] > lista[2] → contador = 2
        j = 3    lista[0] > lista[3] → contador = 3
lista_ordenada[contador] = lista[0];
```

```
THREAD 3
        j = 0    lista[2] > lista[0] → contador = 0
        j = 1    lista[2] > lista[1] → contador = 1
        j = 2    lista[2] > lista[2] → contador = 1
        j = 3    lista[2] > lista[3] → contador = 2
lista_ordenada[contador] = lista[2];
```

```
THREAD 2
        j = 0    lista[1] > lista[0] → contador = 0
        j = 1    lista[1] > lista[1] → contador = 0
        j = 2    lista[1] > lista[2] → contador = 0
        j = 3    lista[1] > lista[3] → contador = 1
lista_ordenada[contador] = lista[1];
```

```
THREAD 4
        j = 0    lista[3] > lista[0] → contador = 0
        j = 1    lista[3] > lista[1] → contador = 0
        j = 2    lista[3] > lista[2] → contador = 0
        j = 3    lista[3] > lista[3] → contador = 0
lista_ordenada[contador] = lista[3];
```

Figura 3.6 Computação paralela do *Algoritmo 3.5.4*

A fig. 3.7 apresenta o programa na linguagem C que implementa o algoritmo que classifica listas pelo método do *rank sort*. Observa-se que no *Algoritmo 3.5.2*, os índices *i e j* variavam de *1* a *n*, na implementação apresentada nessa figura, os índices são utilizados, variando de *0* a *n-1*. O programa em C é seqüencial, mas com a criação de *threads* e existindo outros processadores, cada processador executará esse mesmo programa. Se existirem n^2 processadores, a execução, como ilustrada no *Exemplo 3.5.3*, poderá ser feita diretamente.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define n 2048
main(int argc, char* argv [])
        { int i, j, cont;
          int x;
          int vetorDesord[n];
          int vetorOrd[n];
          srandom(getpid());        // inicialização randomica
          for(i=0; i<n; i++) vetorDesord[i] = rand()/100000;
          for(i=0;i<n;i++){ cont = 0;
                    x = vetorDesord[i];
                    for(j=0;j<n;j++)
                        { if(x < vetorDesord[j])  cont++;}
                    vetorOrd [cont] = x;}
```

Figura 3.7 Programa *Rank Sort* em C

3.5.2 Algoritmo baseado em troca de mensagens

A abordagem do algoritmo *rank sort*, para uma arquitetura de memória distribuída, utilizando-se de troca de mensagens, é a *mestre-escravo*. O processador mestre distribui a lista a ser classificada para todos os processadores escravos através de *broadcast* (primitiva de comunicação de grupo), ou seja, cada processador escravo possuirá sua cópia da lista a ser classificada. Sendo n o tamanho da lista e p o número de processadores disponíveis para a classificação, então cada processador deverá encontrar o *rank* de $n/(p-1)$ elementos a serem ordenados. Usa-se $p-1$ por se estar utilizando a abordagem *mestre-escravo*, um processador se dedicará exclusivamente em enviar a lista para os escravos e, no final da computação, recolher os resultados. Atribui-se a cada processador quais elementos ele deve calcular o *rank*, por exemplo, de acordo com sua identificação (que pode variar de *0 a p-1*, no *Algoritmo 3.5.5)*. O algoritmo é apresentado de forma simplificada, assumindo que cada processador calcula a posição de apenas um elemento da lista ($n = p$).

ALGORITMO 3.5.5 – Rank Sort - Troca de Mensagens

```
Entrada: lista desordenada, tamanho da lista, identificador do processador (Id: 0 a p-1)
Saída: lista ordenada em vetorord
Inicio
   cont:= 0;
   se (id = 0)                        //processo mestre (0) envia dados para todos os escravos
     então broadcast (&lista, mestre);
   para i= 0 até n-1 faça                    // calculo do rank dos elementos pelos escravos
         se (vetor[id] > vetor[i])  então cont := cont +1
   se (id = 0)
     então para i := 1 até p-1 faça                          // Tarefa do Mestre
              inicio  recebe(cont,  ANY_SOURCE);
                      vetorord[cont] = vetor[i];
              fim
     senão   envia(cont, mestre)                             // Tarefa do Escravo
fim
```

Exemplo 3.5.6. Tem-se uma lista de tamanho $n = 8$ e têm-se cinco processadores disponíveis (p=5) para a realização da classificação. O processador identificado por *zero* (*id=0*) será o processador mestre. Os outros são os escravos e deverão calcular o *rank* de *2* elementos da lista. O processador identificado por '*um*' calcula as posições dos elementos lista[0] e lista[1], o processador identificado por '*dois*' calcula as posições dos elementos lista[2] e lista[3], e assim por diante. Logo, tem-se que o intervalo de elementos que cada processador irá calcular é [*(1+(id.n/(p-1)))* ; *((id+1).n/(p-1))*]. Caso o

processo zero atue apenas como gerente, essa fórmula necessita ser adaptada, mas isso afetará o desempenho. O *Algoritmo 3.5.5* é baseado em uma abordagem SPMD (*Simple Program Multiple Data*), isto é, todos os processadores executam o mesmo código, porém sobre dados diferentes.

A fig.3.8 ilustra a estrutura de comunicação do *Algoritmo 3.5.5*, onde as flechas que partem do mestre para os escravos representam o *broadcast* da sublista a ser ordenada, pelos processos escravos. A flechas que partem dos escravos para o processador mestre são as mensagens com o *rank* dos elementos, que foram calculados por esse processador.

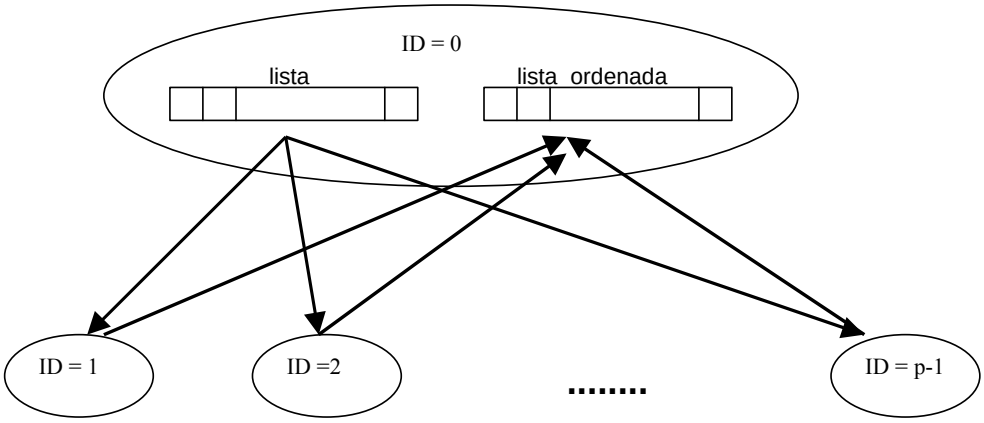


Figura 3.8 Estrutura do *Algoritmo 3.5.5*

```
Vetor original: <2, 8, 3, 7, 9, 1, 6, 5>

// Envio dos dados:
Proc 3 vai mandar o 9 para a posicao 7
Proc 3 vai mandar o 1 para a posicao 0
Proc 2 vai mandar o 3 para a posicao 2
Proc 2 vai mandar o 7 para a posicao 5
Proc 1 vai mandar o 2 para a posicao 1
Proc 1 vai mandar o 8 para a posicao 6
Proc 4 vai mandar o 6 para a posicao 4
Proc 4 vai mandar o 5 para a posicao 3

// Recebimento dos dados
Proc 0 recebeu o valor 2 para ser colocado na posicao 1 (veio do proc 1)
Proc 0 recebeu o valor 8 para ser colocado na posicao 6 (veio do proc 1)
Proc 0 recebeu o valor 6 para ser colocado na posicao 4 (veio do proc 4)
Proc 0 recebeu o valor 5 para ser colocado na posicao 3 (veio do proc 4)
Proc 0 recebeu o valor 3 para ser colocado na posicao 2 (veio do proc 2)
Proc 0 recebeu o valor 7 para ser colocado na posicao 5 (veio do proc 2)
Proc 0 recebeu o valor 9 para ser colocado na posicao 7 (veio do proc 3)
Proc 0 recebeu o valor 1 para ser colocado na posicao 0 (veio do proc 3)

Vetor Ordenado: <1, 2, 3, 5, 6, 7, 8, 9>
```

