**Algoritmos de Seleção
para Máquinas Paralelas
com Memória Distribuída**

Einar Luciano Gattoni Saukas

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO GRAU DE MESTRE
EM
MATEMÁTICA APLICADA

Área de Concentração: **Ciência da Computação**
Orientador: **Prof. Dr. Siang Wun Song**

São Paulo, 9 de fevereiro de 1998

Algoritmos de Seleção para Máquinas Paralelas com Memória Distribuída

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Einar Luciano Gattoni Saukas
e aprovada pela comissão julgadora.

São Paulo, 9 de fevereiro de 1998.

Banca examinadora:

- Prof. Dr. Siang Wun Song (orientador) - IME-USP
- Prof. Dr. José Augusto R. Soares - IME-USP
- Prof. Dr. Valmir C. Barbosa - COPPE-UFRJ

Agradecimentos

Aos professores e colegas do Instituto de Matemática e Estatística, por proporcionarem este longo período de produtiva e agradável convivência. Em especial aos Professores Doutores Siang Wun Song e Imre Simon, pela orientação, dedicação, apoio e extrema paciência.

À família e amigos(as), por lamentarem, em diversos eventos ao longo deste período, não contar com minha inestimável presença.

Ao Departamento de Ciência da Computação do Instituto de Matemática e Estatística e ao Laboratório de Sistemas Integráveis da Escola Politécnica da Universidade de São Paulo, por fornecerem os recursos para a realização das diversas experiências.

Resumo

O tema principal deste trabalho é o problema da seleção: determinar o k -ésimo menor elemento de uma sequência de n elementos. Para o modelo CGM (*Coarse Grained Multicomputer*) com p processadores e memória local de tamanho $O(n / p)$, apresentamos um novo algoritmo paralelo determinístico para o problema da seleção que requer $O(\log p)$ rodadas de comunicação. Além deste número pequeno de rodadas, o algoritmo também procura minimizar a quantidade total de informações transmitidas a cada rodada (apenas $O(p)$ exceto na última rodada).

O algoritmo básico é então adaptado para resolver o problema de q seleções simultâneas na mesma sequência de entrada, também em $O(\log p)$ rodadas de comunicação e assintoticamente mesmo tempo de computação local, caso $q=O(p)$. O algoritmo de seleção simultânea origina um algoritmo de ordenação de comunicação eficiente, com $O(\log p)$ rodadas de comunicação e um total de $O(p^2)$ informações transmitidas em cada rodada exceto na última.

Em complemento à análise teórica de complexidades, apresentamos as implementações destes algoritmos utilizando interfaces de comunicação padrão (PVM e MPI) e os resultados experimentais obtidos em duas máquinas paralelas diferentes. Estes resultados mostram ganhos de desempenho (“*speedups*”) praticamente lineares, indicando a eficiência e escalabilidade dos algoritmos propostos.

Abstract

The main subject of this work is the selection problem of determining the k th smallest element of a sequence of n elements. Under the CGM (*Coarse Grained Multicomputer*) model with p processors and $O(n / p)$ local memory, we present a new deterministic parallel algorithm for the selection problem that requires $O(\log p)$ communication rounds. Besides this low number of rounds, the algorithm also attempts to minimize the total amount of data transmitted in each round (only $O(p)$ except in the last round).

The basic algorithm is then extended to solve the problem of q simultaneous selections using the same input sequence, also in $O(\log p)$ communication rounds and asymptotically same local computing time, if $q=O(p)$. The simultaneous selection algorithm gives rise to a communication efficient sorting algorithm, with $O(\log p)$ communication rounds and a total of $O(p^2)$ data transmitted in each round except in the last one.

In addition to the analysis of theoretical complexities, we present the implementations of these algorithms using standard communication interfaces (PVM and MPI) and the experimental results obtained on two different parallel machines. These results show almost linear speedups, indicating the efficiency and scalability of the proposed algorithms.

Índice

1. Introdução	1
2. Conceitos Básicos	4
3. Modelos de Computação Paralela para Memória Distribuída.....	6
3.1 Modelo BSP	6
3.2 Modelo LogP.....	7
3.3 Modelo CGM	7
3.4 Outros Modelos.....	8
4. O Problema da Seleção	9
5. Descrição de Algoritmos	11
5.1 Algoritmos Sequenciais.....	11
5.2 Algoritmos Paralelos para Memória Compartilhada	12
5.3 Algoritmos Paralelos para Memória Distribuída.....	16
6. Algoritmos Propostos	18
6.1 Algoritmo básico de seleção do k -ésimo elemento.....	18
6.2 Algoritmo de seleção simultânea de q “ k -ésimos” elementos	24
6.3 Algoritmo de ordenação baseado em seleção	29
7. Implementação em PowerXplorer (PVM).....	33
7.1 Algoritmo básico de seleção	35
7.1.1 <i>Descrição dos casos de teste</i>	35
7.1.2 <i>Análise do parâmetro c</i>	36
7.1.3 <i>Análise de tempos parciais</i>	38
7.1.4 <i>Determinação do parâmetro c</i>	39
7.1.5 <i>Análise de desempenho</i>	40
7.1.6 <i>Conclusões</i>	41
7.2 Algoritmo de seleção simultânea	42
7.2.1 <i>Descrição dos casos de teste</i>	42
7.2.2 <i>Análise de desempenho em função do número de elementos n</i>	44
7.2.3 <i>Análise de desempenho em função do número de problemas q</i>	45
7.2.4 <i>Conclusões</i>	46
7.3 Algoritmo de ordenação	46
7.3.1 <i>Descrição dos casos de teste</i>	46
7.3.2 <i>Análise de desempenho</i>	47

7.3.3 <i>Análise de tempos parciais</i>	48
7.3.4 <i>Conclusões</i>	49
8. Implementação em Meiko CS-2 (MPI)	50
8.1 Algoritmo básico de seleção	51
8.1.1 <i>Análise do parâmetro c</i>	51
8.1.2 <i>Análise de tempos parciais</i>	52
8.1.3 <i>Determinação do parâmetro c</i>	52
8.1.4 <i>Análise de desempenho</i>	53
8.1.5 <i>Conclusões</i>	53
8.2 Algoritmo de seleção simultânea	54
8.2.1 <i>Análise de desempenho em função do número de elementos n</i>	54
8.2.2 <i>Análise de desempenho em função do número de problemas q</i>	55
8.2.3 <i>Conclusões</i>	55
8.3 Algoritmo de ordenação	55
8.3.1 <i>Análise de desempenho</i>	55
8.3.2 <i>Análise de tempos parciais</i>	56
8.3.3 <i>Conclusões</i>	57
9. Considerações Finais	58
10. Referências	60
A. Listagens dos Programas Implementados	66
A.1 “alg_seq.h”: Protótipos dos algoritmos sequenciais	67
A.2 “alg_seq.c”: Algoritmos sequenciais básicos	68
A.3 “alg_par.h”: Protótipos dos algoritmos paralelos	78
A.4 “alg_par.c”: Algoritmos paralelos principais	79
A.5 “alg_com.h”: Protótipos das funções de comunicação	91
A.6 “alg_pvm.c”: Funções de comunicação em PVM	93
A.7 “alg_mpi.c”: Funções de comunicação em MPI	106
A.8 “alg_tst1.c”: Exemplo de utilização do algoritmo 1	116
A.9 “alg_tst2.c”: Exemplo de utilização do algoritmo 2	118
A.10 “alg_tst3.c”: Exemplo de utilização do algoritmo 3	120
B. Gráficos de Desempenho	122

Diagramas

1. Arquiteturas de máquinas paralelas.....	6
2. Particionamento pela mediana das medianas.....	12
3. Estratégia de particionamento	19
4. Determinação da mediana ponderada	20
5. Funcionamento do algoritmo básico de seleção.....	22
6. Funcionamento do algoritmo de seleção simultânea	27
7. Funcionamento do algoritmo de ordenação baseado em seleção.....	31

Tabelas

1. Algoritmos de seleção sequenciais.....	12
2. Algoritmos de seleção paralelos para memória compartilhada.....	15
3. Algoritmos de seleção paralelos para memória distribuída.....	17

1. Introdução

O modelo mais utilizado para o estudo de algoritmos paralelos é o PRAM (*Parallel Random Access Machine*), que representa máquinas paralelas como um conjunto de processadores que acessam diretamente uma única memória compartilhada. Entretanto, os computadores paralelos atuais, em sua maioria, possuem memórias distribuídas, sendo constituídos por um conjunto de processadores “estado-da-arte” (por exemplo SPARC), cada um com uma memória local de alta capacidade, que se comunicam entre si através de uma determinada rede de interconexão (como grade ou hipercubo). Nestes computadores, a velocidade de comunicação (e portanto de acesso à memória de outro processador) é muito menor do que a velocidade de processamento local (com acesso somente à memória do próprio processador), o que faz com que em geral implementações de algoritmos PRAM não apresentem um bom resultado prático.

Nesta última década, novos modelos de computação paralela têm sido propostos com o objetivo de representar de forma mais realista o comportamento de máquinas com memória distribuída em relação à comunicação. Estes modelos possuem em comum a idéia de que os algoritmos devem favorecer computações locais e minimizar operações globais. Em particular, o modelo CGM (*Coarse Grained Multicomputer*) proposto por F. Dehne, A. Fabri e A. Rau-Chaplin considera algoritmos paralelos que executam alternadamente rodadas somente de computação local ou somente de comunicação global, sugerindo a minimização do número destas rodadas como uma forma de obter algoritmos paralelos (para memória distribuída) eficientes.

Estas idéias são aplicadas neste trabalho ao problema da seleção, resultando em um novo algoritmo determinístico para máquinas paralelas com memória distribuída (no modelo CGM), desenvolvido a partir de um algoritmo PRAM anterior. Este algoritmo requer apenas $O(\log p)$ rodadas de comunicação, onde p é o número de processadores, portanto um número pequeno de rodadas independente do tamanho do problema. Outro aspecto importante deste algoritmo é a reduzida quantidade de dados transmitidos, apenas um total de $O(p)$ informações a cada rodada de comunicação exceto na última, uma pequena quantidade também independente do tamanho do problema. Embora o modelo CGM não considere este último aspecto, o esforço em também reduzir a quantidade de informações transmitidas mostrou-se crucial para o bom desempenho do algoritmo na prática.

Este algoritmo foi implementado tanto para a interface de comunicação padrão PVM como para MPI. Estas duas implementações foram empregadas na realização de testes de desempenho

nas máquinas paralelas de memória distribuída “Parsytec PowerXplorer” e “Meiko CS-2”, respectivamente. Os resultados experimentais obtidos mostram que o “*speedup*”¹ aumenta quase linearmente, em relação ao aumento do número de processadores, para problemas de tamanho suficientemente grande. Estes resultados são muito significativos e comprovam a eficiência e escalabilidade do algoritmo proposto, principalmente tendo em vista que a primeira destas duas máquinas paralelas apresenta uma considerável latência e pequena largura de banda de comunicação, portanto com tempos de comunicação significativamente altos em relação ao processamento local. Este novo algoritmo tem-se comparado favoravelmente a todos os demais algoritmos de seleção determinísticos para máquinas paralelas com memória distribuída posteriormente pesquisados e, após um extenso trabalho de pesquisa, acreditamos tratar-se do melhor algoritmo de seleção deste tipo.

O algoritmo básico de seleção é então generalizado em um segundo algoritmo para resolver simultaneamente q problemas de seleção em uma mesma sequência de elementos. Para $q \leq p$, o algoritmo requer o mesmo número de rodadas de comunicação do algoritmo básico de seleção, com a transmissão de no máximo q vezes mais informações por rodada. Além disso, para $q=O(p)$, requer assintoticamente o mesmo tempo de computação local em relação à resolução do problema de seleção de um único k -ésimo elemento. Novamente os resultados experimentais comprovam sua eficiência e escalabilidade.

A seguir apresentamos um algoritmo determinístico de ordenação baseado no algoritmo de seleção simultânea. As soluções de um problema de seleção simultânea são utilizadas para particionar, em cada processador, a sequência de dados de entrada em sub-sequências, que são então redistribuídas entre os processadores para serem ordenadas localmente. Este algoritmo requer $O(\log p)$ rodadas de comunicação e um total de apenas $O(p^2)$ dados transmitidos em cada rodada exceto na última. Mais uma vez, foram obtidos experimentalmente *speedups* praticamente lineares.

O objetivo deste trabalho é portanto o estudo do problema da seleção, enfocando principalmente estes novos algoritmos propostos.

A apresentação deste trabalho supõe que o leitor já possua algum conhecimento inicial prévio da área de algoritmos paralelos. O capítulo 2 cita os principais conceitos básicos requeridos.

No capítulo 3 são apresentados os principais modelos de computação paralela para memória distribuída. Estes modelos são a base para o desenvolvimento tanto dos novos algoritmos propostos como de diversos algoritmos anteriores.

O capítulo 4 contém uma breve descrição formal do problema da seleção. No capítulo 5 são descritos os princípios de funcionamento dos diversos algoritmos disponíveis na literatura para este

¹ Uma tradução aproximada para o termo “*speedup*” seria “ganho de desempenho”, mas na ausência de uma tradução mais precisa optou-se por utilizar o termo original em inglês no decorrer deste trabalho.

problema, permitindo acompanhar a evolução das soluções ao longo do tempo e fornecendo uma base para comparação com os novos algoritmos propostos.

O capítulo 6 apresenta a descrição e análise do novo algoritmo básico de seleção aqui proposto, assim como sua generalização para o problema da seleção simultânea e sua aplicação ao problema da ordenação.

O capítulo 7 descreve os resultados experimentais obtidos com a execução destes algoritmos na máquina paralela “*Parsytec PowerXplorer*”, a partir de implementações dos algoritmos para a interface de comunicação padrão PVM. Uma vez que estes testes têm como objetivo estudar o comportamento do algoritmo e não da máquina paralela, efetua-se aqui uma análise de resultados eminentemente qualitativa, com ênfase na variação dos tempos de execução obtidos e não em tempos individuais. Estes testes são repetidos no capítulo 8, desta vez para a máquina paralela “*Meiko CS-2*” com as implementações para a interface de comunicação padrão MPI.

Finalmente, as considerações finais deste trabalho são apresentadas no capítulo 9.

O trabalho encerra-se com uma lista de referências pesquisadas no capítulo 10, as listagens dos programas implementados no apêndice A e as apresentações gráficas dos resultados experimentais no apêndice B.

2. Conceitos Básicos

O modelo PRAM (*Parallel Random Access Machine*) é o principal modelo de computação paralela para memória compartilhada. Este modelo é composto por um conjunto de processadores, cada um com sua própria memória local (portanto capazes de executar instruções diferentes), operando de forma sincronizada e com acesso direto a uma única memória compartilhada global. Há três variações básicas deste modelo, dependendo da capacidade de acesso simultâneo de diversos processadores a uma mesma posição de memória:

- EREW (*Exclusive Read Exclusive Write*) PRAM não permite acesso simultâneo a uma mesma posição de memória.
- CREW (*Concurrent Read Exclusive Write*) PRAM permite acesso simultâneo apenas para instruções de leitura.
- CRCW (*Concurrent Read Concurrent Write*) PRAM permite acesso simultâneo tanto para leitura como para escrita, neste caso a operação de escrita pode ser do tipo arbitrário (em caso de conflito, um processador qualquer realiza a escrita), prioritário (o processador de menor índice realiza a escrita) ou comum (a escrita concorrente é permitida apenas quando todos estes processadores estão tentando escrever o mesmo valor).

O trabalho de um algoritmo paralelo é definido como o número total de operações realizadas por todos os processadores. Diz-se que o algoritmo é ótimo quando este trabalho corresponde à complexidade sequencial do problema. Em geral, os algoritmos paralelos para o modelo PRAM são projetados para utilização de um grande número de processadores, procurando-se reduzir o tempo de execução mas empregando um trabalho próximo do valor ótimo.

De acordo com o Teorema de Brent, qualquer algoritmo paralelo, projetado para um determinado número de processadores, pode ser convertido em um novo algoritmo para um número m vezes menor de processadores e tempo de execução m vezes maior. Esta conversão é obtida fazendo-se com que, a cada rodada, cada processador do novo algoritmo simule as operações de m processadores do algoritmo original. Deve-se entretanto observar que a adaptação de um algoritmo no sentido inverso (aumentando o número de processadores) não significa necessariamente uma redução correspondente no tempo de execução do algoritmo.

Portanto, um algoritmo paralelo ótimo do modelo PRAM, projetado para diversos processadores, pode ser facilmente adaptado para um novo algoritmo paralelo que utilize um número menor de processadores e continue empregando um trabalho ótimo.

O ganho no desempenho de um algoritmo paralelo pode ser medido pelo *speedup*, definido como a razão entre o tempo de execução do melhor algoritmo sequencial para o problema e o tempo obtido para o algoritmo paralelo. Evidentemente o valor do *speedup* é limitado pelo número de processadores utilizados, devendo-se procurar obter um *speedup* o mais próximo possível deste número. Mas na prática existem diversos fatores responsáveis por provocar ineficiências significativas nos algoritmos paralelos, destacando-se os atrasos introduzidos pela comunicação entre processadores, os problemas de concorrência insuficiente da computação e a necessidade de operações adicionais para a sincronização dos processadores e controle do sistema.

Estes conceitos básicos brevemente abordados acima são considerados pré-requisitos para a leitura deste trabalho. Maiores detalhes a respeito destes tópicos podem ser encontrados em [Ja-92]. Uma discussão mais abrangente acerca do conceito de *speedup* é realizada em [NA-91].

3. Modelos de Computação Paralela para Memória Distribuída

O modelo PRAM, devido à sua simplicidade, tem sido o mais utilizado para a representação e análise de algoritmos paralelos. Mas como os computadores paralelos atuais, em geral, possuem memória distribuída (e não compartilhada), os algoritmos desenvolvidos segundo este modelo não são muito adequados para utilização prática. Mesmo em relação a computadores paralelos com memória compartilhada, estes algoritmos ainda apresentam dificuldades de implementação, pois o modelo PRAM não considera os problemas de sincronização entre os processadores e de contenção no acesso simultâneo a diferentes posições de um mesmo módulo de memória (o modelo EREW PRAM considera apenas o problema de contenção em relação a uma mesma posição de memória).

Recentemente diversos modelos têm sido propostos para representar também as questões de comunicação e sincronismo entre processadores, visando assim a elaboração de algoritmos de maior viabilidade prática. Alguns dos principais modelos são apresentados a seguir.