Figura 3.9 Mensagens enviadas e recebidas pelos processos

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"

main(int argc, char* argv[])
{
        int rank;
        int p;
        int n = 30000;
        int vetor[n];
        int vetorord[n];
        int i, j;
        int x;
        int cont = 0;
        int contrecv = 0;
        int tam, y;
        char buffer[100];
        int position;
        int valor;

        MPI_Status status;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &p);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        if(rank ==0)
          {
                srandom(getpid());
                for(i=0;i<n;i++)  vetor[i]= rand()/1000000;
          }

        MPI_Bcast(&vetor, n, MPI_INT, 0, MPI_COMM_WORLD);      //envia o vetor p/ todos os processos

        tam = n/(p-1);
        if(rank !=0)
                {   y = (rank -1)*tam;
                    for(j=0; j<tam;j++)
                      {
                            valor = vetor[y];
                            cont = 0;
                            position = 0;
                            for(i=0;i<n;i++)
                            {       if(vetor[i]<valor)
                                    cont++;
                            }
                            MPI_Pack(&cont, 1, MPI_INT, buffer, 100, &position, MPI_COMM_WORLD);
                            MPI_Pack(&valor, 1, MPI_INT, buffer, 100, &position, MPI_COMM_WORLD);
                            MPI_Send(buffer, 100, MPI_PACKED, 0, 0, MPI_COMM_WORLD);
                            y++;
                      }
                }
         else
                { for(j=0; j< n; j++)
                  {
        MPI_Recv(buffer,100,MPI_PACKED,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);
                    position = 0;

                    MPI_Unpack(buffer, 100, &position, &cont, 1, MPI_INT, MPI_COMM_WORLD);
                    MPI_Unpack(buffer, 100, &position, &valor, 1, MPI_INT, MPI_COMM_WORLD);

                    vetorord[cont] = valor;
                  }
                }
        MPI_Finalize();
}
```

Figura 3.10 Programa *Rank Sort* Troca de Mensagens em MPI

3.5.3 Considerações sobre complexidade

Sabe-se que o limite de complexidade do problema de classificação é O(*n log n)*, o que significa que não existe algoritmo seqüencial de classificação baseado em comparações de complexidade melhor do que O (*n log n)*.

Foram considerados quatro exemplos de algoritmos de classificação: o *comparação e troca*, o *rank sort*, o *mergesort* e o *quicksort.* Desses quatro algoritmos, considerou-se versões paralelas do algoritmo *rank sort*, com comunicação através da memória compartilhada, usando *threads* e com o uso de troca de mensagens.

Viu-se que a complexidade do *Algoritmo 3.5.1, comparação e troca,* é de O (n^2). A complexidade de tempo do *Algoritmo 3.5.2*, *rank sort* seqüencial também é O(n^2).

A complexidade do *Algoritmo 3.5.3, mergesort sequencial* é O *(n log n),* pois utiliza o princípio da equiparação, ou seja, a lista é dividida em duas listas de aproximadamente do mesmo tamanho. Caso isso não fosse observado, se fossem tomadas duas listas de tamanhos distintos, como por exemplo uma com *um* elemento outra com *n-1* elementos, a complexidade resultante se tornaria O *(n^2)*. Já o algoritmo *quicksort*, no caso médio a complexidade é O *(n log n)*. Entretanto, em seu pior caso, a complexidade é O *(n^2)*.

Os algoritmos mais conhecidos de classificação de listas são o *mergesort* e o *quicksort*. Eles são, usualmente, utilizados para ilustrar as abordagens do pior caso e do caso médio, respectivamente. Cada um, em sua modalidade, é um algoritmo ótimo, ou seja, sua complexidade é a melhor possível (O(*n log n*)).

O problema de classificação está fechado, ou seja, conhece-se a complexidade dos algoritmos seqüenciais com complexidade ótima, O *(n log n*). Portanto, a complexidade de tempo esperada para um algoritmo paralelo de classificação, baseado em um algoritmo seqüencial usando *n* processadores é O *(log n*). Então, já se conhecem algoritmos que realizam o menor número possível de comparações, mesmo assim, deseja-se reduzir o tempo de resolução dos problemas através do processamento paralelo, usando ambientes onde a comunicação ocorre por memória compartilhada ou por troca de mensagens.

A Tab. 3.1 apresenta um resumo da complexidade de algoritmos de classificação paralelos, quanto ao número de processadores e quanto a complexidade de tempo.

Tabela 3.1 Comparação da Complexidade dos Algoritmos de Classificação

Algoritmo Paralelo	Nro de Processadores	Tempo de Execução	Referência
Merge Sort	*n*	O(*n*)	
Quick Sort	*n*	O(*log n*)	
Rank Sort	*n* p^2	O(*n*) O(*log p*)	[WIL99]
Par-Impar de Batcher	$n \log^2 n$	O $(\log^2 n)$	[BAT68]
Bitônico de Batcher	$(n \log^2 n)/2$	O $(\log^2 n)$	[BAT68]
Bitônico de Stone	*n/2*	O $(\log^2 n)$	[STO78]
Muller-Preparata	n^2	O *(log n)*	[MUP75]
Hirschberg	$n^{(1+1/k)}$	O *(k.log n)*	[HIR78]
Preparata	*n log n*	O *(log n)*	[PRE78]
Cole	*n*	O *(log n)*	[COL86]

3.6 Exemplo de multiplicação de matrizes

A multiplicação de matrizes constitui-se de uma das principais operações da álgebra linear por ser utilizada na resolução de vários problemas de diferentes áreas de conhecimento, os quais para serem resolvidos, necessitam uma grande quantidade de operações, sendo por isso, conhecidos como problemas de larga escala de computação. Consequentemente, para resolver esses problemas, necessita-se de máquinas de alto desempenho, os computadores paralelos. Assim, o problema de multiplicar matrizes tornou-se um dos problemas clássicos do processamento paralelo e pode ser descrito como segue.

Sejam A e B duas matrizes de ordem *nxn* ($n \geq 2$), com elementos de um corpo qualquer (por exemplo, números reais). A matriz produto C = A·B é, por definição, uma matriz *nxn* cujo elemento na linha *i* e coluna *j* é obtido a partir de *i*-ésima linha de A e da *j*-ésima coluna de B, através do produto escalar, $c_{ij} := \Sigma a_{ik} \cdot b_{kj}$ (para $k = 1$ até n).

É fácil observar que são necessárias *n* multiplicações para se obter c_{ij}. Como a matriz C possui n^2 destes elementos c_{ij}, serão necessárias n^3 multiplicações para se determinar C. A seguir será apresentado um exemplo simples, um produto de duas matrizes 3x3, com valores inteiros.

Exemplo 3.6.1: Sejam A e B as matrizes 3x3 abaixo, a matriz produto C é calculada pela definição matemática.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ -2 & 4 & 0 \\ 1 & 5 & 3 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 0 & 5 \\ 1 & 4 & 3 \\ 1 & 0 & -1 \end{bmatrix}$$

$$C = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ -2 & 4 & 0 \\ 1 & 5 & 3 \end{bmatrix} \times \begin{bmatrix} 2 & 0 & 5 \\ 1 & 4 & 3 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 7 & 8 & 8 \\ 0 & 16 & 2 \\ 10 & 20 & 17 \end{bmatrix}$$

Figura 3.11 Produto de matrizes pela definição

Seis variações podem ser obtidas da definição da multiplicação de matrizes, realizando-se permutações na ordem dos laços (rearranjo dos índices). Cada uma dessas variações difere na maneira como as matrizes dos elementos são acessadas. Cada permutação caracteriza seu próprio modelo de acesso à memória e de como produz os valores parciais, o que influência o desempenho do algoritmo que a descreve.

ALGORITMO 3.6.1 Multiplicação de Matrizes (ordem ijk)

```
Entrada: A ordem n e as duas matrizes quadradas A e B.
Saída: A matriz produto C.
inicio
  para i =1 até n faça
     para j =1 até n faça c(i,j) :=0
  para i =1 até n faça
     para j =1 até n faça
        para k =1 até n faça c(i,j) := c(i,j) + a(i,k).b(k,j)
fim
```

A computação do *Algoritmo 3.6.1* - multiplicação de matrizes na ***forma ijk*** - é apresentada a seguir. Observa-se que cada elemento da matriz produto é calculado quando o algoritmo conclui um laço interno. A ***forma ijk***, calcula os elementos da primeira linha, depois da segunda e, por fim, a terceira linha, enquanto que a ***forma jik*** (troca de ordem dos laços *i* e *j* no *Algortimo 3.6.1*), calcula os elementos da matriz produto por coluna, ou seja, a primeira coluna, a segunda e, então a terceira. Em ambas

as versões cada elemento da matriz produto é calculado de forma integral. Ou seja, cada laço de *k* (interno) realiza um produto escalar (multiplicação de dois vetores, cujo resultado é um escalar, contendo as somas dos produtos). Essa característica será estudada para a exploração do paralelismo na multiplicação de matrizes.

	j =1	j =2	j =3
i = 1	$a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31}$ = 1x2 + 2x1 + 3x1 = 7	$a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32}$ = 1x0 + 2x4 + 3x0 = 8	$a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33}$ = 1x5 + 2x3 + 3x(-1) = 8
i = 2	$a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31}$ = (-2)x2 + 4x1 + 0x1 = 0	$a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32}$ = (-2)x0 + 4x4 + 0x0 = 16	$a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33}$ = (-2)x5 + 4x3 + 0x(-1) = 2
i = 3	$a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31}$ = 1x2 + 5x1 + 3x1 = 10	$a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32}$ = 1x0 + 5x4 + 3x0 = 20	$a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33}$ = 1x5 + 5x3 + 3x(-1) = 17

Figura 3.12 Computação do produto de matrizes ijk

A ***forma kij*** (igualmente a ***forma kji***) é uma variação que produz a cada laço mais externo (laço *k*) uma matriz produto parcial, onde em cada elemento dessa matriz é acumulado o valor parcial da soma do produto escalar. A soma das parcelas pode ser efetuada como soma de matrizes. Identifica-se um paralelismo intrínseco nessa estrutura. O *Algoritmo 3.6.2* descreve o método e, logo a seguir, apresentada-se sua computação para as matrizes do *Exemplo 3.6.1*. A complexidade é O (n^3).

ALGORITMO 3.6.2 Multiplicação de Matrizes (ordem kij)

```
Entrada: A ordem n e as duas matrizes quadradas A e B.
Saída: A matriz produto C.
inicio
  para i =1 até n faça
      para j =1 até n faça c(i,j) := 0
  para k =1 até n faça
     para i =1 até n faça
        para j =1 até n faça c(i,j) := c(i,j) + a(i,k).b(k,j)
fim
```

k =1	***j =1***		***j =2***		***j =3***	
i = 1	$a_{11}b_{11}$	1x2	$a_{11}b_{12}$	1x0	$a_{11}b_{13}$	1x5
i = 2	$a_{21}b_{11}$	(-2)x2	$a_{21}b_{12}$	(-2)x0	$a_{21}b_{13}$	(-2)x5
i = 3	$a_{31}b_{11}$	1x2	$a_{31}b_{12}$	1x0	$a_{31}b_{13}$	1x5

k =2	***j =1***		***j =2***		***j =3***	
i = 1	$a_{12}b_{21}$	2x1	$a_{12}b_{22}$	2x4	$a_{12}b_{23}$	2x3
i = 2	$a_{22}b_{21}$	4x1	$a_{22}b_{22}$	4x4	$a_{22}b_{23}$	4x3
i = 3	$a_{32}b_{21}$	5x1	$a_{32}b_{22}$	5x4	$a_{32}b_{23}$	5x3

k =3	***j =1***		***j =2***		***j =3***	
i = 1	$a_{13}b_{31}$	3x1	$a_{13}b_{32}$	3x0	$a_{13}b_{33}$	3x(-1)
	c_{11} = 7		**c_{12} = 8**		**c_{13} = 8**	
i = 2	$a_{23}b_{31}$	0x1	$a_{23}b_{32}$	0x0	$a_{23}b_{33}$	0x(-1)
	c_{21} = 0		**c_{22} = 16**		**c_{23} = 2**	
i = 3	$a_{33}b_{31}$	3x1	$a_{33}b_{32}$	3x0	$a_{33}b_{33}$	3x(-1)
	c_{31} = 10		**c_{32} = 20**		**c_{33} = 17**	

Figura 3.13 Computação do produto de matrizes kij

A ***forma ikj*** calcula a cada laço interno um dos produtos parciais para cada elemento da linha *i*; a cada laço intermediário é calculado nova parcela (novo produto parcial) do produto escalar, ao final do laço são acumuladas as parcelas recém calculadas, produzindo elementos da matriz produto. Os elementos da matriz produto são calculados por linha, sendo que a obtenção de cada elemento só é concluída após a realização de dois dos laços. A descrição dessa forma é dada pelo *Algoritmo 3.6.3* e sua computação é exemplificada com os dados do *Exemplo 3.6.1*.

ALGORITMO 3.6.3 Multiplicação de Matrizes (ordem ikj)