3.1 Modelo BSP

O modelo BSP (*Bulk-Synchronous Parallel*), proposto por Leslie Valiant em 1990, tem sido o modelo paralelo para memória distribuída de maior aceitação ([Va-90], [GV-92], [Va-92], [Va-

93], [GV-94], [Mc-94a], [Mc-94b], [Mc-94c], [CFSV-94], [Mc-95], [CFSV-95]). Segundo este modelo, um algoritmo é composto por uma sequência de super-passos. Em cada super-passo, cada processador realiza seu processamento local e pode enviar e receber no máximo h mensagens (comunicação denominada por h -relação), sendo que as mensagens enviadas são apenas recebidas no início do próximo super-passo. Ao final de cada super-passo realiza-se uma sincronização por barreira. Os algoritmos devem ser projetados em função dos parâmetros p (número de processadores), g (relação entre capacidade de processamento local e capacidade de comunicação) e L (intervalo de tempo mínimo entre sincronizações, chamado de periodicidade).

Valiant observa que algoritmos PRAM podem ser simulados otimamente no modelo BSP, desde que o algoritmo PRAM seja projetado para um determinado número de processadores virtuais v suficientemente maior que p , de forma que cada processador real do computador BSP simule diversos processadores virtuais do algoritmo PRAM (esta relação mínima entre v e p é denominada “*slackness*”). No entanto, as constantes envolvidas nestas simulações são em geral muito grandes (devido por exemplo ao fato de a maioria das máquinas paralelas apresentar um parâmetro g elevado), portanto permanece a necessidade de projetar algoritmos considerando especificamente uma memória distribuída.

3.2 Modelo LogP

O modelo LogP é proposto por diversos autores em 1993 [CKPSSSSE-93] com o objetivo de representar de forma mais detalhada (mas portanto mais complexa) as características da transmissão de mensagens. Os principais parâmetros utilizados neste modelo são latência L (tempo entre transmissão e recebimento da mensagem), *overhead* o (tempo gasto pelo processador exclusivamente na atividade de transmissão ou recebimento), *gap* g (mínimo intervalo de tempo entre transmissão ou recebimento de mensagens consecutivas em um processador) e número de processadores p .

3.3 Modelo CGM

O modelo CGM(n, p) (*Coarse Grained Multicomputer*), proposto por Frank Dehne, Andreas Fabri e Andrew Rau-Chaplin em 1993 ([DFR-93], [DKF-94], [DDDFK-95]), baseia-se apenas nos parâmetros n e p , onde n é o tamanho do problema a ser resolvido e p é o número de processadores. O objetivo deste modelo é fornecer uma representação simples porém suficientemente próxima da arquitetura das máquinas paralelas existentes, de forma que as complexidades teóricas dos algoritmos coincidam com os desempenhos de suas implementações. O termo “*coarse grained*” (granularidade grossa) deve-se ao fato de este modelo supor que o tamanho do problema é consideravelmente maior do que o número de processadores, com cada processador armazenando portanto uma parcela significativa do problema, pois esta situação corresponde, em geral, à real utilização dos algoritmos na prática.

O modelo considera um conjunto de p processadores, cada um com memória local considerável (tipicamente de tamanho $O(n/p)$ com $n/p > p$), interligados através de uma rede de interconexão arbitrária (ou até mesmo através de uma área de memória compartilhada). Os algoritmos devem realizar um número reduzido de rodadas alternadas de computação local e comunicação global, sendo a comunicação realizada através de operações globais de redistribuição das informações contidas nas memórias locais.

As rodadas do modelo CGM equivalem aos super-passos do modelo BSP, considerando-se a realização de (n/p) -relações com $L=(n/p)g$, ou seja, cada processador pode trocar um total de $O(n/p)$ informações com outros processadores a cada rodada de comunicação.

De acordo com este modelo, o custo de comunicação de um algoritmo paralelo é medido apenas pelo número de rodadas de comunicação. Considera-se que o esforço em minimizar este número de rodadas seja suficiente para garantir um bom desempenho do algoritmo, não sendo considerada a quantidade de informações transmitidas a cada rodada de comunicação. Assim, a estratégia básica para a resolução de um problema é dividi-lo em uma série de sub-problemas, realizando-se alternadamente execuções locais de algoritmos sequenciais ótimos sobre cada sub-problema e operações globais de comunicação para permutar os sub-problemas entre os processadores. O maior desafio para a elaboração de um algoritmo é identificar uma forma de divisão do problema que necessite de um número reduzido destas operações.

3.4 Outros Modelos

Existem diversos outros modelos paralelos para memória distribuída, entre eles o BSP* (uma extensão do modelo BSP que também considera o tamanho de cada mensagem [Go-97]), o BDM (*Block Distributed Memory*, que parametriza o tempo de computação local, o número de operações de comunicação e a quantidade de informações acessadas por estas operações [BJ-96]) e as diversas extensões do modelo PRAM (ao qual acrescentam-se algumas restrições, como um intervalo mínimo de tempo para a leitura de informações gravadas por outro processador [CKPSSSE-93]). Em geral, estes modelos procuram representar as características de comunicação de forma independente de estruturas de interconexão, o que permite a elaboração de algoritmos adequados a uma grande variedade de arquiteturas paralelas.

Muitos autores utilizam-se ainda de modelos específicos de interconexão, que consideram apenas mensagens entre processadores diretamente conectados fisicamente. Este tipo de modelo tem sido utilizado principalmente para obter algoritmos com ganhos marginais (não assintóticos) de desempenho, às custas de portabilidade, ou para analisar o problema para um número de processadores $O(n)$.

4. O Problema da Seleção

O problema da seleção consiste em determinar o k -ésimo menor elemento de uma sequência (não ordenada) de tamanho n . Constituem-se em casos especiais deste problema a determinação do elemento mínimo ($k = 1$), do elemento máximo ($k = n$) e da mediana ($k = \lceil n/2 \rceil$) de uma sequência.

Trata-se de um problema clássico da computação, cuja análise do ponto de vista algorítmico inicia-se na década de 1930 (segundo [Kn-73]) com uma série de artigos de diversos autores, nos quais estabelece-se o número mínimo de partidas (comparações) necessário para selecionar corretamente não apenas o primeiro, mas também o segundo melhor jogador em uma competição esportiva.

Formalmente, o problema da seleção é definido em [Ja-92] da seguinte forma:

Dados um conjunto de elementos (distintos) $A = \{a_1, a_2, \dots, a_n\}$ e um inteiro k , com $1 \leq k \leq n$, determinar o elemento a_i de A tal que $\text{rank}(a_i; A) = k$.

Por sua vez, rank é definido como:

Seja $X = \{x_1, x_2, \dots, x_l\}$ uma sequência cujos elementos pertencem ao conjunto S . Seja $x \in S$. O rank de x em X , denotado por $\text{rank}(x; X)$, é o número de elementos de X que são menores ou iguais a x .

A definição fornecida acima é adequada somente para sequências com elementos distintos, uma restrição adotada (ao menos na formalização do problema) pelas referências consultadas. No caso de sequências com elementos não necessariamente distintos, a aplicação direta desta definição implicaria que, por exemplo, o problema de seleção da mediana da sequência $A = \{5, 6, 7, 7, 8\}$ não teria solução, pois esta sequência não possui nenhum elemento com rank igual a 3 (note que $\text{rank}(7; A) = 4$).

Para contornar esta restrição, o problema da seleção pode ser mais genericamente definido desta forma:

Dados uma sequência de elementos $A = \{a_1, a_2, \dots, a_n\}$ e um inteiro k , com $1 \leq k \leq n$, determinar um elemento a_i de A tal que $\text{rank}'(a_i; A) < k \leq \text{rank}(a_i; A)$.

Esta nova definição do problema utiliza o mesmo conceito anterior de *rank* e considera *rank'* como:

*Seja $X = \{x_1, x_2, \dots, x_l\}$ uma sequência cujos elementos pertencem ao conjunto S . Seja $x \in S$. O *rank'* de x em X , denotado por $\text{rank}'(x: X)$, é o número de elementos de X que são estritamente menores que x .*

Observe que esta definição corresponde à idéia de considerar a seleção como o problema de determinar qual seria o elemento índice k na sequência A após a ordenação de seus elementos (independentemente desta sequência ser ou não composta somente de elementos distintos).

Alguns dos algoritmos publicados para este problema podem ser aplicados sem alterações a sequências com elementos não necessariamente distintos. No caso dos demais algoritmos, esta restrição pode ser contornada utilizando-se o índice de cada elemento na sequência (ou o seu próprio endereço físico de armazenamento inicial na memória) como um critério “artificial” de desempate na comparação entre estes elementos.

Os novos algoritmos que são propostos neste trabalho já consideram sequências com elementos não necessariamente distintos, sem utilizar este “critério de desempate”.

5. Descrição de Algoritmos

A seguir são descritos os principais algoritmos (disponíveis na literatura) para o problema da seleção. Em geral, os algoritmos paralelos para memória distribuída consistem em adaptações de algoritmos paralelos para memória compartilhada, que por sua vez foram elaborados a partir de algoritmos sequenciais, o que motivou a pesquisa destes algoritmos.

5.1 Algoritmos Sequenciais

O problema da seleção pode ser resolvido sequencialmente em tempo linear $O(n)$ através da estratégia de particionamentos sucessivos. Note-se que $\Omega(n)$ é um limite inferior trivial para este problema.

O algoritmo *FIND*, proposto por Hoare em 1961 [Ho-61], resolve este problema em tempo linear no caso médio. A solução encontrada consiste em selecionar aleatoriamente um elemento da sequência e particionar esta sequência com base neste “separador” (utilizando o mesmo método de particionamento do algoritmo *QUICKSORT*, do mesmo autor [Ho-62]), de forma a obter uma partição contendo apenas elementos menores ou iguais ao separador e outra contendo apenas elementos maiores ou iguais. Elimina-se então uma das partições a partir da comparação do valor k com o tamanho da partição de elementos menores: caso k seja menor, o algoritmo é reaplicado recursivamente apenas aos elementos desta partição, caso contrário, subtrai-se de k o tamanho desta partição e reaplica-se recursivamente o algoritmo apenas aos elementos da outra partição.

O primeiro algoritmo sequencial em tempo linear no pior caso é desenvolvido somente em 1973 por Blum, Floyd, Pratt, Rivest e Tarjan [BFPRT-73]. Este algoritmo, denominado *PICK*, divide inicialmente a sequência de entrada de n elementos em grupos de tamanho constante $c \geq 5$, calcula a mediana de cada um destes n/c grupos, determina a mediana das n/c medianas (empregando recursivamente o próprio algoritmo) e utiliza esta mediana das medianas como separador para o particionamento da sequência. Como no algoritmo anterior, particionamentos sucessivos são empregados para reduzir o tamanho do problema, mas a utilização da mediana das medianas garante que a cada particionamento sejam eliminados pelo menos $1/4$ dos elementos analisados.

Floyd e Rivest propõem também em 1975 ([FR-75a], [FR-75b]) o algoritmo probabilístico (não determinístico) *SELECT*, assintoticamente equivalente ao algoritmo *FIND*, mas exigindo um número menor de comparações. Este algoritmo realiza uma amostragem aleatória da sequência de n elementos e determina, com base nesta amostra, dois separadores a serem utilizados na divisão da sequência em 3 partições. A partir da comparação de k com o tamanho das partições obtidas, mantém-se apenas uma das partições e aplica-se o algoritmo recursivamente. Com alta probabilidade, a partição contida entre os dois separadores é mantida e possui um tamanho significativamente menor do que o número total de elementos analisados, o que garante um bom desempenho do algoritmo.

Algoritmo	Tipo	Tempo
Ordenação	Determinístico	$O(n \log n)$
FIND: Hoare (1961)	Probabilístico	$O(n)$
PICK: Blum, Floyd, Pratt, Rivest, Tarjan (1973)	Determinístico	$O(n)$
SELECT: Floyd, Rivest (1975)	Probabilístico	$O(n)$

Tabela 1: Algoritmos de seleção sequenciais

5.2 Algoritmos Paralelos para Memória Compartilhada

Em 1975, Valiant propõe o modelo PCT (*Parallel Comparison Tree*) para estudar algoritmos paralelos baseados em comparações [Va-75]. Este modelo considera apenas a realização de p comparações (simultâneas) a cada passo do algoritmo, ignorando o custo de analisar os resultados

destas comparações e decidir quais comparações devem ser realizadas no passo seguinte. Com este modelo, Valiant demonstra que a seleção do elemento máximo requer $\Omega(n/p + \log \log p)$ passos de comparação (para $p \leq n$), sendo este portanto um limite inferior de tempo para resolver o problema genérico da seleção utilizando $p \leq n$ processadores.

Em 1981, Shiloach e Vishkin ([SV-81]) apresentam um algoritmo de seleção do elemento máximo, desenvolvido a partir do trabalho de Valiant, desta vez para o modelo CRCW PRAM, portanto abordando também os diversos problemas de implementação (como a alocação de processadores a cada passo) e não apenas as operações de comparação. Este algoritmo seleciona o elemento máximo em tempo $O(n/p + \log \log p)$ para $p \leq n$, coincidindo portanto com o limite inferior de tempo estabelecido por Valiant para este problema. Observa-se que o algoritmo é ótimo (ou seja, apresenta um total de $O(n)$ operações, correspondendo ao caso sequencial) para $p \leq n / \log \log n$.

Quanto ao problema mais geral da seleção do k -ésimo menor elemento, este pode ser resolvido no modelo PRAM em tempo $O(\log n)$ simplesmente ordenando-se a sequência de n elementos, para tanto empregando-se por exemplo o algoritmo paralelo *MERGESORT* de Cole [Co-88a]. Mas esta solução requer um total de $O(n \log n)$ operações, sendo portanto não ótima.

A partir do algoritmo sequencial *PICK*, Vishkin desenvolve em 1983 um algoritmo paralelo ótimo (total de $O(n)$ operações) para EREW PRAM empregando a técnica de aceleração em cascata ([CV-86], [Ja-92]). A cada passo, este algoritmo divide a sequência de entrada em grupos de $\log n$ elementos, calcula a mediana de cada grupo, determina a mediana destas medianas (através da ordenação da sequência de $n/\log n$ medianas), determina o número de elementos menores, iguais e maiores do que a mediana das medianas e compara estes números com o valor de k , o que permite eliminar pelo menos $1/4$ dos elementos analisados (ou interromper a execução do algoritmo caso k já corresponda à mediana das medianas). Este processo é repetido até que o número de elementos restantes seja reduzido a $n/\log n$, o que permite que estes elementos sejam então ordenados em $O(n)$ operações. Como são necessárias $O(\log \log n)$ passos e cada um gasta tempo $O(\log n)$, este algoritmo requer um tempo total de $O(\log n \cdot \log \log n)$.

Em 1985, Cole e Yap [CY-85] apresentam um algoritmo paralelo (determinístico), para o modelo PCT com n processadores, baseado no algoritmo sequencial probabilístico *SELECT*. Seu funcionamento consiste em determinar, em tempo $O(\log \log n)$, dois separadores para o particionamento da sequência de n elementos, de forma que o k -ésimo elemento pertença sempre à partição contida entre os dois separadores. Garante-se também que, após $O(\log \log n)$ passos, restem menos de $\sqrt{n}$ elementos, que podem então ser ordenados pelos n processadores (comparando-se todos os elementos dois a dois) em tempo $O(1)$. Portanto o tempo total é de $O((\log \log n)^2)$ e o trabalho $O(n (\log \log n)^2)$.

No ano seguinte, este algoritmo é aperfeiçoado por Ajtai, Komlós, Steiger e Szemerédi ([AKSS-86], [AKSS-89]), que o alteram de forma que os primeiros $O(1)$ passos sejam executados

em tempo $O(\log \log n)$ (reduzindo para $n/(\log n)^3$ o número de elementos) e os $O(\log \log n)$ passos seguintes possam ser executados em tempo $O(1)$. Obtém-se assim um algoritmo para o modelo PCT de tempo $O(\log \log n)$ e trabalho $O(n \log \log n)$, que coincide com o limite inferior de tempo estabelecido por Valiant para n processadores.

Em 1988, Cole adapta estes algoritmos para o modelo PRAM, obtendo um algoritmo paralelo ótimo [Co-88b]. A cada passo do algoritmo, a determinação de dois separadores (para particionar a sequência de elementos) é feita em tempo $O(\log n)$, e após $O(\log^* n)$ passos o número de elementos é reduzido a $n/\log^2 n$, utilizando-se então o *MERGESORT* para ordenar estes elementos. O algoritmo requer tempo $O(\log n \cdot \log^* n)$ no modelo EREW PRAM, podendo ainda ser modificado para tempo $O(\log n \cdot \log^* n / \log \log n)$ no modelo CRCW PRAM, em ambos os casos executando apenas $O(n)$ operações.

Novamente para o modelo PCT, Azar e Pippenger apresentam em 1990 ([AP-90]) um novo algoritmo paralelo determinístico para o problema específico de seleção da mediana, observando entretanto que qualquer problema genérico de seleção pode ser reduzido à seleção da mediana de uma nova sequência com até o dobro de elementos simplesmente acrescentando-se elementos $-\infty$ ou $+\infty$. O algoritmo utiliza o conceito de “mediana aproximada” (um elemento qualquer de *rank* entre $n/4$ e $3n/4$) para determinar dois separadores, particionando a sequência de n elementos de forma que a mediana pertença à partição contida entre os dois separadores e esta partição tenha no máximo $\frac{n}{32\sqrt{1+6p/n}}$ elementos. Este procedimento é repetido (acrescentando-se novos elementos $-\infty$ ou $+\infty$ se necessário) até reduzir o número de elementos restantes para $\sqrt{p}$, que são então ordenados em tempo constante. O tempo de execução do algoritmo é $O\left(\frac{n}{p} + \log\left(\frac{\log n}{\log(1+p/n)}\right)\right)$ para $1 \leq p \leq \binom{n}{2}$, sendo que os autores também demonstram ser este o limite inferior de tempo para o problema da mediana no modelo PCT. Note-se que, no caso de $O(n/\log \log n)$ processadores, o algoritmo requer tempo $O(\log \log n)$ e trabalho $O(n)$, portanto igualando o tempo estabelecido anteriormente pelo algoritmo de Ajtai, Komlós, Steiger e Szemerédi, mas agora com trabalho ótimo.

Mais recentemente, Chaudhuri, Hagerup e Raman desenvolvem em 1993 [CHR-93] um algoritmo ótimo de tempo $O(\log n / \log \log n)$ para o modelo CRCW PRAM a partir do conceito de λ -seleção aproximada (que consiste em selecionar um elemento de *rank* entre $k - \lambda \cdot n$ e $k + \lambda \cdot n$, o que requer um trabalho consideravelmente menor do que a seleção exata do k -ésimo elemento). Utiliza-se a seleção aproximada para obter uma amostra de tamanho inferior a $\sqrt{n}$ e determinar dois separadores a partir desta amostra, reduzindo-se para $O(n / 2^{\sqrt{\log n}})$ o número de elementos da sequência de entrada e então aplicando-se o algoritmo *MERGESORT* aos elementos restantes.