```
Entrada: A ordem n e as duas matrizes quadradas A e B.
Saída: A matriz produto C.
inicio
   para i =1 até n faça
      para j =1 até n faça c(i,j) := 0
   para i =1 até n faça
       para k =1 até n faça
           para j =1 até n faça  c(i,j) := c(i,j) + a(i,k).b(k,j)
fim
```

i =1	***j =1***		***j =2***		***j =3***	
k = 1	$a_{11}b_{11}$	1x2	$a_{11}b_{12}$	1x0	$a_{11}b_{13}$	1x5
k = 2	$+a_{12}b_{21}$	+ 2x1	$+ a_{12}b_{22}$	+ 2x4	$+ a_{12}b_{23}$	+ 2x3
k = 3	$+ a_{13}b_{31}$	+ 3x1	$+ a_{13}b_{32}$	+ 3x0	$+ a_{13}b_{33}$	+ 3x(-1)
c_{ij}		**$c_{11} = 7$**		**$c_{12} = 8$**		**$c_{13} = 8$**

i =2	***j =1***		***j =2***		***j =3***	
k = 1	$a_{21}b_{11}$	(-2)x2	$a_{21}b_{12}$	(-2)x0	$a_{21}b_{13}$	(-2)x5
k = 2	$+ a_{22}b_{21}$	+ 4x1	$+ a_{22}b_{22}$	+ 4x4	$+ a_{22}b_{23}$	+ 4x3
k = 3	$+ a_{23}b_{31}$	+ 0x1	$+ a_{23}b_{32}$	+ 0x0	$+ a_{23}b_{33}$	+ 0x(-1)
c_{ij}		**$c_{21} = 0$**		**$c_{22} = 16$**		**$c_{23} = 2$**

i =3	***j =1***		***j =2***		***j =3***	
k = 1	$a_{31}b_{11}$	1x2	$a_{31}b_{12}$	1x0	$a_{31}b_{13}$	1x5
k = 2	$+ a_{32}b_{21}$	+ 5x1	$+ a_{32}b_{22}$	+ 5x4	$+ a_{32}b_{23}$	+ 5x3
k = 3	$+ a_{33}b_{31}$	+ 3x1	$+ a_{33}b_{32}$	+ 3x0	$+ a_{33}b_{33}$	+ 3x(-1)
c_{ij}		**$c_{31} = 10$**		**$c_{32} = 20$**		**$c_{33} = 17$**

Figura 3.14 Computação do produto de matrizes ikj

A ***forma jki***, analogamente, calcula os elementos por coluna, mas para obtenção de cada elemento da matriz produto, é necessário a conclusão dos dois laços internos.

Outra metodologia para o desenvolvimento de algoritmos para o cálculo da multiplicação de matrizes consiste na utilização da *técnica de divisão e conquista*. A divisão e conquista se baseia na recursão, divide-se o problema em subproblemas e reaplica o método nos subproblemas, a tal ponto que esses sejam resolvidos diretamente. Por fim, combinam-se todas as soluções parciais para se obter a solução do problema original.

O algoritmo baseado nessa técnica, consiste em particionar as matrizes A e B em quatro submatrizes quadradas, cada uma delas de dimensão *n/2 x n/2* (supondo que *n* seja uma potência de *2*). Então o produto C, pode ser calculado por:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

onde $C_{ij} := A_{i1}.B_{1j} + A_{i2}.B_{2j}$, para $i \, e \, j \in [1, 2]$.

Nos cálculos dos C_{ij}, necessita-se efetuar multiplicações das submatrizes, pode-se aplicar recursivamente o particionamento, obtendo assim quatro novas submatrizes, de dimensão $n/4 \times n/4$, esses particionamentos sucessivos resultarão em vários casos de multiplicações de matrizes $2x2$, que podem ser efetuados diretamente. Ou, pode-se efetuar diretamente as multiplicações, sendo então necessárias *8* multiplicações de matrizes e *4* adições de matrizes. Considerando que multiplicações requerem mais tempo para serem efetuadas do que somas matriciais, pode-se reformular o cálculo dos C_{ij}, resultando no algoritmo de *Strassen*:

$C_{11} = D_5 + D_2 - D_3 + D_6$
$C_{12} = D_1 + D_3$
$C_{21} = D_4 + D_2$
$C_{22} = D_5 + D_1 - D_4 + D_7$

onde,

$D_1 = A_{11} (B_{12} - B_{22})$
$D_2 = A_{22} (B_{21} - B_{11})$
$D_3 = (A_{11} + A_{12}) B_{22}$
$D_4 = (A_{21} + A_{22}) B_{11}$
$D_5 = (A_{11} + A_{22}).(B_{11} + B_{22})$
$D_6 = (A_{12} - A_{22}).(B_{21} + B_{22})$
$D_7 = (A_{21} - A_{11}).(B_{11} + B_{12})$

A complexidade do *Algoritmo de Strassen* é dada pela equação de recorrência:

$$c[Strassen](n) = \begin{cases} O(1), & \text{se } n \leq 2 \\ O(n^2) + 7c[Strassen](n/2), & \text{se } n > 2 \end{cases}$$

cuja solução é $c[Strassen](n) = n^{\lceil \log_2 7 \rceil}$. Logo, esse algoritmo é assintoticamente mais rápido do que o algoritmo trivial ($O(n^3)$). Entretanto, nada se pode afirmar sobre a qualidade dos resultados, uma vez que não foram desenvolvidos estudos sobre a qualidade numérica dos resultados em função dos arredondamentos.

3.6.1 Algoritmo baseado em memória compartilhada

Nas versões decorrentes das variações dos índices, o paralelismo pode ser explorado na forma como os elementos da matriz produto são calculados. No *Algoritmo 3.6.1*, a cada laço interno, um elemento é calculado através do produto escalar de dois vetores. Esses cálculos são independentes e podem ser calculados em paralelo. No *Algoritmo 3.6.3* quando os laços internos são computados, são produzidos um vetor de elementos da matriz produto. Já no *Algoritmo 3.6.2*, a estrutura gera matrizes com parcelas resultantes de produtos que constituem o produto escalar, os quais só são completados quando os três laços são concluídos, entretanto, a estrutura matricial pode facilitar o desenvolvimento do algoritmo paralelo, pois todos os produtos são síncronos e são seguidos de somas de matrizes.

A seguir será discutido a questão da paralelização do produto escalar. Será considerado o algoritmo paralelo que efetua a multiplicação de matrizes, como *n* produtos concorrentemente da matriz por vetor, o que é uma interpretação do *Algoritmo 3.6.3.* A versão paralela desse algoritmo está baseada na arquitetura da máquina paralela, ou seja, no número de processadores disponíveis e na forma como eles estão interconectados. Outra questão importante é sobre a organização da memória, se cada

processador tem ou não sua memória local, as formas e os canais de acesso a essa memória, pois conflitos de acesso à memória podem retardar a execução de um programa paralelo e comprometer o seu desempenho.

Sejam A e B matrizes de ordem *n* e, seja X um vetor de ordem *n*, (obtido entre as *n* colunas de B), ambos armazenados na memória compartilhada. Será assumido que o número de processadores *p* seja menor ou igual à dimensão *n* ($p \leq n$), tal que o quociente $^{n}/_{p} = r$ seja um inteiro e que o modo de operação seja assíncrono. A matriz é particionada por blocos, onde cada bloco A_i é do tamanho *rxn*. O problema de computar o produto Y= A·X, onde Y é uma das colunas da matriz produto, pode ser calculado como segue. Cada processador P_i lê A_i e X_i da memória compartilhada; calcula $Z_i = A_i \cdot X_i$ e armazena seus *r* componentes de Z nas componentes apropriadas da variável compartilhada Y.

No *Algoritmo 3.6.4* é utilizado A(*l:u, s:t*) para denotar a submatriz de A constituída das linhas *l, l+1, ..., u* e das colunas *s, s+1, ...,t*. A mesma notação será usada para indicar um subvetor de um dado vetor.

ALGORITMO 3.6.4 Multiplicação de Matriz por Vetor

Entrada: Uma matriz nxn e um vetor X de ordem n residentes na memória compartilhada;
Variáveis locais: a ordem n, a identificação do processador e nro de processadores p onde p ≤ n tal que r = n/p seja um inteiro.
Saída: Componentes (i-1).r+1, ...,ir do vetor Y=AX armazenado na variável compartilhada Y.
inicio
1. leia (X, w)
2. leia (A(1:n ,(i-1)r+1:ir),B)
3. ***compute*** z = B.w
4. escreva (z,y((i-1)r+1:ir))

fim

Cada processador executa o mesmo algoritmo. Observe que a leitura (leia) é realizada concorrentemente por todos os *p* processadores no passo 1, pois todos lêem a variável compartilhada X. Entretanto, a escrita não é concorrente, pois não há dois processadores escrevendo na mesma posição da memória compartilhada.

Uma característica importante do *Algoritmo 3.6.4* é que os processadores não necessitam ter suas atividades sincronizadas, uma vez que o produto da matriz pelo vetor foi particionado. Por outro lado, pode-se projetar um algoritmo paralelo baseado no particionamento de A e de X em *p* blocos, tais que $A = (A_1, A_2, .., A_p)$, onde cada A_i é do tamanho *nxr* e $X = (X_1, X_2, .., X_p)$, onde cada X_i é do tamanho *r*. O produto Y=A·X é calculado por $Y = A_1 \cdot X_1 + A_2 \cdot X_2 + ... + A_p \cdot X_p$. Portanto, o processador P_i calcula $z_i = A_i \cdot X_i$ depois de ter A_i e X_i da memória compartilhada, para $1 \leq i \leq p$.

Nenhum processador pode realizar a soma $z_1 + z_2 + ...+z_r$ antes que todos tenham terminado seus produtos de matriz por vetor. Portanto, uma primitiva explícita de sincronização necessita ser adicionada no programa depois de cada processador ter calculado seu $z_i = A_i \cdot X_i$, para forçar que todos os processadores sincronizem antes que continuem a execução de seus programas. A fig.3.15 ilustra o método.

Considere, agora, o problema de calcular o produto C das duas matrizes A e B, de ordem *n*, onde $n=2^k$, para algum inteiro *k*. Suponha que existam n^3 processadores disponíveis na máquina paralela, denotados por $P_{i,j,l}$ onde $1 \leq i, j, l \leq n$. O processador $P_{i,j,l}$ calculará o produto $A(i, l) \cdot B(l, j)$ em um único passo. Então para cada par (i, j), os *n* processadores $P_{i,j,l}$ onde $1 \leq l \leq n$ calculam a soma:

$$\sum_{l=1}^{n} A(i, l) \cdot B(l, j)$$ como descrito no exemplo anterior.

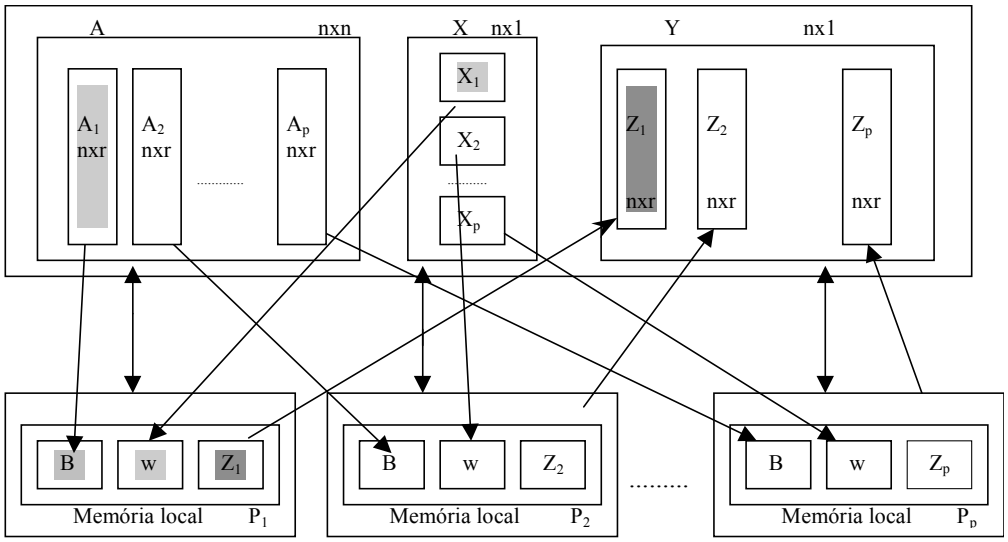


Figura 3.15 Ilustração do *Algoritmo 3.6.4* em uma PRAM.

ALGORITMO 3.6.5 Multiplicação de Matrizes - processador P_{ijl}
Entrada: Duas matrizes A e B de ordem nxn armazenadas em memória compartilhada ($n=2^k$);
variáveis locais são inicializadas com n e a tripla (i,j,l) identifica o processador.
Saída: A matriz produto C=AB, armazenada em memória compartilhada.
inicio
compute C'(i, j, l) = A(i,l).B(l,j)
para h=1 até (log n) faça
se (l ≤ n/2h) então *compute* C'(i,j,l):= C'(i,j,2l-1)+ C'(i,j,2l)
se (l=1) então *compute* C(i,j) := C'(i,j,1)
fim

O *Algoritmo 3.6.5* define a computação do processador $P_{i,j,l}$. Observe que é requerida a capacidade de leitura concorrente (CREW), logo diferentes processadores podem acessar os mesmos dados na execução do passo 1. Por exemplo, os processadores $P_{i,1,l}$, $P_{i,2,l}$, ..., $P_{i,n,l}$, todos requerem os valores A(i,l) de memória compartilhada na execução do passo 1. O tempo de execução é da ordem de O(*log n*) passos paralelos.

3.6.2 Algoritmo baseado em troca de mensagens

As ***formas ikj*** e ***jki*** podem ser transformados em algoritmos paralelos as quais fazem uso de cópias rápidas (*broadcasting*), facilitando as máquinas paralelas.

Seleciona-se um vetor de cada uma das duas matrizes (uma linha e uma coluna, por exemplo). Duplica-se esses vetores *n* vezes, para construir uma nova matriz, onde cada elemento é a multiplicação dos componentes correspondentes. Observa-se que os vetores são escolhidos, de tal forma que, os elementos (*i,j*) dessas matrizes são as parcelas do cálculo do produto escalar. Por exemplo, inicialmente pega-se a primeira coluna de A e a primeira linha de B, duplica-se esses vetores, monta-se a primeira matriz parcela, onde cada elemento dessa matriz, corresponde a um dos produtos que é necessário ser efetuado para o cálculo do produto escalar que calcula cada elemento da matriz produto, ou seja,

$$\begin{matrix} a_{11} & a_{11} & a_{11} \\ a_{21} & a_{21} & a_{21} \\ a_{31} & a_{31} & a_{31} \end{matrix} \quad \begin{matrix} b_{11} & b_{12} & b_{13} \\ b_{11} & b_{12} & b_{13} \\ b_{11} & b_{12} & b_{13} \end{matrix} \quad \Rightarrow \quad \begin{matrix} a_{11}b_{11} & a_{11}b_{12} & a_{11}b_{13} \\ a_{21}b_{11} & a_{21}b_{12} & a_{21}b_{13} \\ a_{31}b_{11} & a_{31}b_{12} & a_{31}b_{13} \end{matrix}$$

Repete-se o processo, pegando a segunda coluna da A e a segunda linha de B, montando-se a segunda matriz parcela, que na verdade, será acumulada a primeira.

$$\begin{matrix} a_{12} & a_{12} & a_{12} \\ a_{22} & a_{22} & a_{22} \\ a_{32} & a_{32} & a_{32} \end{matrix} \quad \begin{matrix} b_{21} & b_{22} & b_{23} \\ b_{21} & b_{22} & b_{23} \\ b_{21} & b_{22} & b_{23} \end{matrix} \quad \Rightarrow \quad \begin{matrix} a_{12}b_{21} & a_{12}b_{22} & a_{12}b_{23} \\ a_{22}b_{21} & a_{22}b_{22} & a_{22}b_{23} \\ a_{32}b_{21} & a_{32}b_{22} & a_{32}b_{23} \end{matrix}$$

A soma acumulada resultante é matriz soma:

$$\begin{matrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} & a_{11}b_{13} + a_{12}b_{23} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} & a_{21}b_{13} + a_{22}b_{23} \\ a_{31}b_{11} + a_{32}b_{21} & a_{31}b_{12} + a_{32}b_{22} & a_{31}b_{13} + a_{32}b_{23} \end{matrix}$$

Repete-se o processo, pegando a terceira coluna da A e a terceira linha de B, montando-se a terceira matriz parcela, que, também, será acumulada, produzindo a matriz produto.

$$\begin{matrix} a_{13} & a_{13} & a_{13} \\ a_{23} & a_{23} & a_{23} \\ a_{33} & a_{33} & a_{33} \end{matrix} \quad \begin{matrix} b_{31} & b_{32} & b_{23} \\ b_{31} & b_{32} & b_{23} \\ b_{31} & b_{32} & b_{23} \end{matrix} \quad \Rightarrow \quad \begin{matrix} a_{13}b_{31} & a_{13}b_{32} & a_{13}b_{33} \\ a_{23}b_{31} & a_{23}b_{32} & a_{23}b_{33} \\ a_{33}b_{31} & a_{33}b_{32} & a_{33}b_{33} \end{matrix}$$

A soma acumulada resultante é matriz produto (no caso de matrizes de ordem $n=3$):

$$\begin{matrix} a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31} & a_{11}b_{12}+a_{12}b_{22}+a_{13}b_{32} & a_{11}b_{13}+a_{12}b_{23}+a_{13}b_{33} \\ a_{21}b_{11}+a_{22}b_{2}+a_{23}b_{31} & a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32} & a_{21}b_{13}+a_{22}b_{23}+a_{23}b_{33} \\ a_{31}b_{11}+a_{32}b_{2}+a_{13}b_{31} & a_{31}b_{12}+a_{32}b_{22}+a_{33}b_{32} & a_{31}b_{13}+a_{32}b_{23}+a_{33}b_{33} \end{matrix}$$

ALGORITMO 3.6.6 Multiplicação de Matriz