Finalmente, Dietz e Raman apresentam em 1994 [DR-94] outro algoritmo ótimo que resolve o problema da seleção no modelo CRCW PRAM em tempo $O(\log \log n + \log k / \log \log n)$. Para $k \leq n^{1/16}$, diversos outros algoritmos são utilizados para obter entre k e k^+ dos menores elementos em tempo $O(\log \log n)$ e comparar estes elementos entre si em tempo $O(1)$ (pois $k^+ \leq \sqrt{n}$), utilizando-se então soma prefixa para determinar o *rank* de cada um destes elementos em tempo $O(\log k / \log \log n)$. Para $k > n^{1/16}$, utiliza-se o algoritmo anterior de Chaudhuri, Hagerup e Raman. O resultado obtido coincide com o limite inferior de tempo, demonstrado pelos próprios autores, de $\Omega(\log \log n + \log k / \log \log n)$ para a resolução deste problema no modelo CRCW PRAM para até n processadores.

Em relação a soluções probabilísticas, destaca-se o trabalho de Reischuk [Re-85], que propôs em 1985 um algoritmo probabilístico para o modelo PCT com n processadores que executa em tempo esperado constante $O(1)$. Este algoritmo seleciona aleatoriamente uma amostra de tamanho $\sqrt{n}$ da sequência de n elementos e a ordena em tempo constante, utilizando-a em seguida para obter dois separadores, possivelmente próximos ao k -ésimo elemento, que são então empregados no particionamento da sequência. Com alta probabilidade, o número de elementos restantes é reduzido a $\sqrt{n}$ após a execução de um número constante destas rodadas, sendo então possível comparar todos os elementos restantes em tempo constante.

Algoritmo	Modelo	Tipo	Trabalho	Tempo
Ordenação	EREW PRAM	Determ.	$O(n \log n)$	$O(\log n)$
Vishkin (1983)	EREW PRAM	Determ.	$O(n)$	$O(\log n \log \log n)$
Cole, Yap (1985)	PCT	Determ.	$O(n(\log \log n)^2)$	$O((\log \log n)^2)$
Ajtai, Komlós, Steiger, Szemerédi (1986)	PCT	Determ.	$O(n \log \log n)$	$O(\log \log n)$
Cole (1988)	EREW PRAM	Determ.	$O(n)$	$O(\log n \log^* n)$
Cole (1988)	CRCW PRAM	Determ.	$O(n)$	$O(\frac{\log n \log^* n}{\log \log n})$
Azar, Pippenger (1990)	PCT	Determ.	$O(n)$	$O(\log \log n)$
Chaudhuri, Hagerup, Raman (1993)	CRCW PRAM	Determ.	$O(n)$	$O(\frac{\log n}{\log \log n})$
Dietz, Raman (1994)	CRCW PRAM	Determ.	$O(n)$	$O(\log \log n + \log k / \log \log n)$
Reischuk (1985)	PCT	Probab.	$O(n)$	$O(1)$

Tabela 2: Algoritmos de seleção paralelos para memória compartilhada

5.3 Algoritmos Paralelos para Memória Distribuída

Existem poucos trabalhos publicados acerca deste problema para máquinas paralelas com memória distribuída, em sua maioria abordando o problema do ponto de vista de estruturas de interconexão específicas com $O(n)$ processadores ([Fr-88], [Ra-90], [ST-90], [Na-92], [RT-92], [Su-93], [Mc-93a], [Ch-95], [Ra-96], [KNR-96]).

Santoró, J. Sidney e S. Sidney desenvolvem em 1987 um algoritmo probabilístico que supõe uma distribuição aleatória uniforme dos elementos entre os processadores ([SSS-87], [SSS-92]). A cada rodada, considera-se como uma amostra aleatória a (maior) sequência de elementos armazenados na memória local de um dos processadores e escolhe-se nesta amostra um elemento que, com alta probabilidade, possua um *rank* próximo a k . Este elemento é então propagado aos demais processadores para a realização do particionamento, eliminando-se uma das partições e corrigindo-se o valor de k em relação aos elementos restantes. Com alta probabilidade, após $O(\log \log k)$ rodadas o valor corrigido de k será reduzido a $O(p)$, o que permite que cada processador elimine todos exceto seus k menores (ou maiores) elementos. Realiza-se então uma nova série de rodadas de particionamento a partir de elementos selecionados aleatoriamente, resolvendo-se o problema após mais $O(\log p)$ rodadas. Portanto este algoritmo requer um total de $O(\log \log k + \log p)$ rodadas de comunicação².

Em 1996, Bader e JáJá apresentam um novo algoritmo determinístico para computadores paralelos com memória distribuída segundo o modelo BDM [BJ-96] e desenvolvido a partir dos algoritmos anteriores (sequenciais e paralelos para memória compartilhada). A cada rodada, realiza-se inicialmente uma etapa de comunicação para redistribuir igualmente, entre os processadores, a sequência de elementos a serem analisados. Em seguida, cada processador calcula a mediana dos elementos armazenados em sua memória local, determina-se a mediana dessas medianas, particiona-se a sequência de elementos com base nesta mediana e elimina-se pelo menos $1/4$ destes elementos. Após $O(\log(n/p^2))$ rodadas (de comunicação), o número de elementos é reduzido de n para p^2 e os elementos restantes são então reunidos em um único processador, aplicando-se um algoritmo sequencial para completar a resolução do problema.

Também em 1996, Gerbessiotis e Siniolakis baseiam-se no algoritmo de Reischuk e propõem um novo algoritmo probabilístico para seleção da mediana de n elementos [GS-96]. Empregando um número constante de rodadas de comunicação, realiza-se uma amostragem aleatória dos elementos, distribui-se esta amostra igualmente entre os processadores, ordena-se apenas a amostra

² O texto cita também outro algoritmo, dos mesmos autores, para resolver o problema deterministicamente em $O(\log(n/p))$ rodadas, com um desempenho inferior ao de Bader e JáJá (apresentado a seguir). Infelizmente não foi possível obter esta referência.

(utilizando um algoritmo de ordenação paralelo para memória distribuída), selecionam-se dois separadores nesta amostra e realiza-se um particionamento com base nestes separadores. Com alta probabilidade, a partição contida entre os dois separadores contém a mediana e possui um tamanho reduzido o suficiente para que um único processador complete a resolução do problema. Portanto o número esperado de rodadas de comunicação é $O(1)$.

Finalmente, deve-se observar que os algoritmos de ordenação para memória distribuída podem ser empregados para resolver este problema deterministicamente em um número constante de rodadas de comunicação, às custas porém de um tempo significativamente maior de processamento local. Nestes algoritmos, em geral, cada processador realiza inicialmente a ordenação local de seus $O(n/p)$ elementos, uma amostragem determinística é utilizada para particionar e redistribuir sub-sequências ordenadas destes elementos entre os processadores, e então cada processador ordena localmente as sub-sequências recebidas ([Le-85], [SS-92], [LLS-93], [Ba-94], [LS-94], [ABK-95], [BHJ-95], [HBJ-95], [HBJ-96], [GS-96]).

Algoritmo	Tipo	Processamento Local	Rodadas de Comunicação
Ordenação	Determ.	$O((n/p) \log (n/p))$	$O(1)$
Santoro, J. Sidney, S. Sidney (1987)	Probab.	$O((n/p)(\log \log k + \log p))$	$O(\log \log k + \log p)$
Bader, JáJá (1996)	Determ.	$O(n/p)$	$O(\log (n/p^2))$
Gerbessiotis, Siniolakis (1996)	Probab.	$O(n/p)$	$O(1)$

Tabela 3: Algoritmos de seleção paralelos para memória distribuída

6. Algoritmos Propostos

Os novos algoritmos aqui propostos destinam-se a máquinas paralelas, com memória distribuída, de granularidade grossa $CGM(n, p)$, portanto com p processadores, cada um com memória local $O(n/p)$, interligados por uma rede arbitrária de comunicação. O termo “granularidade grossa” deve-se à imposição de que o tamanho n do problema seja significativamente maior do que o número p de processadores. Nestes algoritmos, considera-se que a sequência de entrada A de n elementos encontra-se inicialmente distribuída entre p processadores, com cada processador armazenando em sua memória local $O(n/p)$ elementos.

Estes algoritmos foram desenvolvidos exclusivamente a partir dos algoritmos sequenciais e paralelos para memória compartilhada apresentados anteriormente (principalmente com base no algoritmo proposto por Vishkin em 1983 para o modelo PRAM), independentemente de outros algoritmos para memória distribuída (pesquisados posteriormente).

6.1 Algoritmo básico de seleção do k -ésimo elemento

Os algoritmos de seleção propostos, assim como diversos algoritmos de seleção do k -ésimo elemento anteriores, baseiam-se na aplicação sucessiva da estratégia de particionamento, que funciona da seguinte forma. Escolhe-se inicialmente um elemento M da sequência A de tamanho n , particionando esta sequência nas sub-sequências A_l , A_e e A_g , respectivamente com os elementos menores, iguais e maiores que M . Em seguida, compara-se o valor de k com os tamanhos das três sub-sequências. Se k corresponde à primeira partição A_l (ou seja, $k \leq |A_l|$), então o algoritmo de seleção prossegue apenas com a sub-sequência A_l e descarta os demais elementos, reduzindo assim o tamanho do problema. Se k corresponde à partição A_e (ou seja, $|A_l| < k \leq |A_l| + |A_e|$), então M é a solução do problema. Se k corresponde à última partição A_g (ou seja, $k > |A_l| + |A_e|$), então o algoritmo de seleção prossegue com a sub-sequência A_g para selecionar o elemento $k - (|A_l| + |A_e|)$, novamente reduzindo o tamanho do problema.

No algoritmo paralelo básico de seleção, repetidamente os elementos são agrupados em blocos e utiliza-se uma mediana, escolhida entre as medianas de cada bloco, para realizar o particionamento, de forma a reduzir o número total de elementos a serem analisados em pelo menos $1/4$ a cada rodada.

Para reduzir o número de rodadas de comunicação necessárias, os particionamentos sucessivos são repetidos apenas até que o número total de elementos a serem analisados seja reduzido de $O(n)$ para $O(n/p)$, o que é obtido após $O(\log p)$ rodadas. Estes elementos restantes são então reunidos em um único processador para que o restante do problema seja resolvido sequencialmente em tempo $O(n/p)$. Deve-se lembrar que $\Omega(n/p)$ é claramente um limite inferior para o algoritmo paralelo, uma vez que o algoritmo sequencial requer tempo linear $\Theta(n)$.

Para também minimizar a comunicação entre os processadores em cada rodada, o algoritmo assume que cada bloco é composto apenas pela sequência de elementos já armazenados em cada processador. Embora o modelo CGM enfoque apenas o número de rodadas, considerou-se necessário também reduzir a comunicação entre os processadores a cada rodada do algoritmo, pois em máquinas paralelas com memória distribuída o custo de operações de comunicação é geralmente bem maior do que de processamento local.

Para garantir a eliminação de pelo menos $1/4$ dos elementos a cada rodada, o algoritmo poderia utilizar a mediana das medianas de cada bloco como base para o particionamento, o que exigiria que estes blocos tivessem o mesmo tamanho. Entretanto, após a primeira rodada permanece, em geral, um número diferente de elementos restantes em cada processador, sendo que a redistribuição destes elementos para garantir novamente uma mesma quantidade de elementos em cada processador implicaria em um custo muito elevado em termos de comunicação. Para evitar a necessidade de redistribuição, a solução encontrada é que a mediana escolhida para o particionamento não seja simplesmente a mediana das medianas, mas sim a “mediana ponderada” das medianas, de forma que nesta escolha sejam considerados não só os valores das medianas dos blocos mas também o número de elementos restantes em cada bloco. Isto permite que o algoritmo trabalhe nas rodadas seguintes com um número diferente de elementos em cada processador e mesmo assim seja atingido o objetivo de garantir a redução em pelo menos $1/4$ no número total de elementos restantes a cada rodada.

Ao evitar a redistribuição dos elementos a cada rodada, o algoritmo obtém uma redução considerável nos tempos de comunicação, suficiente para compensar o tempo adicional gasto com processamento local devido ao desbalanceamento de carga entre os processadores.

A definição geral de mediana ponderada é dada em [CLR-90]:

Para p elementos (distintos) $m_1, m_2, \dots, m_p$ com pesos positivos $w_1, w_2, \dots, w_p$ tais que $\sum_{i=1}^p w_i = 1$, a mediana ponderada é o elemento m_k que satisfaz $\sum_{m_i < m_k} w_i \leq \frac{1}{2}$ e $\sum_{m_i > m_k} w_i \leq \frac{1}{2}$.

Embora a definição acima, de acordo com a referência citada, refira-se apenas a elementos distintos, esta pode ser aplicada sem restrições (ao menos neste caso) a sequências que contenham elementos iguais. No caso do algoritmo descrito, a determinação da mediana ponderada é realizada considerando-se cada elemento m_i como a mediana dos elementos restantes no processador i , e com $w_i = n_i / N$, onde n_i corresponde ao número de elementos restantes no processador i e

$N = \sum_{i=1}^p n_i$ é o número total de elementos restantes.

O algoritmo detalhado é apresentado a seguir:

Algoritmo 1: Seleção do k -ésimo elemento.

Arquitetura: Máquina paralela com memória distribuída, com p processadores de memória local $O(n/p)$.

Entrada: Uma sequência A de n elementos distribuídos entre p processadores, com cada processador i armazenando em sua memória local $n_i = O(n/p)$ elementos, e um inteiro k tal que $1 \leq k \leq n$.

Saída: O k -ésimo menor elemento de A .

Algoritmo:

1. Iniciar $N := n$
2. Repetir rodada até $N \leq (n/p)$
 - 2.1 Cada processador i calcula a mediana m_i de seus n_i elementos

- 2.2 Cada processador i transfere m_i e n_i para o processador 1
- 2.3 Processador 1 calcula a mediana ponderada M
- 2.4 Processador 1 irradia M para os demais processadores
- 2.5 Cada processador i calcula os números de elementos locais menores l_i , iguais e_i e maiores g_i que M
- 2.6 Cada processador i transfere l_i , e_i e g_i para o processador 1
- 2.7 Processador 1 calcula os números totais de elementos menores $L = \sum_{i=1}^p l_i$, iguais $E = \sum_{i=1}^p e_i$ e maiores $G = \sum_{i=1}^p g_i$ que M
- 2.8 Processador 1 irradia L , E e G para os demais processadores
- 2.9 CASE:
 - se $L < k \leq L + E$: Retorna solução M e aborta execução do algoritmo
 - se $k \leq L$: Cada processador i compacta seus elementos, mantendo em A apenas os elementos menores que M , faz-se $N := L$
 - se $k > L + E$: Cada processador i compacta seus elementos, mantendo em A apenas os elementos maiores que M , faz-se $N := G$ e $k := k - (L + E)$
3. Todos os N elementos restantes são transferidos para o processador 1
4. Processador 1 termina sequencialmente a resolução do problema

O parâmetro c , introduzido neste algoritmo, é arbitrário. Com este parâmetro, o número de rodadas de comunicação (no passo 2) é $O(\log(cp))$ e o número total de elementos transmitidos na fase final de comunicação (no passo 3) é $O(n / cp)$. Assim, a utilização de um parâmetro $c \geq 1$ permite ajustar experimentalmente o algoritmo para máquinas paralelas com uma taxa de transmissão mais baixa, pois aumentando-se c obtém-se uma redução significativa na quantidade de elementos transmitidos na fase final às custas de um aumento apenas logarítmico no número de rodadas.

A propriedade atribuída à utilização da mediana ponderada é verificada através do seguinte teorema:

Teorema 1.1: A cada rodada (do passo 2) o número de elementos restantes é reduzido em pelo menos $1/4$.

Demonstração: No início de cada rodada, o número total de elementos restantes é N . Pela definição de mediana ponderada M :

$$\sum_{m_i < M} w_i \leq \frac{1}{2} \Rightarrow \frac{\sum_{m_i < M} n_i}{N} \leq \frac{1}{2} \Rightarrow \sum_{m_i < M} n_i \leq \frac{N}{2}$$

$$\sum_{m_i > M} w_i \leq \frac{1}{2} \Rightarrow \frac{\sum_{m_i > M} n_i}{N} \leq \frac{1}{2} \Rightarrow \sum_{m_i > M} n_i \leq \frac{N}{2}$$

$$\text{Como } N = \sum_i n_i, \text{ então } \sum_{m_i \leq M} n_i \geq \frac{N}{2} \text{ e } \sum_{m_i \geq M} n_i \geq \frac{N}{2}.$$

Para cada mediana local m_i , pela própria definição de mediana, o número de elementos a_j tais que $a_j \leq m_i$ é no mínimo $n_i / 2$, portanto a mediana ponderada M é maior ou igual a pelo menos $\sum_{m_i \leq M} \frac{n_i}{2} \geq \frac{N}{4}$ elementos. Analogamente, M é menor ou igual a pelo menos $\sum_{m_i \geq M} \frac{n_i}{2} \geq \frac{N}{4}$ elementos. Portanto o número total de elementos restantes que são estritamente menores que M é $L \leq (3N/4)$, e que são estritamente maiores que M é $G \leq (3N/4)$. Como na rodada seguinte são mantidos apenas L ou G elementos, tem-se que a cada rodada o número total de elementos restantes N é reduzido em pelo menos $1/4$. $\square$

Na análise do tempo de execução do algoritmo, considera-se $T_{transm}(p, x)$ o tempo de comunicação necessário para que cada um dos p processadores transmita uma informação de tamanho x para o processador 1, ou equivalentemente para o processador 1 enviar para cada um dos demais p processadores uma informação (possivelmente distinta) de tamanho x . Denomina-se $T_{concentr}(p, x)$ o tempo necessário para que uma informação de tamanho total x distribuída entre p processadores seja reunida no processador 1.

Teorema 1.2: O algoritmo paralelo proposto resolve o problema da seleção em tempo $O(\frac{n}{p} \log(cp) + T_{transm}(p, 1) \cdot \log(cp) + T_{concentr}(p, \frac{n}{cp}))$, para $n \geq p^2 \log p$ e $c \geq 1$.

Demonstração: Como a cada rodada do passo 2 o número de elementos restantes N é reduzido em pelo menos $1/4$, o número máximo de rodadas necessário para satisfazer a condição $N \leq (n / cp)$ é de $\lceil \log_{4/3}(cp) \rceil = O(\log(cp))$, uma vez que $c \geq 1$ e, evidentemente, $p \geq 2$.

Em relação aos passos que requerem apenas processamento local, o passo 2.1 é resolvido pelo algoritmo sequencial de seleção em tempo $O(n_i) = O(n / p)$, o passo 2.3 pode ser facilmente resolvido através de uma ordenação prévia das p medianas, portanto em tempo $O(p \log p)$, os passos 2.5 e 2.9 são resolvidos por algoritmos triviais em tempo $O(n_i) = O(n / p)$ e o passo 2.7 em tempo $O(p)$. O passo 4 consiste na aplicação do algoritmo sequencial de seleção para $N \leq (n / cp)$ elementos, portanto em tempo $O(n / cp)$.