```
Entrada: Duas matrizes A e B  nxn.
Saída: O produto C = AB armazenado na memória compartilhada.
inicio
  faça-paralelo {1≤ i ≤ n}
    faça-paralelo {1≤ j ≤ n}
        temp(i,j):=0
        para k:=1 to n faça
           inicio matrixfa(i,j):=a(i,k)
                  matrixfb(i,j):=b(k,j)
                  temp(i,j):= temp(i,j)+matrixfa(i,j)×matrixfb(i,j)
            fim
         C(i,j):= temp(i,j)
     fim-paralelo
  fim-paralelo
fim
```

Genericamente, o processo é repetido n vezes, até que todas as matrizes soma são montadas e acumuladas, produzindo o cálculo completo da matriz produto. Determinando-se a complexidade de tempo e espaço desse um algoritmo, observa-se que novamente o tempo é de $O(n)$ e o espaço $O(n^2)$.

Cada um desses algoritmos do produto de matrizes utiliza características específicas de processamento paralelo. Pode acontecer que para uma máquina específica não seja possível implementar eficientemente algumas dessas características utilizadas nesses algoritmos e isso pode resultar em uma implementação, cujo o desempenho seja ineficiente.

Considere, agora, a versão do *Algoritmo 3.6.4* em um ambiente baseado em troca de mensagens (*Algoritmo 3.6.7*), esse algoritmo foi implementado em MPI e seu código é apresentando na fig.3.16. Dados uma matriz A nxn e um vetor de $O(n)$, considere o problema de calcular o produto da matriz pelo vetor $Y = A \cdot X$ em um *arquitetura de anel* de p processadores, onde $p \leq n$. Assume-se que p divide n e, finalmente, seja $r = n/p$.

Se A e X são particionados em p blocos, segue que $A = (A_1, A_2, .., A_p)$ e $X = (X_1, X_2, .., X_p)$, onde A_i é de tamanho nxr e cada X_i é de tamanho r. Pode-se determinar o produto $Y= A \cdot X$ pelo cálculo de $z_i = A_i \cdot X_i$, para $1 \leq i \leq p$ e, então, realizando a soma $z_1+z_2+....+z_p$.

O processador P_i da rede faz inicialmente $B= A_i$ e $W= X_i$ na sua memória local, para todo $1 \leq i \leq p$. Então cada processador pode calcular localmente o produto $B \cdot W$ e a soma desses vetores pode ser calculada através da circulação das somas parciais através do *array cíclico*. O vetor de saída será armazenado em P_1. Esse raciocínio é implementado no *Algoritmo 3.6.7.*

ALGORITMO 3.6.7 Produto assíncrono de matriz por vetor em arquitetura anel.

Entrada: Número do processador; o número de processadores p, a i-ésima submatriz B = A(1:n, (i-1).(r+1): ir) de tamanho nxr (onde r = n / p) e o i-ésimo subvetor W= x((i-1).r+1:ir) de tamanho r.

Saída: Processador P_i calcula o vetor $Y=A_1x_1 +A_2x_2 +.... +A_ix_i$ e passa o resultado à direita. Quando o algoritmo termina, P_1 terá o produto Ax.

```
inicio
   compute produto(Z = B.W)
   se  i=1 então Y:=0 senão recebe(Y, esquerdo)
   Y:=Y+Z
   envia(Y,direito)
   se i=1 então recebe(Y, esquerdo)
fim
```

Cada processador P_i começa computando $A_i \cdot X_i$ e armazena o vetor resultante na variável local z. Depois do passo 2, o processador P_1 inicializa o vetor Y como zero, onde cada um dos outros processadores suspende a execução de seus programas, aguardando o recebimento de dados dos seu vizinho esquerdo. O processador P_1 atribui $Y=A_1 \cdot X_1$ e envia o resultado para seu vizinho da direita nos passos 3 e 4 (respectivamente). Então, o processador P_2 recebe $A_1 \cdot X_1$ e conclui a execução de seu programa computando $A_1 \cdot X_1 + A_2 \cdot X_2$ no passo 3. Ele então envia o novo vetor para o processador vizinho à direita no passo 4. Quando a execução de todos os programas termina, o processador P_1 terá o produto $Y=A \cdot X$.

A computação efetuada por cada processador consiste de duas operações nos passos 1 e 3; portanto, o tempo de execução do algoritmo é $O(n^2/p)$. Por outro lado, o processador P_1 tem que esperar até que a soma parcial $A_1 \cdot X_1+.....+A_p \cdot X_p$ tenha sido acumulada antes de executar o passo 5. Entretanto, o tempo total de comunicação é proporcional ao produto $p.comm(n)$, onde $comm(n)$ é o tempo gasto para se transmitir n números entre processadores adjacentes. Esse valor pode ser aproximado por $comm(n) = \sigma + n\tau$, onde σ é o tempo de início de transmissão e τ é o intervalo de tempo necessário para a mensagem ser transferida. A complexidade do algoritmo é a soma do tempo de execução e do tempo de comunicação. Esse tempo é minimizado, quando se tem $p= n\sqrt{\alpha}/(\sigma+n\tau)$.

```
#include <string.h>
#include "mpi.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define DIME 21
#define MAX_BUF 1000000
MPI_Status Status;
int my_id, size, posicao;
float *matriz, *vetor, *vetorr, *pvetor, *pvetor2;
float *mt;
char  bpack[MAX_BUF];
```