Quanto aos passos de comunicação, os passos 2.2, 2.4, 2.6 e 2.8 consistem apenas na transmissão de mensagens de tamanho constante e podem ser executados em tempo $O(T_{transm}(p, 1))$. Já o passo 3 consiste na reunião de $N \leq (n / cp)$ elementos no processador 1, portanto em tempo $T_{concentr}(p, n / cp)$.

Atribui-se para a execução do passo 1 um tempo constante. Na realidade, mesmo que o número total de elementos n não seja previamente conhecido, este pode ser facilmente determinado a partir dos valores parciais n_i armazenados em cada processador i . Para tanto, bastaria executar um passo de comunicação em tempo $O(T_{transm}(p, 1))$ para reunir estes valores parciais em um único processador, somar localmente estes valores em tempo $O(p)$ e redistribuir o resultado $n = \sum_{i=1}^p n_i$ para os demais processadores utilizando novamente um passo de comunicação em tempo $O(T_{transm}(p, 1))$. De qualquer forma, estes tempos de execução seriam dominados assintoticamente pelos demais passos do algoritmo, não interferindo em sua análise. Do mesmo modo, supõe-se que o valor da constante k é inicialmente conhecido por todos os processadores.

Portanto o tempo total de execução do algoritmo paralelo é dado por:

$$O\left(\frac{n}{p} + p \log p + \frac{n}{p} + p + \frac{n}{p} + T_{transm}(p,1)\right) \log(cp) + T_{concentr}\left(p, \frac{n}{cp}\right) + \frac{n}{cp} =$$

$$O\left(\frac{n}{p} \log(cp) + T_{transm}(p,1) \cdot \log(cp) + T_{concentr}\left(p, \frac{n}{cp}\right)\right), \text{ para } n \geq p^2 \log p \text{ e } c \geq 1. \quad \square$$

A análise do algoritmo já permite também estimar boas possibilidades de aplicação prática, uma vez que as constantes envolvidas são relativamente pequenas. Neste algoritmo de seleção paralelo para n elementos, o tempo de execução é dominado assintoticamente, quanto ao processamento local, por no máximo $\lceil \log_{4/3}(cp) \rceil$ aplicações do algoritmo sequencial de seleção em no máximo (n/p) elementos (para uma quantidade inicial de elementos exatamente igual em cada processador), e quanto à comunicação, por no máximo $\lceil \log_{4/3}(cp) \rceil$ rodadas de transmissão de mensagens de tamanho constante (portanto o tempo de execução desta comunicação é independente de n) e por um único passo de reunião de no máximo (n/cp) elementos em um processador, uma quantidade que pode ser controlada através de uma escolha conveniente do parâmetro c .

6.2 Algoritmo de seleção simultânea de q “ k -ésimos” elementos

Analisa-se a seguir o problema de determinar q diferentes “ k -ésimos” elementos $k_1, k_2, \dots, k_q$ de uma mesma sequência $A = \{a_1, a_2, \dots, a_n\}$. Neste caso, ao invés de aplicar consecutivamente o algoritmo básico de seleção q vezes, pode-se adaptar este algoritmo para determinar estes q elementos simultaneamente.

Esta adaptação garante uma grande economia nas fases de comunicação, uma vez que a determinação simultânea destes q elementos pode ser realizada empregando-se o mesmo número de rodadas de comunicação, no pior caso, de uma única execução do algoritmo básico (para $q \leq p$), embora a quantidade total de informação transmitida seja até q vezes maior.

Mas a maior economia é obtida nas fases de processamento local, pois a determinação destes q elementos pode ser feita assintoticamente com o mesmo tempo de processamento local necessário para a determinação de um único k -ésimo elemento (para $q = O(p)$). Na primeira rodada, a determinação das medianas locais e da mediana ponderada e o particionamento consistem em passos com processamento local comum a todos os q problemas, portanto podem ser executados uma única vez, independentemente de q . Este particionamento inicial da sequência de elementos pode dividir estes q problemas em dois grupos, um relativo à sub-sequência com elementos menores do que a mediana ponderada e o outro à sub-sequência com elementos maiores. A cada nova rodada, cada grupo de problemas realiza um novo particionamento sobre uma sub-sequência

comum de elementos, eventualmente dividindo-se novamente em dois grupos de problemas a serem resolvidos em partições diferentes. Como, a cada rodada, os q problemas de seleção a serem resolvidos podem ser sempre agrupados em grupos de problemas, com todos os problemas de um grupo atuando sobre uma mesma sub-sequência de elementos, então as operações precisam ser processadas apenas uma vez por grupo a cada rodada. Além disso, cada grupo atua sobre uma sub-sequência disjunta de elementos, o que significa que cada elemento é processado por no máximo apenas um destes grupos a cada rodada. Assim, os passos de determinação das medianas locais e de particionamento podem ser executados em tempo linear em relação ao número de elementos armazenados em cada processador, independentemente de q . Portanto, o processamento local a cada rodada permanece $O(n / p)$, como no algoritmo básico de seleção.

De forma semelhante ao algoritmo anterior, estas rodadas são executadas apenas até que o número de elementos de cada um destes problemas (ou grupo de problemas) seja significativamente reduzido. Em seguida, os q problemas são redistribuídos entre os p processadores para terminar sequencialmente suas resoluções. Entretanto, para reduzir a quantidade de informações a serem transmitidas nesta fase final de remanejamento dos elementos, deve-se verificar que cada grupo de problemas seja atribuído a um mesmo processador, de forma a evitar que a mesma sub-sequência de elementos a ele relacionada tenha que ser enviada para processadores distintos.

Algoritmo 2: Seleção de q k -ésimos elementos.

Arquitetura: Máquina paralela com memória distribuída, com p processadores de memória local $O(n / p)$.

Entrada: Uma sequência A de n elementos distribuídos entre p processadores, com cada processador i armazenando em sua memória local uma sub-sequência A_i com $n_i = O(n / p)$ elementos, e uma sequência ordenada de inteiros $k_1, k_2, \dots, k_q$ tais que $1 \leq k_1 \leq k_2 \leq \dots \leq k_q \leq n$.

Saída: Uma sequência de elementos $b_1, b_2, \dots, b_q$ tal que b_j é o k_j -ésimo menor elemento de A , $1 \leq j \leq q$.

Algoritmo:

1. Iniciar $N_j := n$ e $A_{ij} := A_i$, $1 \leq j \leq q$

2. Repetir rodada até $N_j \leq \frac{n}{c \cdot \max\{q, p\}}$, simultaneamente para $1 \leq j \leq q$

2.1 Cada processador i calcula as medianas m_{ij} de seus n_{ij} elementos em A_{ij}

2.2 Cada processador i transfere m_{ij} e n_{ij} para o processador 1

2.3 Processador 1 calcula as medianas ponderadas M_j

2.4 Processador 1 irradia M_j para os demais processadores

- 2.5 Cada processador i calcula os números de elementos em A_{ij} menores l_{ij} , iguais e_{ij} e maiores g_{ij} que M_j
- 2.6 Cada processador i transfere l_{ij} , e_{ij} e g_{ij} para o processador 1
- 2.7 Processador 1 calcula os números totais de elementos menores $L_j = \sum_{i=1}^p l_{ij}$, iguais $E_j = \sum_{i=1}^p e_{ij}$ e maiores $G_j = \sum_{i=1}^p g_{ij}$ que M_j
- 2.8 Processador 1 irradia L_j , E_j e G_j para os demais processadores
- 2.9 CASE:
- se $L_j < k_j \leq L_j + E_j$: Retorna solução M_j e aborta seleção do k_j -ésimo elemento
 - se $k_j \leq L_j$: Cada processador i compacta seus elementos, mantendo em A_{ij} apenas os elementos menores que M_j , faz-se $N_j := L_j$
 - se $k_j > L_j + E_j$: Cada processador i compacta seus elementos, mantendo em A_{ij} apenas os elementos maiores que M_j , faz-se $N_j := G_j$ e $k_j := k_j - (L_j + E_j)$
3. Os q problemas de seleção são redistribuídos entre os p processadores, com os problemas relativos à mesma subsequência de elementos restantes sendo enviados ao mesmo processador e de forma a minimizar a quantidade máxima de elementos por processador
4. Cada processador termina sequencialmente a resolução de seus problemas recebidos

Este algoritmo pode ser considerado simplesmente uma generalização do algoritmo básico de seleção, pois para $q=1$ são executados exatamente os mesmos passos do algoritmo anterior.

Nesta próxima análise do tempo de execução do algoritmo, considera-se $T_{remanej}(p, s, x)$ o tempo de comunicação necessário para remanejar um conjunto de informações, distribuídas de forma não uniforme numa rede de p processadores, para um sub-conjunto de s destes processadores, de forma que cada processador receba um máximo de x informações.

Teorema 2.1: Este novo algoritmo paralelo proposto resolve simultaneamente q problemas de seleção (em relação à mesma sequência de n elementos) em tempo $O(\frac{n}{p} \log(c(q+p)) + T_{transm}(p, q) \cdot \log(c(q+p)) + T_{remanej}(p, \min\{q, p\}, \frac{n}{cp}))$, para $n \geq qp^2 \log p$ e $c \geq 1$.

Demonstração: A cada rodada do passo 2, o número de elementos restantes em N_j é reduzido em pelo menos $1/4$, para qualquer $1 \leq j \leq q$, portanto o número máximo de rodadas necessário para satisfazer a condição $N_j \leq \frac{n}{c \cdot \max\{q, p\}}$ é de $\lceil \log_{4/3}(c \cdot \max\{q, p\}) \rceil = O(\log(c(q+p)))$, com $c \geq 1$ e $p \geq 2$.

Em cada processador, o passo 2.1 consiste em no máximo q execuções do algoritmo sequencial de seleção sobre sub-sequências disjuntas de elementos locais, portanto o tempo total de execução de cada passo é de $O(n_i) = O(n/p)$. O passo 2.3 pode ser resolvido com no máximo q execuções de um algoritmo baseado na ordenação prévia das p medianas, portanto em tempo $O(qp \log p)$. Os passos 2.5 e 2.9 consistem em no máximo q execuções de um algoritmo linear trivial sobre sub-sequências disjuntas de elementos locais, portanto em tempo total

$O(n_i) = O(n/p)$. O passo 2.7 consiste em no máximo q somatórias de p elementos, portanto requer tempo $O(qp)$.

Os passos de comunicação 2.2, 2.4, 2.6 e 2.8 consistem na transmissão de mensagens de tamanho proporcional a q , sendo executados em tempo $O(T_{transm}(p, q))$.

Se $q \leq p$, o passo 3 consiste na distribuição, entre os p processadores, de no máximo q problemas de tamanho $N_j \leq n/cp$, portanto a comunicação neste passo é realizada em tempo $T_{remanej}(p, q, n/cp)$. Se $q > p$, o passo 3 consiste em redistribuir estes q problemas, cada um de tamanho $N_j \leq n/cq$, entre os p processadores, de forma a minimizar a quantidade máxima de elementos por processador, portanto cada um dos p processadores recebe no máximo $O(\frac{q}{p} \cdot \frac{n}{cq}) = O(\frac{n}{cp})$ elementos, em tempo $O(T_{remanej}(p, p, n/cp))$.

No passo 4, cada processador possui um conjunto de sub-sequências com um total de no máximo $O(n/cp)$ elementos e deve resolver no máximo q problemas de seleção sobre estes elementos (o que ocorreria caso todas as q k -ésimas permanecessem no mesmo grupo de problemas), o que pode ser resolvido em tempo $O((n/cp) \log q)$.

Da mesma forma que no algoritmo anterior, atribui-se para a execução do passo 1 um tempo constante. A determinação do número total de elementos n poderia ser realizada facilmente a partir dos valores n_i locais a cada processador em tempo total $O(p + T_{transm}(p, 1))$, não influenciando no comportamento assintótico do algoritmo. Até mesmo a ordenação da sequência de valores $k_1, k_2, \dots, k_q$ e a distribuição destes valores para todos os processadores poderiam ser realizadas, em tempo $O(q \log q)$, eliminando-se o requisito do algoritmo quanto a esta sequência estar previamente ordenada, sem afetar o seu comportamento.

Assim o tempo total de execução do algoritmo paralelo é dado por:

$$\begin{aligned} & O\left(\left[\frac{n}{p} + qp \log p + \frac{n}{p} + qp + \frac{n}{p} + T_{transm}(p, q)\right] \log(c(q + p)) + T_{remanej}\left(p, \min\{q, p\}, \frac{n}{cp}\right) + \right. \\ & \quad \left. + \frac{n}{cp} \log q\right) = \\ & O\left(\frac{n}{p} \log(c(q + p)) + T_{transm}(p, q) \cdot \log(c(q + p)) + T_{remanej}\left(p, \min\{q, p\}, \frac{n}{cp}\right)\right), \end{aligned}$$

para $n \geq qp^2 \log p$ e $c \geq 1$. $\square$

Para comparação, deve-se observar que a simples aplicação consecutiva do algoritmo básico de seleção q vezes resultaria em um tempo total de execução igual a

$O(q \cdot \frac{n}{p} \log(cp) + q \cdot T_{transm}(p,1) \cdot \log(cp) + q \cdot T_{concentr}(p, \frac{n}{cp}))$, portanto um desempenho claramente inferior.

Outra alternativa seria dividir os problemas inicialmente entre os processadores disponíveis. Para valores de $q \leq p$, seriam alocados p / q processadores para resolver cada problema utilizando o algoritmo básico de seleção (o que exigiria que a memória local de cada processador fosse proporcionalmente aumentada em q vezes para permitir o armazenamento de uma cópia da sequência de elementos A em cada um desses grupos de processadores), portanto o tempo total de execução seria de $O(q \cdot \frac{n}{p} \log(\frac{cp}{q}) + T_{transm}(\frac{p}{q}, 1) \cdot \log(\frac{cp}{q}) + T_{concentr}(\frac{p}{q}, q \cdot \frac{n}{cp}))$. Para valores de $q > p$, cada processador poderia ser alocado para resolver sequencialmente q / p problemas (o que exigiria uma memória local de tamanho $O(n)$), em tempo total $O(n \log(q / p))$. Estes resultados são melhores do que os obtidos aplicando-se o algoritmo básico de seleção q vezes, mas ainda exigiriam um tempo de processamento local muito maior do que o do novo algoritmo de seleção simultânea proposto. Deve-se novamente lembrar que neste algoritmo, em relação ao tempo de comunicação, os passos de transmissão (executados a cada rodada) são independentes de n e o passo final de remanejamento dos elementos pode ser controlado através da escolha conveniente do parâmetro c .

6.3 Algoritmo de ordenação baseado em seleção

O algoritmo anterior pode ser utilizado para a implementação de um método de ordenação completa da sequência A . Através do algoritmo de seleção simultânea, são determinados $p - 1$ elementos que particionam a sequência a ser ordenada em p sub-sequências de n / p elementos cada. Então os elementos são redistribuídos de forma que cada sub-sequência seja reunida em um processador e em seguida cada processador ordena sequencialmente a sua sub-sequência recebida.

Entretanto, como os elementos da sequência não são todos necessariamente distintos, podem existir diversos elementos iguais aos separadores selecionados. Neste caso, é preciso decidir como dividir estes elementos entre as sub-sequências de forma que todas estas contenham o mesmo número de elementos, o que requer alguns passos adicionais para que o processador 1 determine esta divisão com base no número de elementos menores, iguais e maiores que cada separador.

Algoritmo 3: Ordenação baseada em seleção.

Arquitetura: Máquina paralela com memória distribuída, com p processadores de memória local $O(n / p)$.

Entrada: Uma sequência A de n elementos distribuídos entre p processadores, com cada processador i armazenando em sua memória local $n_i = O(n / p)$ elementos.

Saída: A sequência A ordenada e distribuída entre p processadores, de forma que cada processador i armazena uma sub-sequência ordenada A_i de A tal que para $\forall a \in A_r$ tem-se $a \geq b$ para $\forall b \in A_s$, $s < r$.

Algoritmo:

- 1...4. Aplicar o algoritmo 2 para $q = p-1$ e $k_j = j \cdot n / p$, $1 \leq j \leq q$
5. Cada processador i comunica as soluções dos seus problemas finalizados localmente, de forma que todos os processadores conheçam todos os q k_j -ésimos elementos
6. Cada processador i separa os seus elementos iniciais de A em $O(p)$ sub-sequências a partir destes q k_j -ésimos elementos
7. Cada processador i transfere para o processador 1 o número de elementos locais menores, iguais e maiores que cada k_j -ésimo elemento
8. Processador 1 determina, para cada processador i e cada k -ésimo elemento k_j , como re-dividir os três grupos de elementos locais $<$, $=$ ou $>$ que a_{k_j} em apenas dois grupos de elementos locais $\leq$ e $\geq$ que a_{k_j} , de forma que cada grupo global tenha no máximo $\lceil n / p \rceil$ elementos
9. Processador 1 transmite para cada processador i as informações acerca destas divisões
10. As p sub-sequências de cada processador são redistribuídas entre os p processadores, de forma que cada processador i receba um total de $O(n / p)$ elementos correspondentes às i -ésimas sub-sequências
11. Cada processador i ordena localmente seus elementos recebidos

Para a análise deste algoritmo define-se $T_{reloc}(p, x)$ como o tempo de comunicação necessário para realizar uma realocação de elementos numa rede com p processadores, sendo que cada processador transmite e recebe um máximo de x elementos.

Teorema 3.1: O algoritmo de ordenação baseado em seleção resolve o problema em tempo

$$O\left(\frac{n}{p} \log\left(\frac{n}{p}\right) + T_{transm}(p, p) \cdot \log(cp) + T_{remanej}\left(p, p, \frac{n}{cp}\right) + T_{reloc}\left(p, \frac{n}{p}\right)\right), \text{ para } n \geq p^3 \log p, \quad c \geq 1 \text{ e } n \geq cp^2.$$

Demonstração: De acordo com o teorema 2.1, os passos 1 a 4 são processados em tempo $O(\frac{n}{p} \log(cp) + T_{transm}(p, p) \cdot \log(cp) + T_{remanej}(p, p, \frac{n}{cp}))$, para $n \geq p^3 \log p$ e $c \geq 1$.

O passo 5 pode ser implementado através de um remanejamento das soluções dos problemas de seleção entre os p processadores, de forma que cada processador receba as mesmas $O(p)$ soluções, portanto em tempo $O(T_{remanej}(p, p, p))$.

No passo 6, cada processador i separa seus $O(n/p)$ elementos armazenados localmente em p sub-sequências. Realizando-se esta separação de forma binária, ou seja, iniciando o particionamento com base no $k_{\lceil p/2 \rceil}$ -ésimo elemento e assim por diante, este passo pode ser executado em tempo $O((n/p) \log p)$.

O passo 7 consiste na transmissão de uma informação de tamanho $O(p)$ a partir de cada processador para o processador 1, portanto em tempo $O(T_{transm}(p, p))$.

O passo 8 pode ser resolvido com p execuções de um algoritmo trivial de tempo linear $O(p)$, portanto em tempo total $O(p^2)$.

No passo 9 realiza-se a transmissão de $O(p)$ informações para cada processador i a partir do processador 1, portanto em tempo $O(T_{transm}(p, p))$.

O passo 10 consiste na redistribuição dos elementos de A a partir da separação anterior, com cada processador enviando e recebendo um total de $O(n/p)$ elementos, portanto em tempo $O(T_{reloc}(p, n/p))$.

Finalmente, no passo 11 cada processador executa um algoritmo sequencial de ordenação sobre $O(n/p)$ elementos em tempo $O((n/p) \log(n/p))$.

O tempo total de execução do algoritmo é de:

$$\begin{aligned} & O\left(\frac{n}{p} \log(cp) + T_{transm}(p, p) \cdot \log(cp) + T_{remanej}\left(p, p, \frac{n}{cp}\right) + T_{remanej}(p, p, p) + \frac{n}{p} \log p + \right. \\ & \quad \left. + T_{transm}(p, p) + p^2 + T_{reloc}\left(p, \frac{n}{p}\right) + \frac{n}{p} \log\left(\frac{n}{p}\right)\right) = \\ & O\left(\frac{n}{p} \log\left(\frac{n}{p}\right) + T_{transm}(p, p) \cdot \log(cp) + T_{remanej}\left(p, p, \frac{n}{cp}\right) + T_{reloc}\left(p, \frac{n}{p}\right)\right), \end{aligned}$$

para $n \geq p^3 \log p$, $c \geq 1$ e $n \geq cp^2$. $\square$

Neste tempo de execução assintótico obtido, o componente relativo ao processamento local $O((n/p) \log(n/p))$ refere-se à ordenação dos elementos armazenados em cada processador, não podendo portanto ser evitado, e quanto à comunicação, há um componente relativo à transmissão de apenas $O(p)$ elementos por $O(\log(cp))$ rodadas (portanto um componente independente de n), um componente relativo ao remanejamento de elementos necessário para finalizar o processo de seleção simultânea (que pode ser controlado através de uma escolha conveniente do parâmetro c) e finalmente um componente relativo à redistribuição final dos elementos entre os processadores, componente este inevitável dada a própria especificação do problema.

Finalmente deve-se observar que, como $n \geq p^2$ então:

$$n \geq \frac{n}{p} \geq \frac{n}{\sqrt{n}} = \sqrt{n} \Rightarrow \log(n) \geq \log\left(\frac{n}{p}\right) \geq \frac{1}{2} \log(n) \Rightarrow O\left(\log\left(\frac{n}{p}\right)\right) = O(\log(n))$$

Portanto este componente relativo ao processamento local $O((n/p) \log(n/p))$ corresponde exatamente ao limite inferior de $\Omega(n \cdot \log(n)/p)$ estabelecido a partir da complexidade sequencial da ordenação.

7. Implementação em PowerXplorer (PVM)

Os três novos algoritmos propostos neste trabalho foram implementados em linguagem C e testados inicialmente na máquina paralela “Parsytec PowerXplorer 16/32 Parallel Computing System”³. Este computador é composto de 16 nós interconectados em topologia 2D (grade), sendo que cada nó possui um processador de aplicação *PowerPC601* de 80 MHz, com memória local de 32 Mbytes, e um processador auxiliar de comunicação *T805*.

A implementação dos algoritmos utiliza a interface de comunicação PVM (*Parallel Virtual Machine* [GBDJMS-93], [FV-94]), que foi escolhida por ser uma interface de comunicação padrão. Garante-se assim a portabilidade da implementação, às custas de uma pequena perda de desempenho em relação à utilização direta das primitivas de comunicação específicas da interface PARIX ([Vi-94a], [Vi-94b]) fornecida pelo fabricante⁴.

Os algoritmos trabalham com sequências de elementos representados em ponto flutuante de dupla precisão (tipo “*double*”), com cada processador⁵ armazenando inicialmente a mesma quantidade de elementos. As implementações permitem o processamento de sequências de entrada contendo elementos repetidos. Se fosse imposta a restrição de aceitar apenas sequências de elementos distintos, seriam possíveis algumas simplificações nos algoritmos, principalmente permitindo a utilização de uma rotina sequencial de particionamento mais eficiente (cada operação de particionamento poderia ser realizada com uma única “varredura” pela sequência de elementos), lembrando-se que esta rotina é utilizada tanto pelos algoritmos de seleção paralelos como pelos sequenciais. Entretanto testes preliminares indicaram que, com estas simplificações, o ganho no desempenho do algoritmo sequencial seria da ordem de apenas 2% (medidos para sequências com

³ Equipamento adquirido pelo Instituto de Matemática e Estatística da USP para o projeto ASPAS (“*Application Oriented Software Development for Highly Parallel Systems*”), supervisionado pelo Prof. Dr. Siang Wun Song, como parte da iniciativa CEC/ITDC (“*Commission of the European Communities/Information Technologies in Developing Countries*”).

⁴ Alguns destes algoritmos chegaram a ser também implementados na interface PARIX, mas testes preliminares efetuados com o algoritmo básico de seleção indicaram um desempenho muito próximo nas duas interfaces, o que nos levou a decidir realizar o estudo de desempenho apenas para a interface PVM.

⁵ Neste trabalho, o termo “processador”, quando empregado de forma genérica, refere-se ao processador (principal) de aplicação de cada nó de processamento da máquina paralela, conforme o costume usual.

distribuição uniforme), prevendo-se um ganho ainda menor no caso do algoritmo paralelo, portanto estas simplificações não teriam um efeito significativo o suficiente para interferir na análise dos resultados dos algoritmos.

Os tempos de execução apresentados são medidos em “ticks” do relógio da máquina, sendo que cada “tick” corresponde a 10^{-6} segundos. A contagem de tempo não inclui o tempo de carga inicial dos dados de entrada na memória dos processadores, realizada antes de cada teste.

O tempo de processamento paralelo consiste no intervalo de tempo entre o instante em que o primeiro processador inicia a execução e o instante em que o último processador termina de executar o algoritmo. Entretanto, no caso da máquina paralela com memória distribuída não há uma contagem de tempo centralizada ou um relógio global para ser utilizado como referência na cronometragem, o que dificulta uma medição precisa deste intervalo de tempo. Em vista disso, a implementação dos algoritmos foi planejada considerando-se também formas de permitir uma contagem de tempo mais precisa.

Antes de cada execução, é realizada uma etapa inicial de sincronização dos processadores, que funciona da seguinte maneira: após a carga inicial dos dados, cada processador envia uma mensagem para o processador 1 indicando estar pronto para iniciar a execução do algoritmo e então aguarda uma resposta. O processador 1, após receber as mensagens de todos os processadores, irradia uma única mensagem para todos os demais e logo em seguida inicia a execução do algoritmo. Os demais processadores também iniciam a execução à medida em que recebem esta mensagem. Desta forma, garante-se que o processador 1 inicie a execução do algoritmo sempre antes dos demais processadores, mas impondo-se a estes outros processadores o menor atraso possível (apenas o tempo de transmissão de uma mensagem).

Por outro lado, a própria construção dos algoritmos garante que este processador 1 é sempre o último a terminar a execução do algoritmo. No algoritmo 1, isto ocorre naturalmente pois apenas o processador 1 executa o último passo. No algoritmo 2, este comportamento é obtido fazendo-se com que, nesta implementação, todos os resultados obtidos pelos diversos processadores sejam transmitidos para o processador 1 ao final da execução. No algoritmo 3, isto também ocorre naturalmente porque a carga de processamento atribuída para o processador 1 no último passo é no mínimo igual à dos outros processadores.

Assim, o tempo total de execução dos algoritmos é obtido simplesmente medindo-se o tempo de execução apenas no processador 1. Esta solução tem a vantagem de dispensar a realização de uma nova sincronização ao final da execução do algoritmo, o que acarretaria em um aumento adicional no tempo de execução.

As medições para o caso $p=1$ são sempre obtidas com a execução do melhor algoritmo sequencial (determinístico) em um único processador da máquina paralela e não com a execução do algoritmo paralelo em um processador. Os algoritmos sequenciais de particionamento, seleção e ordenação baseiam-se nas implementações apresentadas em [Wi-89] e [Se-90] e foram aperfeiçoados para garantir o melhor desempenho possível. Até mesmo a alocação de áreas

temporárias de trabalho na memória dinâmica, que normalmente representaria uma parte significativa do tempo de processamento sequencial, é agora realizada antes da execução dos algoritmos, de forma a não ser considerada na medição do tempo de execução.

O objetivo dos testes não é simplesmente medir o desempenho dos algoritmos em uma máquina paralela específica, mas principalmente analisar o comportamento destes algoritmos em uma real situação prática. A ênfase deste estudo não está portanto na obtenção de valores isolados de desempenho para determinadas condições e sim no levantamento das variações deste desempenho em função dos diversos parâmetros dos algoritmos. Por isso, optou-se pela apresentação dos resultados na forma de gráficos.

Cada teste de desempenho é repetido 20 vezes e os gráficos de tempo de execução apresentados indicam o pior tempo obtido. Pretende-se assim analisar os algoritmos sempre na situação mais desfavorável, uma vez que a maior vantagem de um algoritmo determinístico em relação a algoritmos probabilísticos deve ser apresentar o comportamento desejado em todas as suas execuções, não permitindo a possibilidade de um desempenho inferior mesmo com baixa probabilidade. Em geral observa-se uma diferença de cerca de 5% entre as medidas de melhor e pior tempo obtidas para cada teste.

O apêndice A contém as listagens dos algoritmos desenvolvidos, que incluem comentários mais detalhados acerca das implementações. Os gráficos de desempenho obtidos para a interface PVM na máquina paralela “Parsytec PowerXplorer” são apresentados nas figuras 1 a 24 do apêndice B.

7.1 Algoritmo básico de seleção

7.1.1 Descrição dos casos de teste

A análise das características de desempenho deste algoritmo baseia-se no estudo de 4 casos de teste, escolhidos de forma a representar os diversos tipos de problemas e distribuições dos dados de entrada. Estes casos são denominados da seguinte forma:

- “Caso Médio”: Os dados de entrada são distribuídos uniformemente (aleatoriamente) entre os processadores. O valor de k é estabelecido em $1/3$ do número de elementos. Com esta distribuição uniforme, este caso representa algumas das aplicações práticas mais frequentes do algoritmo.
- “Caso Especial”: Os dados de entrada encontram-se já previamente ordenados, com o processador 1 contendo inicialmente a sub-sequência (ordenada) dos menores elementos, o processador 2 contendo a sub-sequência com os menores elementos entre os demais e assim por diante. Adota-se $k=1$, o que equivale a selecionar o menor elemento da sequência. Esta distribuição representa um caso especial de funcionamento do algoritmo, pois após $O(\log p)$ rodadas todos os elementos restantes já estão contidos exclusivamente no processador 1, o que dispensa a execução

da fase final de comunicação do algoritmo (que normalmente representaria a maior parte do tempo gasto em comunicação).

- “Pior Caso”: Os dados de entrada encontram-se em ordem decrescente, com o processador 1 contendo inicialmente a sub-sequência (inversamente ordenada) dos maiores elementos e assim por diante. Adota-se também $k=1$. O termo “pior caso” é aqui empregado no sentido de ser esta a distribuição mais desfavorável à paralelização, por provocar a pior “distribuição de carga” (“*load balancing*”) entre os processadores (após $O(\log p)$ rodadas, praticamente todos os elementos restantes encontram-se no processador p e maior quantidade de dados transmitidos na fase final de comunicação (os elementos restantes devem ser transmitidos do processador p para o processador 1), ao mesmo tempo em que esta é uma distribuição de dados muito favorável à aplicação do algoritmo sequencial porque o fato dos elementos estarem ordenados facilita a operação de particionamento. Este é portanto o caso mais difícil para a obtenção de tempos de execução paralelos melhores que os tempos sequenciais.

- “Pior Caso Randômico”: Os dados de entrada encontram-se distribuídos em ordem decrescente apenas entre os processadores (o processador 1 contém inicialmente a sub-sequência dos maiores elementos e assim por diante), mas na memória de cada processador esses dados não são ordenados e sim distribuídos uniformemente. Adota-se $k=1$. Este caso possui não apenas as mesmas características de comunicação e “distribuição de carga” desfavoráveis do “Pior Caso”, mas também uma distribuição inicial de dados desfavorável para as fases de processamento local.

O algoritmo sequencial determinístico PICK de seleção é utilizado como o algoritmo sequencial (para $p=1$) e também como implementação da fase final do algoritmo paralelo. O algoritmo PICK foi ajustado para operar com base em grupos de 21 elementos, uma vez que este valor havia sido sugerido pelos próprios autores no artigo original [BFPRT-73] e testes preliminares (para sequências com distribuição uniforme) indicaram que este valor realmente pertencia ao conjunto de constantes em que o algoritmo apresentava melhor desempenho. Mas este algoritmo emprega uma rotina de particionamento modificada para aceitar também sequências contendo elementos repetidos, o que não era previsto pelo algoritmo original.

7.1.2 Análise do parâmetro c

Os primeiros testes de desempenho visam estudar a forma como o parâmetro c interfere no funcionamento do algoritmo. Para tanto, o algoritmo básico de seleção é aplicado para diversos valores de c em cada um dos casos de estudo. O tamanho do problema é fixado inicialmente em 1M elementos (ou seja, $2^{20} = 1048576$ elementos), o que corresponde à maior sequência de entrada que pode ser armazenada em um único processador devido à limitação de memória local da máquina paralela utilizada.

A figura 1 apresenta gráficos do pior tempo total de execução observado (em 20 execuções) em função do parâmetro c . Estes gráficos já permitem comparar o desempenho do algoritmo paralelo em relação ao do algoritmo sequencial. A execução em paralelo com apenas 2

processadores apresenta um desempenho inferior ao do algoritmo sequencial em todos os casos, devido aos custos adicionais (como comunicação) inerentes ao processamento em paralelo. Entretanto, a partir de 4 processadores o desempenho do algoritmo paralelo já supera o do sequencial em todos exceto no “Pior Caso”, que requer 8 processadores para superá-lo. Também já é possível observar que o tempo de execução do algoritmo paralelo efetivamente diminui com o aumento do número de processadores.

Nestes gráficos observa-se que os tempos de execução do “Pior Caso” são em geral menores que os tempos do “Caso Médio”. Isto ocorre porque o conceito de “Pior Caso” foi aqui empregado apenas no sentido de caso mais desfavorável à aplicação do algoritmo paralelo em relação ao algoritmo sequencial, como explicado anteriormente. De fato, o “Pior Caso” é o único em que o desempenho do algoritmo sequencial ainda é melhor que o do algoritmo paralelo para 4 processadores. Já o “Pior Caso Randômico” apresenta tempos de execução maiores que os demais casos, pois sua sequência de entrada é desfavorável para a aplicação de algoritmos paralelos tanto em termos de comunicação como de processamento local.

O parâmetro c havia sido introduzido na apresentação dos algoritmos com a intenção de permitir ajustar o número de rodadas e assim controlar o número de elementos restantes a serem transmitidos na fase final de comunicação. Os gráficos obtidos justificam claramente a utilização deste parâmetro. Para valores pequenos de c (em geral para c inferior a 10), um pequeno aumento neste parâmetro implica em uma redução significativa no número de elementos restantes a serem transmitidos, reduzindo bastante o tempo gasto na fase final de comunicação, às custas de um pequeno aumento no número de rodadas do algoritmo. Observa-se assim que, em 3 dos 4 casos estudados, o tempo de execução inicialmente decai rapidamente com o aumento do parâmetro c . A partir de determinado ponto, o número de elementos restantes passa a ser pequeno o suficiente para que a fase final de comunicação deixe de representar uma parcela significativa do tempo de execução do algoritmo, assim o aumento de c (e portanto aumento no número de rodadas do passo 2) não mais proporciona uma redução significativa no tempo gasto pelo passo final de comunicação.

No entanto, o “Caso Especial” apresenta um comportamento diferente, com o tempo total de execução sempre aumentando em relação ao aumento de c , principalmente para valores pequenos deste parâmetro. Isto ocorre porque, após as primeiras $O(\log p)$ rodadas, apenas o processador 1 armazena todos os elementos restantes, portanto o aumento do parâmetro c e o consequente aumento no número de rodadas do algoritmo paralelo não proporcionam uma redução no tempo gasto na fase final de comunicação, mas apenas um aumento no tempo de processamento do passo 2.

Os gráficos da figura 2 indicam o número de elementos restantes (após o passo 2) em função do parâmetro c em cada caso. Estes gráficos comprovam que o número de elementos restantes é inversamente proporcional ao parâmetro c , conforme esperado, uma vez que as rodadas do passo 2 são repetidas até que o número total de elementos restantes seja reduzido de n para (n / cp) . É possível notar também que este número de elementos restantes diminui com o aumento de p .

Observa-se ainda que, embora a distribuição inicial dos elementos não influencie muito significativamente a relação entre o número de elementos restantes e o parâmetro c , algumas distribuições são mais favoráveis para permitir que o k -ésimo elemento seja determinado já durante as rodadas do passo 2 (para valores suficientemente grandes de c), o que justifica a “interrupção” observada principalmente nas curvas do gráfico de “Caso Médio”.

A figura 3 mostra o número de rodadas de execução (do passo 2) em função do parâmetro c . Pode-se observar que o número de rodadas é proporcional ao logaritmo de c (para p fixo) e que também aumenta com o aumento de p , pois de acordo com o Teorema 1.2 o número de rodadas é $O(\log(cp))$. Observa-se ainda que, mesmo para valores grandes de c , ocorre uma limitação no número de rodadas, principalmente no “Caso Médio” (em 18 rodadas para 2 a 4 processadores e em 16 rodadas para 8 a 16 processadores), uma vez que este número de rodadas já é suficiente para a determinação direta do k -ésimo elemento durante o passo 2, confirmando o comportamento observado nos gráficos de elementos restantes da figura anterior.

7.1.3 Análise de tempos parciais

A seguir realiza-se a medição dos tempos parciais de execução, com o objetivo de comprovar as considerações anteriores sobre como o parâmetro c influencia os passos do algoritmo. Neste estudo são identificados os tempos parciais dos 4 passos principais deste algoritmo: passo 1 (apenas atribuição de valores iniciais a variáveis de controle), passo 2 (rodadas de comunicação e processamento em paralelo), passo 3 (transferência dos elementos restantes para um único processador) e passo 4 (resolução sequencial do problema restante). Os tempos parciais são medidos com uma única execução do algoritmo em cada caso.