```
void calcula(int tamp, float *M, float *V, float *VR)
{ int v1, v2, v3, vf1;
  v3 = 0;
  for (v1=0;v1<DIME;v1++) { for (v2=0;v2<tamp;v2++) {  VR[v1]+= V[v2]*M[v3];  v3 ++;  } } }
void  carrega_matriz()
{ int cont1, cont2, tamanho, em, pm;
  tamanho = DIME * DIME;
  matriz=malloc(tamanho*sizeof(float));
  vetor=malloc(DIME*sizeof(float));
  vetorr=malloc(DIME*sizeof(float));
  pm = 0;   em = 0;
  for (cont1=0;cont1<DIME;cont1++) {em++; for (cont2=0;cont2<DIME;cont2++)  {matriz[pm] = em; pm ++;} }
  for (cont1=0;cont1<DIME;cont1++) vetor[cont1] = 1;
  for (cont1=0;cont1<DIME;cont1++) vetorr[cont1] = 0; }
void  recebe_result()
{ int nd, ls;
  pvetor2 =malloc(DIME*sizeof(float));
  for (nd=1;nd<size;nd++)
     { MPI_Recv(&bpack,MAX_BUF,MPI_PACKED,MPI_ANY_SOURCE, MPI_ANY_TAG,MPI_COMM_WORLD,&Status);
       posicao = 0;
       MPI_Unpack(&bpack,MAX_BUF,&posicao,pvetor2,DIME, MPI_FLOAT,MPI_COMM_WORLD);
       for (ls=0;ls<DIME;ls++) vetorr[ls] = vetorr[ls] + pvetor2[ls]; }
  free(pvetor2);  free(vetorr);  }
void  distribui_matriz()
{ int nll, sll, taml, nd;
  div_t partes;
int vf1, vf2, posi, apo;
partes  = div(DIME,size);
 nll     = partes.quot;
 sll     = partes.rem;
 taml = (nll+sll);
mt=malloc((taml*DIME)*sizeof(float));
pvetor=malloc((nll+sll)*sizeof(float));
for (nd=1;nd<size;nd++)
     { apo = 0;  posi = (sll+(nll*nd));
       for (vf1=0;vf1<DIME;vf1++){ for (vf2=0;vf2<nll;vf2++) { mt[apo]= matriz[posi]; apo++; posi++;} posi= posi+DIME-nll;}
       posi = nd * nll;
       for (vf1=0;vf1<nll;vf1++) { pvetor[vf1]  = vetor[posi]; posi++;}
       posicao = 0;
       MPI_Pack(&nll,1,MPI_INT,&bpack,MAX_BUF,&posicao,MPI_COMM_WORLD);
       MPI_Pack(mt,(DIME*nll),MPI_FLOAT,&bpack,MAX_BUF,&posicao,MPI_COMM_WORLD);
       MPI_Pack(pvetor,nll,MPI_FLOAT,&bpack,MAX_BUF,&posicao,MPI_COMM_WORLD);
       MPI_Send(&bpack,posicao,MPI_PACKED,nd,0,MPI_COMM_WORLD); }
  apo = 0;  posi = 0;
  for (vf1=0;vf1<DIME;vf1++)
       {  for (vf2=0;vf2<taml;vf2++) {mt[apo]  = matriz[posi]; apo++; posi++;}  posi = posi + DIME - taml; }
  for (vf1=0;vf1<taml;vf1++) pvetor[vf1]  = vetor[vf1];
   free(matriz);   free(vetor);
   calcula(taml,mt,pvetor,vetorr);
   free(mt);    free(pvetor); }
main(int argc, char **argv)
{  int nlp, aux;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
   if (my_id == 0)                                   /* esta parte do codigo sera executado pelo processo zero */
        { carrega_matriz(); distribui_matriz();  recebe_result();  }
    else                                             /* esta parte sera executada pelos demais processos */
     {MPI_Recv(&bpack,MAX_BUF,MPI_PACKED,MPI_ANY_SOURCE, MPI_ANY_TAG,MPI_COMM_WORLD, &Status);
       posicao = 0;
       MPI_Unpack(&bpack,MAX_BUF,&posicao,&nlp,1,MPI_INT,MPI_COMM_WORLD);
       aux = nlp*DIME;
       mt=malloc(aux*sizeof(float));
       vetor=malloc(nlp*sizeof(float));
       vetorr=malloc(DIME*sizeof(float))
       MPI_Unpack(&bpack,MAX_BUF,&posicao,mt,aux,MPI_FLOAT,MPI_COMM_WORLD);
       MPI_Unpack(&bpack,MAX_BUF,&posicao,vetor,nlp,MPI_FLOAT,MPI_COMM_WORLD);
       calcula(nlp,mt,vetor,vetorr);
       posicao = 0;
       MPI_Pack(vetorr,DIME,MPI_FLOAT,&bpack,MAX_BUF,&posicao,MPI_COMM_WORLD);
       MPI_Send(&bpack,posicao,MPI_PACKED,0,0,MPI_COMM_WORLD);
       free(vetor);    free(mt); }
       MPI_Finalize( ); }
```

Figura 3.16 Versão Matriz-Vetor em MPI.

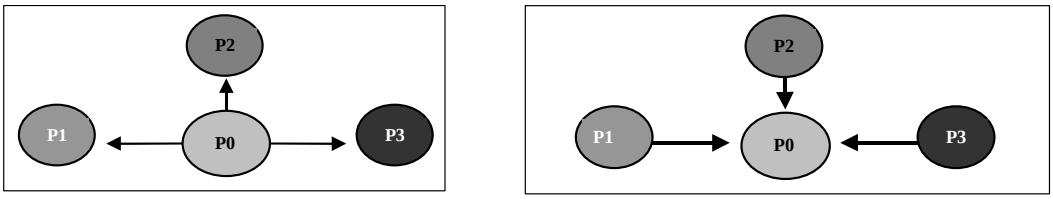
Exemplo 3.6.2 Multiplicação de Matrizes baseado em troca de mensagens. Considere o produto da matriz M *8x8*, por ela mesma (MxM), descrita na figura 3.17. O programa baseado em troca de mensagens, descrito na fig.3.16, realiza apenas o produto de uma matriz por um vetor. Ter-se-ia que executar o programa *8* vezes, uma para cada coluna, para se ter o produto completo. Considere que existam 4 processadores disponíveis (*p=4*), seus identificadores são *0, 1, 2, 3*. O processador *0* possui o domínio completo. Ele divide-o entre *4* processadores, como mostra a fig.3.17.

P0		P1		P2		P3	
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

P0	1	2	3	4	5	6	7	8
	9	10	11	12	13	14	15	16
P1	17	18	19	20	21	22	23	24
	25	26	27	28	29	30	31	32
P2	33	34	35	36	37	38	39	40
	41	42	43	44	45	46	47	48
P3	49	50	51	52	53	54	55	56
	57	58	59	60	61	62	63	64

Figura 3.17 Exemplo de Multiplicação de Matrizes pelo programa da fig 3.16

Após a divisão, o processador 0 distribui os subdomínios entre os processadores, ficando com um subdomínio para processar. A distribuição é feita por mensagens de envio (*send*), como ilustrado na fig.3.18 (a).



(a)Distribuição de dados - *send* **(b)Recebimento dos dados - *receive***

Figura 3.18 Troca de Dados

Após o envio/recebimento cada processador calcula sua parte, como ilustrado na fig.3.19. E, então, são remetidos de volta ao processador *0* (fig. 3.18 (b)).

P0			P1			P2			P3		
1	9		17	25		33	41		49	57	
X			X			X			X		
1	2	19	3	4	151	5	6	411	7	8	799
9	10	99	11	12	487	13	14	1003	15	16	1647
17	18	179	19	20	823	21	22	1595	23	24	2495
25	26 =	259	27	28 =	1159	29	30 =	2187	31	32 =	3343
33	34	339	35	36	1495	37	38	2779	39	40	4191
41	42	419	43	44	1831	45	46	3371	47	48	5039
49	50	499	51	52	2167	53	54	3963	55	56	5887
57	58	579	59	60	2503	61	62	4555	63	64	6735

Figura 3.19 Cálculo local em cada processador

Por fim, o processador *0* efetua a soma dos resultados parciais, calculados em cada processador, gerando o vetor produto desejado. Isso é mostrado na fig.3.20.

19	+	151	+	411	+	799	=	1380
99	+	487	+	1003	+	1647	=	3236
179	+	823	+	1595	+	2495	=	5092
259	+	1159	+	2187	+	3343	=	6948
339	+	1495	+	2779	+	4191	=	8804
419	+	1831	+	3371	+	5039	=	10660
499	+	2167	+	3963	+	5887	=	12516
579	+	2503	+	4555	+	6735	=	14372

Figura 3.20 Cálculo vetor produto - soma dos valores parciais

3.6.3 Considerações sobre complexidade

O algoritmo trivial para multiplicação de matrizes, tendo a complexidade de tempo calculado em função da operação de multiplicação (e adição) é da ordem O (n^3).

Existem, entretanto, algoritmos recursivos para multiplicação de matrizes são mais eficientes, como o algoritmo de *Strassen* [STR69], que possui uma complexidade de O ($n^{\log_2 7}$) = O ($n^{2.81}$).

A complexidade de tempo do algoritmo de *Strassen*, quando executado em um computador com arquitetura com p processadores e com a memória global suficientemente rápida para satisfazer todos os requisitos desses p processadores, é O(n^3/p). Para computadores que se utilizem de troca de mensagens, a complexidade de tempo desse algoritmo é afetada pelo tempo de comunicação (troca de dados), que possui uma complexidade de comunicação da ordem de O (n^2 *(1-(1/p))*). Portanto, a complexidade de tempo nesse caso é de O ($n^2(1-(1/p))+n^3/p$).

A complexidade de tempo e de comunicação do *Algoritmo 3.6.4,* pode ser estimada, pela análise de cada instrução e assim, tem-se a complexidade total do algoritmo, como:

- o passo 3 é o único passo que requer O(n^2/p) operações aritméticas;
- os passos 1 e 2 transferem O(n^2/p) dados (números) da memória compartilhada para a memória local de cada processador;
- o passo 4 armazena n/p números de cada memória local na memória compartilhada.

Algoritmo 3.6.6 é executado em um tempo O($\log n$) usando um total de O(n^3) processadores. Pelo princípio de escalonamento, o algoritmo pode ser simulado através de p processadores sendo executado em um tempo O($n^3/p + \log n$).

Considere a adaptação desse algoritmo para o caso onde há n processadores disponíveis. Em particular, o tempo de execução correspondente deve ser O(n^2), no exame da complexidade de comunicação de *Algoritmo 3.6.6* relativo a um esquema particular de alocação de processadores. Um esquema direto consiste em alocar as operações incluídas em cada unidade de tempo aos processadores. Em particular, as n^3 operações simultâneas do passo *1* podem ser alocadas igualmente entre os n processadores como segue. Para cada $1 \le i \le n$, o processador P_i computa $C'(i, j, l) = A(i, l)B(l, j)$, onde $1 \le j, l \le n$; logo, P_i tem que ler a *i-ésima* linha de A e toda a matriz B da memória compartilhada. Um tráfico de O(n^2) números é criado entre a memória compartilhada e cada uma das memórias locais dos processadores.

A h-ésima iteração do laço no passo *2* requer $n^3/2^h$ operações simultâneas que podem ser alocadas como segue. A tarefa do processador P_i é atualizar os *valores* $C'(i, j, l)$, para todo o $1 \le j, l \le n$; logo, P_i pode ler todos valores necessários $C'(i, j, l)$ para todos os índices $1 \le j, l \le n$, e pode então executar as operações requeridas nesse