A figura 4 apresenta os gráficos obtidos para cada caso de teste com o algoritmo básico de seleção executado em apenas 2 processadores. Na maioria dos casos estudados (exceto no “Caso Especial”), observa-se que para $c=1$ os tempos parciais dos passos 3 e 4 representam uma parcela muito grande do tempo total, devido ao grande número de elementos restantes ao final do passo 2. Estes passos 3 e 4 são mais significativos no “Caso Médio”, no qual chegam a representar quase 80% deste tempo total, porque este caso corresponde ao maior número de elementos restantes (conforme figura 2). Aumentando-se o valor de c para 10, ocorre um aumento significativo no tempo parcial correspondente ao passo 2 (pois aumenta o número de rodadas), porém há também uma redução muito grande nos tempos parciais dos passos 3 e 4 (pois diminui o número de elementos restantes), o que resulta em uma diminuição considerável no tempo total observado, principalmente no “Caso Médio” onde os passos 3 e 4 representavam uma parcela muito grande do tempo total. Aumentando-se novamente este parâmetro, desta vez para $c=100$, ocorre agora mais um pequeno aumento no tempo parcial do passo 2, enquanto os tempos parciais relativos aos passos 3 e 4 deixam de ser significativos em relação ao tempo total, o que faz com que em geral este tempo total apresente uma pequena redução. A partir deste ponto, aumentando ainda mais o valor de c observa-se um aumento muito pequeno no tempo parcial do passo 2 e portanto no tempo

total, uma vez que apenas a parcela de tempo correspondente ao passo 2 é significativa, sendo este aumento observável principalmente no “Pior Caso Randômico”.

Estes gráficos permitem também verificar que a velocidade de comunicação é bem menor do que a velocidade de processamento local, pois em geral a transmissão dos elementos restantes no passo 3 exige um tempo de execução consideravelmente maior do que a aplicação do algoritmo sequencial a estes mesmos elementos no passo 4.

Já no gráfico do “Caso Especial” não é observada a parcela de tempo relativa ao passo 3, uma vez que neste caso não é necessária a transmissão dos elementos restantes. Por isso, para valores pequenos de c , um aumento no valor deste parâmetro corresponde a uma diminuição apenas no tempo parcial do passo 4, que é superada pelo aumento no tempo parcial do passo 2, fazendo com que o tempo total aumente com o aumento de c . Assim como ocorre nos outros casos, a partir de $c=100$ apenas o tempo parcial do passo 2 é significativo, com aumentos no valor de c correspondendo a pequenos aumentos no tempo total de execução do algoritmo.

A figura 5 apresenta os gráficos dos resultados obtidos para os mesmos testes utilizando agora 16 processadores. A principal diferença em relação aos testes anteriores é o fato dos passos 3 e 4 corresponderem a uma parcela de tempo menor, chegando a apenas cerca de 50% do tempo total no “Caso Médio” para $c=1$. O tempo parcial do passo 2 é relativamente maior, em comparação com os tempos dos passos 3 e 4, porque o aumento no número de processadores implica em um aumento (logarítmico) no número de rodadas e um custo maior das operações de comunicação no passo 2, ao mesmo tempo em que reduz (de forma inversamente proporcional) o custo de processamento local em cada rodada e o número de elementos restantes para os passos 3 e 4. Portanto há uma redução maior nos tempos parciais dos passos 3 e 4 do que do passo 2. Apesar desta diferença, as mesmas observações anteriores relativas à figura 4 aplicam-se aqui novamente para todos os casos de teste.

É possível ainda notar que os gráficos de “Caso Médio” e “Pior Caso Randômico” para 16 processadores apresentam variações nos tempos de execução mais “irregulares”. Nestes casos de teste, a carga de processamento local de cada processador apresenta uma certa variação devido à distribuição aleatória dos elementos em cada processador. Como o tempo de execução de cada rodada é determinado pelo processador que gastar mais tempo para executar o seu processamento local, um número maior de processadores permite variações de tempo também maiores.

7.1.4 Determinação do parâmetro c

A figura 6 apresenta uma comparação de tempos de execução do algoritmo paralelo entre os diversos casos de teste. Assim como no primeiro conjunto de gráficos, é possível observar, na maior parte dos casos, uma diminuição considerável no tempo de execução com o aumento inicial de c , devido à grande redução no número de elementos restantes. A partir da região entre $c=10$ e $c=100$, esta diminuição deixa de ser significativa, pois o número de elementos restantes torna-se

muito pequeno para que alterações nos tempos parciais dos passos 3 e 4 sejam relevantes em relação ao tempo total.

Em alguns casos observa-se agora que, para valores de c acima desta região, à medida em que o valor do parâmetro c aumenta, o tempo de execução também passa a aumentar, sendo este comportamento mais claramente observado no caso de problemas menores resolvidos por um número maior de processadores. O aumento do parâmetro c implica em um aumento no número de rodadas, portanto um aumento no tempo gasto na execução do passo 2 do algoritmo. Mas cada rodada do passo 2 requer um tempo de comunicação relativamente baixo, independente de n , portanto este aumento de tempo do passo 2 normalmente não afeta significativamente o tempo total. Mas para valores menores de n e maiores de p , o tempo total de execução é consideravelmente menor, portanto este aumento de tempo no passo 2 passa a ter um peso maior em relação ao tempo total. Valores maiores de p também significam tempos de comunicação maiores e um maior número de rodadas do passo 2.

Conclui-se portanto que uma constante a ser adotada para o parâmetro c deve situar-se após a região de queda de tempo mais significativa da maioria dos casos (região de valores de c entre 10 e 100), mas este valor também não deve ser muito alto porque em alguns casos há um aumento de tempo sensível com o aumento de c . Em vista disso, o valor $c=100$ foi escolhido como o parâmetro mais adequado para esta máquina paralela, sendo adotado para a realização de todos os testes seguintes.

7.1.5 Análise de desempenho

A figura 7 apresenta os tempos de execução do algoritmo para c fixo. Em todos os casos, o tempo de execução do algoritmo paralelo em 2 processadores é maior que o tempo do algoritmo sequencial, devido às rotinas de processamento local mais complexas do algoritmo paralelo e ao custo adicional das fases de comunicação. Mas o aumento do número de processadores efetivamente reduz o tempo de execução, sendo que o desempenho do algoritmo paralelo em 4 processadores só não consegue superar o algoritmo sequencial no “Pior Caso”, justamente por ser este o caso mais desfavorável à paralelização.

A figura 8 indica mais claramente a variação do tempo de execução em função de p . O algoritmo paralelo possui um comportamento extremamente favorável, com um tempo de execução variando em relação ao número de processadores de forma quase inversamente proporcional. Este comportamento só não ocorre para valores menores de n (em geral inferiores a 65536) e maiores de p (principalmente para 8 e 16), em que o aumento do número de processadores em alguns casos leva a um aumento no tempo de execução do algoritmo. Isto ocorre porque um aumento no número de processadores implica em um custo maior de comunicação e até mesmo em um aumento no número de rodadas (do passo 2), e embora estes custos não dependam diretamente de n , sua influência no tempo total é mais sensível no caso de problemas menores (que envolvem tempos totais menores). Mas para valores suficientemente grandes de n , estes custos adicionais não são significativos em relação ao tempo total.

Os gráficos da figura 9 indicam a obtenção de um *speedup* praticamente linear para valores suficientemente grandes de n . O “Caso Médio” apresenta o melhor *speedup*, superando inclusive o do “Caso Especial”, que requer tempos de execução menores tanto na execução paralela como na sequencial. Considerando-se que o “Caso Médio” representa justamente o caso mais comum de aplicação prática do algoritmo, o *speedup* de 7.28 obtido para 16 processadores e 1M elementos é um resultado excelente, que comprova tratar-se de um algoritmo de real utilização prática. Mesmo no “Pior Caso”, o algoritmo apresenta nestas condições um *speedup* acima de 2 em relação a um algoritmo sequencial determinístico otimizado.

Finalmente a figura 10 apresenta o comportamento do algoritmo em relação à variação de n . Para valores maiores de n , o processamento paralelo com o maior número de processadores corresponde a tempos de execução menores. A partir destes valores, uma redução em n corresponde inicialmente a uma redução proporcional no tempo de execução do algoritmo, pois a maior parte do tempo de execução deve-se às fases de processamento local, com tempo de execução diretamente proporcional a n . Entretanto quando o tamanho do problema torna-se inferior a cerca de 10000 elementos, a redução de n deixa de proporcionar uma redução tão significativa no tempo de execução do algoritmo, pois neste ponto a maior parte do tempo de execução passa a corresponder às fases de comunicação, que apresentam uma dependência menor com n (apenas a fase final de comunicação depende do número de elementos restantes, enquanto que as fases de comunicação do passo 2 dependem exclusivamente de p). Para valores menores de n , as execuções que utilizam o maior número de processadores passam a corresponder aos maiores tempos de processamento, justamente por significarem tempos maiores de comunicação.

Pode-se ainda observar que o gráfico de “Caso Médio” apresenta curvas de tempo mais irregulares do que nos outros casos, devido ao caráter aleatório da distribuição de seus elementos. Os gráficos de “Pior Caso” e “Caso Especial” são os que apresentam as curvas mais regulares, nestes casos as variações de tempo nos testes devem-se aos atrasos de comunicação (pela utilização de mensagens assíncronas entre os processadores) e a uma pequena imprecisão das medidas de tempo (tipicamente o primeiro ponto de cada curva de $p=1$ apresenta uma maior irregularidade, pois o tempo medido nestes casos é muito pequeno).

7.1.6 Conclusões

Os testes realizados permitiram estudar, em uma situação prática, o desempenho do algoritmo paralelo de seleção com base na análise teórica apresentada anteriormente, comprovando-se o comportamento assintótico previsto e verificando-se a influência dos parâmetros n , p e c em seu funcionamento.

O algoritmo paralelo proposto apresenta excelentes resultados experimentais em todos os casos de teste, não apenas superando o tempo de execução do algoritmo sequencial (geralmente a partir de $p=4$), mas também apresentando um excelente *speedup* praticamente linear para n suficientemente grande. Estes resultados adquirem uma importância ainda maior ao considerar-se que a máquina paralela utilizada apresenta uma latência de comunicação considerável (da ordem

de milissegundos) e baixa taxa de transmissão de dados, e que o algoritmo de seleção sequencial determinístico é particularmente eficiente, possuindo complexidade linear $O(n)$, portanto com um desempenho difícil de ser superado por um algoritmo paralelo.

O algoritmo pode ser considerado escalável na medida em que o *speedup* cresce de forma praticamente linear com p , desde que sempre mantendo-se n consideravelmente maior do que p , de acordo com o conceito de escalabilidade do modelo CGM.

Um fator importante para a obtenção de um bom desempenho é o fato do algoritmo necessitar de uma comunicação muito reduzida a cada rodada, o que faz com que o custo de comunicação nestas rodadas (exceto na fase final) seja independente de n , dependente apenas de p e c . Esta preocupação adicional em reduzir o tempo de comunicação a cada rodada não é prevista no modelo CGM, que valoriza apenas a minimização do número de rodadas. Outro fator importante foi a introdução do parâmetro c , que permite diminuir o tamanho das mensagens transmitidas na fase final de comunicação às custas de um aumento no número de rodadas, ajustando assim o funcionamento do algoritmo às características de capacidade de processamento e comunicação da máquina paralela utilizada.

Por fim, cabe observar que o desempenho do algoritmo paralelo na configuração de entrada denominada “Pior Caso” poderia ser significativamente melhorado com uma alteração simples no algoritmo de forma que, na fase final, os elementos restantes seriam transferidos para aquele processador que já possuísse o maior número de elementos ao invés de sempre transferi-los para o processador 1. Este aperfeiçoamento não foi aqui adotado justamente para permitir verificar que o desempenho do algoritmo é adequado mesmo em condições muito desfavoráveis.

Outro aperfeiçoamento do algoritmo seria interromper o passo 2 se, ao final de uma rodada, todos os elementos restantes já estivessem armazenados em um único processador, o que representaria um aumento significativo do desempenho não apenas neste “Pior Caso”, mas também no “Pior Caso Randômico” e no “Caso Especial”.

7.2 Algoritmo de seleção simultânea

7.2.1 Descrição dos casos de teste

Para a análise do algoritmo, define-se novamente um conjunto de 4 casos de teste, com denominações semelhantes às empregadas anteriormente:

- “Caso Médio”: Os dados de entrada são distribuídos uniformemente (aleatoriamente) entre os processadores, com $k_j = j \cdot 2n / 3q$, $1 \leq j \leq q$.

- “Caso Especial”: Os dados de entrada já estão previamente ordenados (processador 1 com os menores elementos), com $k_j = j \cdot n / q$, $1 \leq j \leq q$. Neste caso, após $O(\log q)$ rodadas do algoritmo paralelo, o problema inicial relativo a uma sequência de n elementos é reduzido a q

problemas distintos, relativos a sub-sequências disjuntas de $O(n/q)$ elementos, distribuídos entre os processadores.

- “Pior Caso”: Os dados de entrada encontram-se em ordem decrescente (processador 1 com os maiores elementos), com $k_j = j \cdot n / pq$, $1 \leq j \leq q$. Novamente o termo “pior caso” refere-se a um caso extremamente desfavorável à paralelização, pois após as primeiras $O(\log p)$ rodadas do algoritmo paralelo, o processador p permanece com $O(n/p)$ elementos restantes (enquanto os demais processadores não mais possuem um número significativo de elementos restantes), e nas $O(\log q)$ rodadas seguintes ainda não há uma redução significativa no número de elementos restantes no processador p (que apenas dividem-se em $O(q)$ problemas relativos a sub-sequências disjuntas de elementos). Por outro lado, o fato dos elementos estarem previamente ordenados em cada processador facilita a execução do algoritmo sequencial.

- “Pior Caso Randômico”: Os dados de entrada encontram-se distribuídos em ordem decrescente apenas entre os processadores, mas distribuídos aleatoriamente em cada processador, com $k_j = j \cdot n / pq$, $1 \leq j \leq q$. Trata-se portanto de uma distribuição desfavorável tanto para a paralelização (como no “Pior Caso”) como também para o processamento local.

A operação sequencial de seleção emprega um algoritmo capaz de realizar seleções simultâneas de forma muito eficiente: a cada particionamento da sequência de elementos, os tamanhos das partições obtidas são comparados com os valores dos k_j , para já verificar se deve ocorrer a divisão deste problema de seleção simultânea em dois novos problemas sobre sub-sequências disjuntas de elementos.

A maior dificuldade para a implementação deste algoritmo é o fato da solução ótima para o problema da melhor redistribuição dos elementos restantes entre os processadores (realizada no passo 3) não poder ser calculada eficientemente, uma vez que trata-se de um problema NP-completo ([GJ-79]). Este problema é resolvido empregando-se um algoritmo baseado no método “guloso”: os tamanhos das sub-sequências (disjuntas) de elementos restantes são inicialmente ordenados em ordem decrescente, sendo cada sub-sequência atribuída ao processador que até o momento possua a menor quantidade de elementos alocados, com prioridade para o processador de menor índice. As soluções obtidas através deste algoritmo de alocação simples, embora não sejam obrigatoriamente ótimas, proporcionam uma alocação suficientemente boa para o funcionamento adequado do algoritmo⁶.

⁶ É fácil perceber que a quantidade de elementos alocados pelo algoritmo guloso para cada processador não pode ser maior do que duas vezes a quantidade máxima que seria alocada por um algoritmo ótimo e portanto, em relação à carga máxima dos processadores, a solução produzida por este algoritmo guloso difere da solução ótima apenas por fatores constantes.

7.2.2 Análise de desempenho em função do número de elementos n

A figura 11 apresenta gráficos do tempo de execução do algoritmo para a resolução de 16 problemas de seleção simultâneos. Em todos os casos de teste, o tempo de execução paralelo em 2 processadores ainda é maior do que o tempo sequencial, mas a partir de 4 processadores o tempo paralelo já passa a ser melhor do que o sequencial.

De acordo com estes gráficos, o “Caso Especial” e o “Pior Caso” apresentam tempos de execução muito próximos, apesar destes casos representarem situações opostas em termos de distribuição de carga. No “Caso Especial” ocorre a melhor distribuição de carga porque, após as primeiras rodadas do algoritmo paralelo, o problema inicial de seleção é transformado em vários problemas distintos, divididos quase igualmente entre os processadores. Já no “Pior Caso”, após as primeiras rodadas, quase todos os elementos dos diversos problemas encontram-se em um único processador.

Entretanto, mesmo com esta diferença na distribuição de carga, o custo total das operações realizadas nos dois casos é muito semelhante. Em relação à comunicação, a redistribuição dos elementos restantes entre os processadores, realizada no passo 3, poderia ser bem mais eficiente para o “Caso Especial”, pois neste caso os elementos restantes já estão distribuídos entre os processadores de acordo com os problemas a serem resolvidos, enquanto que no “Pior Caso” praticamente todos os elementos restantes encontram-se em um mesmo processador. Mas a implementação adotada não considera a distribuição dos elementos entre os processadores no instante da redistribuição dos problemas, exigindo que mesmo no “Caso Especial” a maior parte dos elementos restantes seja enviada para processadores diferentes. Assim, a quantidade total de elementos transmitidos no passo 3 é praticamente igual nos dois casos.

Quanto ao processamento local, em ambos os casos cada processador deve executar as mesmas rotinas de particionamento de sequências ordenadas de todos os seus elementos nas primeiras rodadas. Apenas nas rodadas seguintes a melhor distribuição de carga passa a favorecer o “Caso Especial”, mas então os tempos de particionamento envolvidos já são significativamente menores (devido ao menor número de elementos a serem particionados) e portanto este favorecimento já não interfere muito significativamente no tempo total. Observa-se assim que os tempos totais de execução do “Caso Especial” são apenas um pouco menores do que do “Pior Caso”.

A figura 12 apresenta novamente estes tempos, agora com o eixo de p em escala logarítmica, permitindo verificar que, para n suficientemente grande, o tempo de execução do algoritmo paralelo é quase inversamente proporcional a p . Este comportamento só não ocorre para valores menores de n e maiores de p , em que o aumento do número de processadores em alguns casos leva a um aumento no tempo de execução, porque nestes casos o custo maior de comunicação tem um peso também maior no tempo total de execução.

A figura 13 mostra um *speedup* progressivamente mais linear com o aumento de n em todos os casos de teste. O “Caso Médio” apresenta um *speedup* de 7.66 para 16 problemas e 1M

elementos em 16 processadores, um excelente resultado que comprova a aplicabilidade prática também deste algoritmo. Mesmo no “Pior Caso”, o *speedup* nestas condições é de 3.90, superando consideravelmente o *speedup* de “Pior Caso” do algoritmo paralelo básico de seleção.

Os valores de *speedup* deste algoritmo são muito superiores aos obtidos para o algoritmo básico de seleção, com exceção do “Caso Médio” que já apresentava anteriormente *speedups* elevados. O algoritmo de seleção simultânea é capaz de realizar uma paralelização mais eficiente porque a resolução simultânea de problemas possibilita uma melhor distribuição de carga entre os processadores, permitindo a atribuição de um sub-conjunto destes problemas para cada processador no último passo. No algoritmo básico de seleção, toda a execução do último passo fica a cargo de apenas um dos processadores, enquanto os demais permanecem ociosos.

A figura 14 contém os gráficos de tempo de execução em função de n , que apresentam um comportamento semelhante ao já observado no algoritmo básico de seleção: os tempos de execução diminuem com o aumento do número de processadores para valores suficientemente grandes de n , mas para valores menores de n observa-se que um número maior de processadores implica em um tempo de execução também maior.

7.2.3 *Análise de desempenho em função do número de problemas q*

Na figura 15 observa-se a variação do tempo de execução em função do número de processadores para diversos valores de número de problemas (sobre a mesma sequência de n elementos) a serem resolvidos simultaneamente. No gráfico do “Caso Médio” é possível observar que, para a resolução de um número suficientemente grande de problemas simultâneos (a partir de $q=32$), o desempenho do algoritmo paralelo mesmo em 2 processadores consegue superar o do algoritmo sequencial, uma vez que valores maiores de q permitem uma melhor distribuição de carga entre os processadores no passo final do algoritmo paralelo. Observa-se também que um número maior de problemas simultâneos corresponde evidentemente a um aumento no tempo de execução, porém este aumento não é muito significativo.

Para $q=1$, este algoritmo equivale exatamente ao algoritmo básico de seleção analisado anteriormente. Comparando-se as curvas para $q=1$ dos gráficos desta figura 15 com as curvas para $n=1048576$ dos gráficos da figura 7, observa-se que o algoritmo de seleção simultânea requer tempos de execução maiores para resolver um mesmo problema. O desempenho deste algoritmo é inferior porque, embora sejam executadas praticamente as mesmas operações do algoritmo básico nestes casos, a generalização do algoritmo requer a utilização de rotinas um pouco menos eficientes com estruturas de controle mais complexas.

Os gráficos da figura 16 indicam que o tempo de execução obtido foi quase inversamente proporcional ao número de processadores, qualquer que seja o número de problemas a serem resolvidos simultaneamente.

A figura 17 permite observar mais detalhadamente este aspecto, revelando que um número maior de problemas simultâneos corresponde a um *speedup* maior e ainda mais linear, mesmo

considerando-se que para $q=1$ o *speedup* obtido já apresenta uma grande linearidade. Em particular, o gráfico de “Caso Médio” indica para 16 processadores um *speedup* sempre acima de 7 para qualquer q (exceto $q=1$), novamente provando tratar-se de um algoritmo com real aplicação prática. No gráfico de “Pior Caso” observa-se que o *speedup* melhora sensivelmente com o aumento de q , sendo próximo a 2 para $q=1$ e próximo a 4 para $q=128$ na execução com 16 processadores. Estas melhoras no *speedup* ocorrem porque valores maiores de q em relação a p favorecem o algoritmo paralelo por possibilitarem uma melhor distribuição de carga entre os processadores, principalmente no último passo, no qual cada processador deve receber um conjunto de problemas a serem resolvidos sequencialmente.

A partir dos gráficos apresentados na figura 18, verifica-se que o tempo de execução do algoritmo cresce logaritmicamente em função de q , conforme previsto. No “Caso Médio” observa-se, mais uma vez, que o algoritmo paralelo em 2 processadores supera o desempenho do algoritmo sequencial a partir de $q=32$.

7.2.4 Conclusões

Novamente são obtidos resultados experimentais que comprovam o funcionamento adequado do algoritmo paralelo proposto, com a obtenção de um *speedup* considerável, praticamente linear em p em todos os casos, para n suficientemente grande e qualquer q .

Os tempos de execução do algoritmo paralelo no “Caso Especial” poderiam ser consideravelmente menores se o passo 3 do algoritmo paralelo fosse aperfeiçoado, fazendo-se com que a lógica de alocação dos problemas entre os processadores considerasse a distribuição dos elementos restantes ao final do passo 2 e procurasse assim minimizar o número de elementos restantes a serem transmitidos para processadores diferentes. Entretanto optou-se por manter o algoritmo original de alocação para permitir estudar o comportamento do algoritmo neste caso, verificando-se que mesmo sem este aperfeiçoamento foi obtido um bom desempenho.

7.3 Algoritmo de ordenação

7.3.1 Descrição dos casos de teste

Os casos de teste para este novo algoritmo são assim definidos:

- “Caso Médio”: Os dados de entrada são distribuídos uniformemente (aleatoriamente) entre os processadores.
- “Caso Especial”: Os dados de entrada já estão previamente ordenados (processador 1 com os menores elementos), o que elimina a necessidade de comunicação no passo 10 do algoritmo.

- “Pior Caso”: Os dados de entrada encontram-se em ordem decrescente (processador 1 com os maiores elementos), portanto todos os elementos são transmitidos para processadores diferentes no passo 10 do algoritmo.

- “Pior Caso Randômico”: Os dados de entrada encontram-se distribuídos em ordem decrescente apenas entre os processadores, mas distribuídos aleatoriamente em cada processador. Assim, todos os elementos também são transmitidos para processadores diferentes no passo 10 do algoritmo.

O algoritmo “merge sort” foi escolhido para ser empregado no passo 11 do algoritmo paralelo e como algoritmo sequencial de ordenação, por tratar-se do algoritmo de ordenação determinístico (tempo $O(n \cdot \log n)$ no pior caso) mais eficiente, de acordo com [Wi-89].

7.3.2 Análise de desempenho

A figura 19 apresenta o tempo de execução do algoritmo em cada caso. O gráfico do “Caso Médio” apresenta um desempenho melhor do algoritmo paralelo em relação ao sequencial mesmo para apenas 2 processadores, exceto para o caso $n=1048576$ (discutido adiante). No gráfico do “Caso Especial”, o desempenho do algoritmo paralelo supera o do sequencial em todas as condições. Nos demais gráficos de “Pior Caso” e “Pior Caso Randômico”, o algoritmo paralelo supera o sequencial a partir de 4 processadores.

Entretanto, na maioria dos gráficos é possível notar uma distorção no ponto da curva correspondente a $p=2$ e valor máximo de n , que apresenta um tempo de execução do algoritmo paralelo excepcionalmente alto.

A figura 20 mostra novamente o tempo de execução do algoritmo em cada caso. Excluindo-se os pontos de distorção já identificados, observa-se um tempo de execução quase inversamente proporcional a p , com exceção das regiões dos gráficos correspondentes a valores menores de n e maiores de p , portanto um comportamento similar ao identificado nos algoritmos anteriores.

A figura 21 representa a quantidade de elementos transmitidos por cada processador durante o passo 10 do algoritmo, em que os n elementos devem ser realocados entre os processadores. Esta figura indica que os pontos de distorção observado nos gráficos anteriores correspondem exatamente às condições em que ocorre a transmissão de mensagens maiores entre os processadores, o que nos permite estimar que tal distorção ocorra devido a limitações da máquina paralela quanto ao tamanho máximo das mensagens (mensagens de tamanho superior à capacidade de armazenamento do processador auxiliar de comunicação exigiriam que o processador de aplicação aguardasse a transmissão da mensagem, atrasando consideravelmente o processamento). Os algoritmos paralelos de seleção anteriores não apresentam tais distorções porque seus passos de comunicação envolvem quantidades muito menores de elementos, uma vez que realizam a transmissão apenas dos elementos restantes após diversas rodadas do algoritmo, enquanto no algoritmo paralelo de ordenação pode ser realizada a transmissão de todos os elementos no passo final de comunicação.

Estes gráficos comprovam que o “Pior Caso” e o “Pior Caso Randômico” demandam um custo de comunicação muito maior, principalmente para $p=2$, quando ocorre a transmissão de aproximadamente o dobro de elementos em relação ao “Caso Médio”. Verifica-se ainda que no “Caso Especial” realmente não há transmissão de elementos neste passo.

A figura 22 mostra um *speedup* progressivamente mais linear com o aumento de n em todos os casos de teste, observando-se ainda curvas distorcidas para o caso $n=1048576$, correspondentes às distorções dos gráficos anteriores. O “Caso Médio” apresenta um *speedup* de 5.15 para n máximo e 16 processadores, o que ainda representa um bom desempenho do algoritmo paralelo, embora inferior ao *speedup* superior a 7 obtido pelos algoritmos de seleção paralelos nestas condições. Este *speedup* menor deve-se principalmente ao alto custo de comunicação, que requer a transmissão entre os processadores de uma parte considerável dos n elementos. Por outro lado, todos os casos apresentam um *speedup* superior a 3.5 em 16 processadores nestas condições, pois embora o custo de comunicação seja significativamente maior neste algoritmo (em relação aos algoritmos de seleção), o processamento em paralelo reduz o número de elementos a serem ordenados sequencialmente por cada processador no passo 11 do algoritmo paralelo, o que representa uma diminuição muito grande no tempo total de execução.

O “Caso Especial” apresentou os melhores valores de *speedup*, uma vez que este caso favorece consideravelmente o processamento paralelo por não haver transmissão de dados no passo 10, o que normalmente representaria a maior parte do custo em comunicação da execução do algoritmo.

A figura 23 apresenta as curvas de tempo total de execução em função de n . Novamente obtém-se um comportamento semelhante ao observado nos algoritmos de seleção anteriores.

7.3.3 Análise de tempos parciais

Por fim, para comprovar a viabilidade de utilizar o algoritmo de seleção simultânea para resolver o problema da ordenação, recorre-se novamente à análise de tempos parciais.

Na figura 24 são apresentados os tempos parciais de execução do algoritmo paralelo de ordenação para diversos casos. Esta análise identifica os tempos gastos pelos passos 1 a 4 (execução do algoritmo de seleção simultânea), passos 5 a 8 (decisão de como dividir os elementos entre os processadores), passos 9 e 10 (realocação dos elementos entre os processadores) e passo 11 (ordenação sequencial dos elementos em cada processador). Para a obtenção dos tempos parciais, o algoritmo foi executado apenas uma vez para cada teste.

De acordo com estes gráficos, um número maior de processadores implica em um aumento relativo no tempo parcial dos passos da seleção simultânea. Por outro lado, para um número fixo de processadores, um aumento no valor de n significa uma diminuição relativa neste tempo parcial, sendo esta diminuição mais facilmente observada no caso de um número maior de processadores. Este comportamento é justificado pela análise teórica dos algoritmos, pois nestas condições o

termo dominante no tempo de execução do algoritmo de seleção simultânea é $O((n/p) \cdot \log p)$ e o tempo de execução do algoritmo de ordenação local é $O((n/p) \cdot \log(n/p))$.

Observa-se que, para valores de n suficientemente grandes em relação a p , os passos correspondentes à realocação dos elementos e ordenação sequencial passam a representar a maior parcela do tempo de execução do algoritmo, superando o tempo gasto na execução dos passos de seleção simultânea.

O tempo parcial dos passos adicionais que decidem como realocar os elementos entre os processadores é muito pequeno em todos os casos. O tempo destes passos tem alguma relevância (em relação ao tempo total) apenas para valores pequenos de n e grandes de p , situação em que todos os tempos envolvidos são muito menores.

Os gráficos permitem ainda comprovar que, no “Caso Especial”, não há tempo gasto na realocação de elementos, uma vez que estes elementos já estão inicialmente distribuídos de forma ordenada entre os processadores.

7.3.4 Conclusões

O algoritmo de ordenação paralelo proposto, da mesma forma como os algoritmos de seleção, apresenta resultados experimentais muito bons, com a obtenção de um *speedup* elevado e praticamente linear em p , em todos os casos, para n suficientemente grande.

Existem na literatura outros algoritmos paralelos determinísticos mais simples para o problema da ordenação. Em particular, estima-se que um algoritmo de ordenação baseado em amostragem determinística (como apresentado em [GS-96]) apresente um desempenho um pouco melhor do que o novo algoritmo de ordenação aqui proposto. Entretanto, cabe ressaltar que os passos de realocação dos elementos e de ordenação sequencial local representam a maior parte do tempo de execução deste algoritmo (para n suficientemente grande em relação a p) e a realização de operações equivalentes a estes passos é sempre necessária para resolver o problema, portanto não é possível a obtenção de desempenhos muito melhores com qualquer outro algoritmo paralelo de ordenação para memória distribuída.

Finalmente deve-se observar que desta vez o funcionamento deste algoritmo não pode ser aperfeiçoado significativamente quanto à configuração de entrada de “Pior Caso”, uma vez que o alto custo de transmissão de elementos que ocorre no passo 10 é inerente ao problema e não ao algoritmo. Para valores maiores de p até mesmo o “Caso Médio” requer que a maior parte dos elementos sejam efetivamente transmitidos para processadores diferentes, porém com o tamanho de cada mensagem decrescendo com o aumento de p .

8. Implementação em Meiko CS-2 (MPI)

Os algoritmos propostos foram a seguir testados na máquina paralela “*Meiko Computing Surface-2*”⁷. Este computador possui 10 nós (um dos quais encontra-se atualmente desativado), dispostos de acordo com a topologia “árvore gorda” (“*fat tree*”), sendo que cada nó possui um processador principal *Superscalar SPARC* de 100 MHz, com memória local de 128 Mbytes, dois processadores auxiliares *Fujitsu Vector Processor* de 50 MHz e um processador auxiliar de comunicação *Meiko Elan Communications Processor*.

Ao invés de simplesmente realizar estes novos testes utilizando a mesma implementação anterior para a interface de comunicação PVM, optou-se por adaptar os algoritmos para a interface de comunicação MPI (*Message Passing Interface* [MPIF-94], [GL-94]), escolhida por ser a interface de comunicação padrão de maior aceitação atualmente. No caso da máquina paralela “*Parsytec PowerXplorer*” esta escolha não havia sido possível porque a interface MPI não estava disponível para aquela máquina.

As duas máquinas paralelas testadas diferem não apenas em relação à interface de comunicação utilizada, mas também quanto a topologia de rede, tipo de processador, capacidade de memória e velocidade de comunicação, não sendo possível portanto comparar isoladamente os valores individuais de desempenho obtidos. Mas a análise do comportamento dos algoritmos a partir de testes realizados em máquinas tão diferentes permite justamente verificar o grau de independência destes algoritmos em relação a características específicas da arquitetura da máquina paralela utilizada.

A execução destes novos testes segue exatamente os mesmos procedimentos adotados anteriormente, mas com cada teste de desempenho sendo executado uma única vez. Optou-se por não realizar repetições das medições como uma forma de restringir o tempo total de utilização da máquina paralela, considerando-se que cada execução individual é bem mais demorada (pois o tamanho máximo dos problemas a serem testados é muito maior devido à maior capacidade de memória de cada processador) e a disponibilidade desta máquina é mais limitada (por pertencer a outro instituto). Entretanto, esta alteração não afeta significativamente os resultados obtidos, pois a variação observada nas medidas é de apenas 2% em média.

⁷ Equipamento pertencente ao Laboratório de Sistemas Integráveis da Escola Politécnica da USP, que gentilmente permitiu sua utilização para este trabalho.

Assim, os gráficos de execução apresentam simplesmente os tempos obtidos e não mais os piores tempos. Como anteriormente, os tempos são medidos em “ticks”, sendo que no relógio desta máquina cada “tick” também equivale a 10^{-6} segundos.

No apêndice A encontram-se as listagens dos algoritmos implementados. A maioria dos arquivos listados é comum às duas interfaces, já que as rotinas específicas para PVM e MPI foram isoladas em arquivos separados. As figuras 25 a 48 do apêndice B contêm os gráficos com os resultados obtidos para a interface MPI na máquina paralela “Meiko CS-2”, que correspondem exatamente aos mesmos testes apresentados anteriormente nas figuras 1 a 24.

8.1 Algoritmo básico de seleção

8.1.1 Análise do parâmetro c

A figura 25 apresenta os tempos de execução obtidos em função do parâmetro c em cada caso para um total de 8M elementos, que corresponde à maior capacidade de armazenamento em memória de cada processador. Assim como na figura 1, o tempo de execução paralelo diminui com o aumento do número de processadores, já apresentando um desempenho melhor do que o algoritmo sequencial a partir de 4 processadores (exceto no “Pior Caso”).

Para valores pequenos do parâmetro c , o seu aumento corresponde a uma diminuição do tempo de execução do algoritmo apenas no “Caso Médio” e a um aumento deste tempo de execução nos demais casos. Nos testes anteriores, havia sido observada uma diminuição no tempo de execução na maioria dos casos estudados e esta diminuição era bem acentuada no “Caso Médio” (principalmente para $p=2$). Esta diferença de comportamento ocorre porque a máquina paralela “Meiko CS-2” é capaz de realizar operações de comunicação com taxas de transferência muito mais altas, o que possibilita a transmissão de grandes quantidades de informações entre os processadores em um tempo muito menor, portanto o tempo de execução do passo 3 do algoritmo não é tão significativo em relação ao tempo total. Assim, com o aumento do parâmetro c , na maioria dos casos a redução no tempo de execução dos passos 3 e 4 (devido à diminuição no número de elementos restantes) não é grande o suficiente para compensar o aumento no tempo de execução do passo 2 (devido ao aumento no número de rodadas).

Nesta figura é possível notar irregularidades nas curvas dos gráficos do “Caso Médio” e principalmente do “Pior Caso Randômico”, por serem os dois casos correspondentes a uma distribuição aleatória dos dados de entrada.

Os gráficos da figura 26 confirmam que o número de elementos restantes é inversamente proporcional a c e também diminui com o aumento de p . Para valores suficientemente grandes de c , ocorrem algumas interrupções nos gráficos de “Caso Médio” e “Pior Caso Randômico”, que correspondem às situações em que a solução do problema é determinada já durante o passo 2 do algoritmo. Os gráficos de “Caso Especial” e “Pior Caso” não apresentam nenhuma interrupção

porque a distribuição dos dados de entrada nestes casos nunca permite a determinação desta solução durante o passo 2.

A figura 27 indica que o número de rodadas é diretamente proporcional ao logaritmo de c e também aumenta com p . Em algumas curvas, o número de rodadas pára de aumentar para valores muito grandes de c , correspondendo às situações da figura anterior em que o algoritmo é interrompido, ou seja, o número de rodadas é suficiente para a determinação da solução durante o passo 2.

8.1.2 Análise de tempos parciais

A análise de tempos parciais apresentada nas figuras 28 e 29 permite comprovar que o passo 3 do algoritmo corresponde a uma parcela muito pequena do tempo de execução, sendo perceptível apenas para $c=1$. Em comparação com as figuras 4 e 5, tomando-se como exemplo o “Caso Médio” com $c=1$ utilizando 2 processadores e número máximo de elementos, observa-se que na máquina paralela “Meiko CS-2” o passo 3 representa agora menos de 2% do tempo total de execução, enquanto que na máquina paralela “Parsytec PowerXplorer” este passo representava sozinho cerca de 60% do tempo total.

Observa-se também que a execução de um algoritmo de tempo linear com os elementos restantes no passo 4 requer um tempo consideravelmente maior do que a transmissão destes mesmos elementos restantes do passo 3, ao contrário do que ocorria com a máquina paralela anterior.