conjunto de valores. Novamente, $O(n^2)$ entradas são trocadas entre a memória compartilhada e a memória local de cada processador.

Finalmente, o passo 3 pode ser implementado facilmente com $O(n)$ comunicações, desde que o processador P_i tenha que armazenar a *i-ésima* linha da matriz produto na memória compartilhada, para $1 \leq i \leq n$. Então, têm-se um esquema de alocação de processadores que adapta o princípio do escalonamento com uma exigência de comunicação de $O(n^2)$.

O limite superior para o problema de multiplicação de matrizes conhecido é $O(n^{2.374})$, atribuído a *Pan* ([PAN78]).

3.7 Conclusões

Complexidade de um algoritmo é o esforço (quantidade de trabalho) de um algoritmo. As principais medidas de complexidade são tempo e espaço, relacionadas à velocidade e quantidade de memória, respectivamente. A complexidade depende da particular entrada: os principais critérios são pior caso e caso médio.

A complexidade é determinada com base em operações fundamentais e no tamanho da entrada. Ambos devem ser apropriados ao algoritmo ou ao problema. A complexidade experimental costuma depender de detalhes de implementação, variando de máquina a máquina. Análise matemática fornece a complexidade intrínseca do algoritmo.

A complexidade exata fornece muita informação, sendo, muitas vezes, difíceis suas determinação e análise. A complexidade assintótica dá o desempenho para entradas de tamanho grande.

As principais notações de comportamento assintótico dão limites superior (O), inferior (Ω) e exato (Θ), a menos de constantes multiplicativas. Algoritmos polinomiais têm complexidade $O(n^k)$; os com complexidade $\Omega(2^n)$ são exponenciais.

Observou-se que a computação do multiplicação de matrizes é um processo particularmente propício para o processamento paralelo síncrono. Entretanto, a análise realizada dos algoritmos sincronizados de produto de matrizes, levou ao entendimento de quais características específicas de arquiteturas para estruturas paralelas influenciam na determinação do melhor algoritmo paralelo.

A determinação do melhor algoritmo tem seu lado teórico e seu lado prático. No lado teórico, considera-se apenas a análise do algoritmo em função do número de operações fundamentais e do número ótimo de processadores necessários para essa resolução. Infelizmente, para comprovação prática dessa escolha, encontram-se outras dificuldades, como: tempo de comunicação, sincronização e troca de dados; o tamanho do problema pode vir a determinar diferentes algoritmos como ótimos em uma mesma máquina paralela, como por exemplo em uma máquina vetorial, onde o tamanho da entrada determinará se o tempo de montagem do *pipeline* (*startup*) irá ou não afetar o tempo total de processamento.

Percebe-se com esse curso que muitas questões ainda devem ser pesquisadas, para que se tenham repostas práticas que auxiliem na formação de um conhecimento básico de programação paralela, na criação de uma cultura paralela. Essas questões dizem respeito a importância do tempo de gerência (*overhead*) e seu impacto no tempo de processamento e na complexidade de um problema; impacto das contingências de memória no desempenho de programas paralelos; uso de algoritmos síncronos e assíncronos em diferentes ambientes; definição de linguagens de descrição de algoritmos paralelos capazes de descrever a exploração do paralelismo, de facilitar o

cálculo da complexidade de tempo e espaço; determinação do número ótimo de processadores, enfim. Esse curso apenas iniciou a discutir a importância da teoria, da análise de complexidade de algoritmos paralelos para o desenvolvimento de programas paralelos eficientes.

3.8 Agradecimentos

Agradecemos ao CNPq e a FAPERGS pelo suporte econômico aos projetos de pesquisa dos nossos Grupos, que viabilizaram o desenvolvimento desse trabalho e de muitos outros. E, por fim, aos bolsistas de Iniciação Científica e de Mestrado que nos auxiliaram no desenvolvimento desse trabalho, especialmente a Karina Kohl Silveira e o Delcino Picinin Junior.

3.9 Bibliografia

[AKL89] AKL, SELIM.G. **The design and analysis of parallel algorithms**. Englewood Cliffs: Prentice Hall, 1989. 401p.

[BAT68] BATCHER, K.E. Sorting networks and their applications. In: SPRING JOINT COMPUTER CONFERENCE. **Proceedings...** Atlantic City, 30 april. Reston: AFIPS Press, 1968. pp.307-314.

[BED 89] BERTSEKAS, DIMITRI.P; TSITSIKLIS, JOHN.N. **Parallel and distributed computations numerical methods**. Englewood Cliff: Prentice Hall, 1989. 715p.

[BER 74] BERNSTEIN, P.A.; TSICHRITZIS, D. C. **Operating Systems.** New York: Academic Press, 1974. 298p.

[BOV93] BOVET, D.; CRESCENZI, P. **Introduction to the Theory of Complexity**. Prentice Hall, 1993.

[CNA2001] CNAPAD. **Introdução ao MPI**. São Paulo, 2001.

[COL86] COLE, R. Parallel Mergsort. In: IEEE Focs, 27. **Proceedings...** 1986. p.511-516.

[COO71] COOK, S.A.The Complexity of Theorem-proving procedures. Proc. 3rd Ann. ACM Symp. on Theory of Computation, Association for Computing Machinery, **Proceedings...** New York: 1971. p.151-158.

[DIV92] DIVERIO, T. A; CLAUDIO, D. M.; NAVAUX, P. O. A. **Divisão e Conquista uma técnica para paralelização de algoritmos.** Porto Alegre: CPGCC da UFRGS, 1992. RP-177. 132p

[FLY 72] FLYNN, M. J. Some organizations and their effectiveness. **IEEE Transactions Computer**. New York,v.C-21,n.9, p.948-960, Sep, 1972.

[FOS95] FOSTER, IAN T. **Designing and Building Parallel Programs** concepts and tools for parallel software engineering. Reading: Addison-Wesley, 1995. 379p.

[GAR79] GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability: a guide to the theory of NP-completeness**. San Francisco: W. H. Freeman, 1979.

[HIR78] HIRSCHBERG, D.S. Fast Parallel Sorting Algorithm. **Communications ACM.** v.22, n.8, 1978.

[HWA 84] HWANG, KAI; BRIGGS FAYE ALAYE. **Computer Architecture and Parallel Processing.** New York: McGraw Hill, 1984. 846p.

[JAJ92] JAJA, JOSEPH. **An introduction to parallel algorithms.** Reading: Addison-Wesley, 1992. 566p.

[KRO86] KRÖNSJO,L. **Computational complexity of sequential and parallel algorithms**. Chichester: John Wiley, 1986.

[KUM94] KUMAR,V; GRAMA, A.; GUPTA, A.; KARYPIS, G. **Introduction to Parallel Computing - design and analysis of algorithms.** Redwood City: Benjamin/Cummings Publishing, 1994. 597p.

[MAN 89] MANBER U. **Introduction to Algorithms: A Creative Approach** Addison-Wesley, 1989.

[MUP75] MULLER, D.E.; PREPARATA, F.P. Bounds to complexities of networks for sorting and for switching. **Journal of the ACM.** v.22, n.2, p.195-201, 1975.

[PAC97] PACHECO, PETER S. **Parallel Programming with MPI.** San Francisco: Morgan Kaufman, 1997. 418p.

[PAN78] PAN, V. Y. Strassen's algorithm is not optimal. In: ANNUAL SYMPOSIUM ON FOUNDATION OF COMPUTER SCIENSE, 19. **Proceedings...** 1978.

[PIP79] PIPPNGER, NICHOLAS. On simultaneous resource bounds (preliminary version) In: IEEE Symposium on Foundations of Computer Science. **Proceedings...** Los Angeles: IEEE, 1979. p.307-311.

[PRE78] PREPARATA, F.P. New Parallel Sorting Schemes. **IEEE Transactions on Computers.** v.C-22, n.7. 1978.

[QUI87] QUINN, MICHAEL JAY. **Designing efficient algorithms for parallel computers.** New York: McGraw-Hill, 1987. 288p.

[STO78] STONE, H. S. Sorting on Star. **IEEE Transactions on Software Engineering ...** v.4. n.2. 1978.

[STR69] STRASSEN, V. Gaussian elimination is not optimal. **Numerische Matematik.** v.13. Berlin: Spring Verlag, 1969. pp.354-356.

[TAN76] TANENBAUM, ANDREW S. **Structured Computer Organization.** Englewood Cliffs, Prentice-Hall, 1976. 443p.

[TAN95] TANENBAUM, ANDREW S. **Distributed Operating Systems.** Upper Saddle River: Prentice-Hall, 1995. 614p.

[TER 90] TERADA,R. **Introdução à complexidade de algoritmos paralelos**. In: ESCOLA DE COMPUTAÇÃO, VII,1990, São Paulo, Curso. São Paulo, IME-USP, 1990. 104p.

[TOS2001] TOSCANI, L. V.; VELOSO, P.A.S. **Complexidade de Algoritmos - análise, projeto e métodos.** Porto Alegre: Sagra-Luzzatto/Instituto de Informática da UFRGS, 2001. 202p. {Série livros didáticos, número 13}.

[TRA73] TRAUB, J. F. **Complexity of sequential and parallel numerical algorithms.** New York: Academic Press, 1973. 300p.

[WIL99] WILKINSON, B.; ALLEN, M. **Parallel Programming Techniques and Applications using networked workstations and parallel computers**. Upper Saddle River: Prentice Hall, 1999. 431p.

[ZOM96] ZOMAYA, ALBERT Y. (Ed.). **Parallel and Distributed Computing Handbook.** New York: McGraw-Hill, 1996. 1199p.