Estas observações mostram que esta máquina é capaz de transmitir uma grande quantidade de elementos entre os processadores a uma velocidade muito elevada. Mesmo assim, no “Caso Médio” o aumento de c causa uma diminuição no tempo total de execução do algoritmo porque, embora a redução no tempo parcial do passo 3 não seja significativa, neste caso a redução apenas do passo 4 (devido à diminuição no número de elementos restantes) já é suficiente para compensar o aumento do passo 2 (devido ao aumento no número de rodadas). Nos demais casos, o passo 4 não tem um peso tão grande porque o número de elementos restantes nestes casos é menor, por isso sua redução é superada pelo aumento do passo 2.

8.1.3 Determinação do parâmetro c

A figura 30 apresenta novos gráficos de tempo de execução em função do parâmetro c . Observa-se novamente que, para valores de c até 10, o seu aumento corresponde a uma diminuição considerável no tempo de execução no “Caso Médio” (sendo esta diminuição mais acentuada para valores menores de n e maiores de p) e a um aumento no tempo de execução nos demais casos. Para valores maiores de c , em geral acima de 100, o seu aumento implica em um pequeno aumento no tempo de execução.

Considerando-se que as curvas do “Caso Médio” são as mais afetadas por valores pequenos de c , o valor $c=100$ foi novamente escolhido como o mais adequado para esta máquina paralela por situar-se, de acordo com os gráficos, entre a região que apresenta a queda mais acentuada nestas curvas e a região restante que apresenta um pequeno aumento em todas as curvas.

8.1.4 Análise de desempenho

De acordo com os gráficos da figura 31, o aumento do número de processadores possibilita uma redução no tempo de execução do algoritmo paralelo, tornando-o menor do que o tempo sequencial a partir de 4 processadores para a maioria dos casos de teste.

Verifica-se também que esta máquina paralela possui uma velocidade de processamento local bem superior à outra máquina utilizada anteriormente. A figura 7 indicava um tempo de execução de 5623957 “ticks” para o algoritmo sequencial no “Caso Médio” com 1048576 elementos, enquanto que a figura 31 indica 2816755 “ticks” nas mesmas condições, portanto uma velocidade quase exatamente 2 vezes maior.

A figura 32 mostra que o tempo de execução do algoritmo paralelo é inversamente proporcional ao número de processadores, exceto para valores menores de n e maiores de p , como observado anteriormente.

Os gráficos da figura 33 indicam que o *speedup* em cada caso é praticamente linear para valores suficientemente grandes de n . Comparando-se estes gráficos com os da figura 9, verifica-se que em todos os casos os novos valores de *speedup* são um pouco maiores do que os obtidos anteriormente para os mesmos valores de n e p , porque as operações de comunicação nesta máquina paralela são muito mais eficientes, mas o fato desta diferença no *speedup* ser muito pequena comprova que o desempenho deste algoritmo não depende muito das características de comunicação da máquina paralela utilizada.

A figura 34 mostra novamente que, para valores suficientemente grandes de n , um número maior de processadores corresponde a um tempo de execução menor, enquanto que para valores pequenos de n um número maior de processadores implica em um tempo de execução maior.

8.1.5 Conclusões

Os novos resultados obtidos experimentalmente para o algoritmo básico de seleção são muito semelhantes aos dos testes anteriores, principalmente em relação aos gráficos de *speedup*, que apresentam valores muito próximos nas duas máquinas paralelas em todos os casos de teste. Estes resultados comprovam novamente o excelente desempenho do algoritmo e em particular a sua escalabilidade em relação ao número de processadores para um tamanho de problema suficientemente grande.

Como a máquina “Meiko CS-2” possui uma taxa de transferência de dados muito elevada (mesmo considerando-se sua maior velocidade de processamento local), o fato do algoritmo ter

sido projetado de forma a minimizar o tamanho das mensagens transmitidas não é tão importante neste caso. No entanto, como ainda há um tempo de latência razoável na transmissão de cada mensagem, o esforço em reduzir o número de rodadas de comunicação e a quantidade de mensagens transmitidas permanece como um fator fundamental para garantir o bom desempenho deste algoritmo.

8.2 Algoritmo de seleção simultânea

8.2.1 *Análise de desempenho em função do número de elementos n*

A figura 35 apresenta os gráficos de tempos de execução para a resolução simultânea de 16 problemas de seleção. Como o algoritmo de seleção simultânea requer uma maior quantidade de memória local devido às suas estruturas de dados mais complexas, o tamanho máximo da sequência de entrada limita-se a 4M elementos.

Estes gráficos indicam novamente que, com a utilização de 4 processadores, a velocidade de execução do algoritmo paralelo é maior que a do sequencial em todos os casos. Mas no “Caso Médio” e “Pior Caso Randômico” a velocidade do algoritmo paralelo já é muito próxima à do sequencial mesmo para 2 processadores, chegando até mesmo a superá-la em algumas situações. O algoritmo paralelo apresenta um melhor desempenho porque, nestes casos, o processamento local é responsável por uma parcela ainda maior do tempo total, já que o tempo necessário para a realização das operações de particionamento é muito maior (devido à distribuição aleatória dos elementos em cada processador e ao maior número de elementos) e as operações de comunicação nesta máquina paralela são muito mais eficientes. Em relação ao algoritmo básico de seleção, a seleção simultânea apresenta ainda a vantagem de permitir uma melhor distribuição de carga no último passo da execução.

A figura 36 mostra tempos de execução inversamente proporcionais ao número de processadores para valores de n suficientemente grandes.

A figura 37 indica que o *speedup* é linear em relação a p para valores maiores de n , comprovando a escalabilidade do algoritmo. Os valores obtidos são um pouco maiores do que os apresentados na figura 13 para os mesmos valores de n e p devido à grande diferença em relação à taxa de transmissão de dados entre as duas máquinas paralelas testadas. Também são maiores do que os do algoritmo básico de seleção porque o problema de seleção simultânea permite uma melhor distribuição de carga entre os processadores.

A figura 38 apresenta, mais uma vez, gráficos de tempo de execução em função do número de elementos.

8.2.2 Análise de desempenho em função do número de problemas q

A figura 39 contém gráficos do tempo de execução do algoritmo em função do número de processadores para diferentes números de problemas simultâneos q . De acordo com estes gráficos, o aumento de q corresponde a um aumento mais significativo no tempo de execução do algoritmo sequencial do que do paralelo porque, embora aumentem os tempos de processamento local dos dois algoritmos, valores maiores de q favorecem o algoritmo paralelo por permitirem uma melhor distribuição de carga entre os processadores. Por isso, no “Caso Médio” e “Pior Caso Randômico” o desempenho do algoritmo paralelo em dois processadores chega a igualar o do sequencial para valores maiores de q .

A figura 40 comprova que o tempo de execução é inversamente proporcional ao número de processadores para qualquer valor de q .

Na figura 41, observa-se que o *speedup* obtido aumenta linearmente em relação ao número de processadores, sendo ainda mais linear para valores maiores de q . Os novos valores de *speedup* são um pouco maiores do que os da figura 17 porque a taxa de transmissão desta máquina paralela é bem maior e porque o número de elementos utilizados é também maior.

A figura 42 mostra que o tempo de execução do algoritmo paralelo é proporcional ao logaritmo de p . Também comprova-se que no “Caso Médio” e “Pior Caso Randômico” o algoritmo paralelo para $p=2$ equipara-se ao sequencial para valores maiores de q .

8.2.3 Conclusões

Os resultados experimentais confirmam novamente o comportamento assintótico do algoritmo de seleção simultânea, com a obtenção de valores de *speedup* praticamente lineares em relação ao número de processadores para valores suficientemente grandes de n .

8.3 Algoritmo de ordenação

8.3.1 Análise de desempenho

A figura 43 apresenta os tempos de execução do algoritmo de ordenação, mostrando que o algoritmo sequencial é superado pelo paralelo mesmo para 2 processadores em todos os casos de teste. Como o algoritmo paralelo de ordenação precisa realizar uma realocação dos elementos entre os processadores, a alta taxa de transferência desta máquina paralela favorece muito o desempenho deste algoritmo. Nos algoritmos paralelos de seleção, a quantidade de dados a serem transmitidos entre os processadores era muito menor e por isso essa taxa de transferência não exercia um efeito tão significativo. Além disso, o algoritmo sequencial de ordenação possui uma complexidade bem maior do que o de seleção, com o tempo de execução aumentando mais do que linearmente em relação a n , o que torna mais eficiente o seu processamento em paralelo.

A figura 44 mostra que o tempo de execução do algoritmo é inversamente proporcional ao número de processadores.

A figura 45 comprova que o “Pior Caso” e “Pior Caso Randômico” correspondem à maior quantidade de elementos transmitidos no passo 10 de realocação dos elementos entre os processadores, enquanto que no “Caso Especial” nenhum elemento é transmitido neste passo.

Os gráficos da figura 46 mostram que o *speedup* é praticamente linear em relação a p para n suficientemente grande. Em relação à figura 22, os novos valores de *speedup* são cerca de 2 vezes maiores do que os obtidos anteriormente na maior parte dos casos devido à taxa de transferência de dados muito maior desta máquina paralela, que permite um desempenho muito melhor do algoritmo paralelo de ordenação, principalmente quanto à realocação de elementos no passo 10. Apenas o “Caso Especial” apresenta valores de *speedup* mais próximos nas duas máquinas paralelas porque neste caso não há realocação de elementos entre os processadores no passo 10, por isso neste caso a comunicação bem mais eficiente da máquina “Meiko CS-2” não representa uma grande vantagem.

Em todos os casos, os valores de *speedup* deste algoritmo são próximos a 5 com a utilização de 8 processadores, superando os valores de *speedup* dos algoritmos de seleção. A obtenção de valores tão elevados ocorre porque, como a taxa de transferência desta máquina paralela é bem maior, a maior parte do tempo de execução do algoritmo é gasto apenas no passo 11 de ordenação sequencial, cujo tempo de execução é muito reduzido com o aumento do número de processadores.

A figura 47 indica novamente que, para valores elevados de n , o algoritmo paralelo apresenta o comportamento assintótico previsto, com o tempo de execução aumentando proporcionalmente a n e sendo menor para valores maiores de p . Para valores pequenos de n , o tempo de comunicação passa a constituir uma parcela muito grande do tempo total, o que faz com que este tempo não apresente uma dependência tão grande em relação a n e torne-se maior para valores maiores de p .

8.3.2 Análise de tempos parciais

A análise de tempos parciais apresentada na figura 48 indica que os passos correspondentes à realocação de elementos representa uma parcela muito pequena do tempo total de execução do algoritmo de ordenação devido à alta taxa de transferência desta máquina paralela. Mas mesmo no “Caso Especial”, em que não há transferência de elementos entre os processadores, esta parcela de tempo ainda é perceptível (embora pequena) porque, nesta implementação, todos os elementos locais de cada processador precisam ser movidos para uma área temporária de memória antes da realocação para garantir uma utilização mais eficiente das primitivas de comunicação da interface MPI. Com isso, mesmo que nenhum elemento seja efetivamente transmitido entre os processadores, estes passos exigem algum tempo de processamento local. Neste caso específico, o desempenho do algoritmo poderia ser melhorado verificando-se previamente se há algum elemento a ser transmitido para outro processador antes de utilizar esta área temporária de memória, mas mesmo

sem este aperfeiçoamento o “Caso Especial” já apresenta um *speedup* muito alto e que supera o obtido nos demais casos.

Novamente a parcela de tempo correspondente aos passos de seleção simultânea é mais significativa em relação ao tempo total para valores maiores de p e menos significativa para valores maiores de n . Mas em todos os casos o passo de ordenação sequencial constitui a maior parcela de tempo de execução do algoritmo paralelo de ordenação, e como este passo apresenta complexidade $O((n/p) \cdot \log(n/p))$, a utilização de um número maior de processadores proporciona uma melhora muito grande no desempenho do algoritmo.

8.3.3 Conclusões

Os resultados obtidos atestam mais uma vez a aplicabilidade prática dos algoritmos propostos, com a obtenção de valores de *speedup* muito elevados.

O algoritmo paralelo de ordenação apresenta um desempenho muito melhor em uma máquina paralela com alta taxa de transferência entre os processadores, ao contrário dos algoritmos paralelos de seleção nos quais esta característica não influi de forma muito significativa. Mesmo com uma grande redução no tempo parcial dos passos de realocação de elementos, o passo de ordenação sequencial sozinho ainda requer um tempo de execução maior do que a seleção simultânea, portanto mesmo que o desempenho deste algoritmo de ordenação baseado em seleção simultânea seja superado por algum outro algoritmo paralelo de ordenação para memória distribuída, o aumento no desempenho estará limitado a pequenos fatores constantes.

9. Considerações Finais

Embora diversos algoritmos tenham sido apresentados abordando o problema da seleção para máquinas paralelas com memória compartilhada, existem poucos trabalhos publicados acerca deste problema para memória distribuída. Com base no modelo CGM (*Coarse Grained Multicomputer*), apresenta-se aqui, para o problema da seleção, um novo algoritmo paralelo determinístico para máquinas com memória distribuída que requer apenas $O(\log p)$ rodadas de comunicação e $O(p)$ dados transmitidos a cada rodada exceto na última. Após um extenso trabalho de pesquisa bibliográfica, acredita-se ser este o melhor algoritmo com tais características em comparação com os disponíveis na literatura.

A partir deste algoritmo básico de seleção, é desenvolvido um segundo algoritmo paralelo determinístico para a seleção simultânea de q problemas relativos a uma mesma sequência de elementos. Para $q=O(p)$, este algoritmo requer também $O(\log p)$ rodadas de comunicação e assintoticamente mesmo tempo de computação local do algoritmo básico, apenas com a transmissão de uma quantidade até q vezes maior de dados por rodada.

Por sua vez, o algoritmo de seleção simultânea permite o desenvolvimento de um algoritmo de ordenação paralelo determinístico para memória distribuída, que requer $O(\log p)$ rodadas de comunicação e $O(p^2)$ dados transmitidos a cada rodada exceto na última.

Os novos algoritmos propostos são implementados e testados em duas máquinas paralelas com memória distribuída de características bem diferentes. Para um tamanho de problema suficientemente grande, são obtidos desempenhos muito bons e *speedups* elevados em ambas as máquinas, o que atesta a eficiência, portabilidade e real possibilidade de aplicação prática destes algoritmos paralelos. A obtenção de *speedups* elevados deve ser ainda mais valorizada considerando-se que o algoritmo sequencial de seleção utilizado para comparação é extremamente eficiente, possuindo complexidade de tempo linear e tendo sido ainda aperfeiçoado para ganhos marginais no desempenho. Os resultados experimentais mostram também *speedups* praticamente lineares em relação ao número de processadores, comprovando assim a escalabilidade dos algoritmos.

Os dois algoritmos de seleção propostos apresentam *speedups* muito próximos mesmo em máquinas paralelas com velocidades de comunicação muito diferentes, indicando que estes algoritmos são bem sucedidos no sentido de manterem uma independência em relação às características particulares de comunicação da máquina paralela utilizada. Já os *speedups* do novo algoritmo de ordenação são bastante influenciados por esta diferença na velocidade de

comunicação, mas como neste caso a maior parte do tempo de execução é gasto em operações impostas pelo próprio problema e não pelo algoritmo utilizado, esta influência é inevitável e portanto o desempenho obtido não pode ser significativamente melhorado por qualquer outro algoritmo.

A obtenção de resultados práticos tão bem sucedidos deve-se aos seguintes fatores:

- A decisão de projetar os algoritmos a partir do modelo CGM e sua estratégia de minimização do número de rodadas de comunicação, o qual mostrou realmente representar de forma muito próxima a arquitetura das máquinas paralelas utilizadas.
- A preocupação adicional em também reduzir a quantidade de informações transmitidas a cada rodada de comunicação (mesmo às custas de um desbalanceamento de carga entre os processadores), que embora não seja prevista pelo modelo CGM, mostrou-se muito importante principalmente para uma máquina paralela com menor velocidade de comunicação.
- A introdução de mais um parâmetro c para ajustar experimentalmente o funcionamento dos algoritmos de acordo com as características específicas da máquina paralela, permitindo assim reduzir significativamente a quantidade total de informações transmitidas às custas de um pequeno aumento no número de rodadas, o que também não é previsto pelo modelo CGM e de certa forma até contraria sua estratégia original.

Finalmente, deve-se lembrar que os algoritmos propostos poderiam ter um desempenho ainda melhor em alguns casos específicos com os aperfeiçoamentos identificados durante a realização das experiências apresentadas neste trabalho.

10. Referências

- [ABK-95] M. ADLER, J. W. BYERS, R. M. KARP, "Parallel Sorting With Limited Bandwidth", Technical Report TR-95-031, International Computer Science Institute, Berkeley, Jul. 1995.
- [AKSS-86] M. AJTAI, J. KOMLÓS, W. L. STEIGER, E. SZEMERÉDI, "Deterministic Selection in $O(\log \log n)$ Parallel Time", *Proc. 18th Annual ACM Symposium on the Theory of Computing*, Berkeley, California, 188-195, May 1986.
- [AKSS-89] M. AJTAI, J. KOMLÓS, W. L. STEIGER, E. SZEMERÉDI, "Optimal Parallel Selection Has Complexity $O(\log \log n)$ ", *Journal of Computer and System Sciences*, vol. 38 (1), 125-133, Feb. 1989.
- [Ak-84] S. G. AKL, "An Optimal Algorithm for Parallel Selection", *Information Processing Letters*, vol. 19 (1), 47-50, Jul. 1984.
- [AP-90] Y. AZAR, N. PIPPENGER, "Parallel Selection", *Discrete Applied Mathematics*, vol. 27, 49-58, 1990.
- [Ba-94] D. A. BADER, "ColumnSort on the CM-5 using Split-C", Electrical Engineering Department, University of Maryland, Mar. 1994.
- [BHJ-95] D. A. BADER, D. R. HELMAN, J. JÁJÁ, "Practical Parallel Algorithms for Personalized Communication and Integer Sorting", Technical Report TR-95-101, Institute for Advanced Computer Studies and Department of Electrical Engineering, University of Maryland, College Park, Dec. 1995.
- [BJ-96] D. A. BADER, J. JÁJÁ, "Practical Parallel Algorithms for Dynamic Data Redistribution, Median Finding, and Selection", *Proc. 10th International Parallel Processing Symposium*, Honolulu, HI, 292-301, Apr. 1996.
- [BH-89] P. BEAME, J. HASTAD, "Optimal Bounds for Decision Problems on the CRCW PRAM", *Journal of the ACM*, vol. 36 (3), Jul. 1989.
- [BFPRT-73] M. BLUM, R. W. FLOYD, V. PRATT, R. L. RIVEST, R. E. TARJAN, "Time Bounds for Selection", *Journal of Computer and System Sciences*, vol. 7, 448-461, 1973.
- [CHR-93] S. CHAUDHURI, T. HAGERUP, R. RAMAN, "Approximate and Exact Deterministic Parallel Selection", *Proc. 18th International Symposium on Mathematical Foundations of*

Computer Science, vol. 711 of Lecture Notes in Computer Science, Berlin, Germany, Springer-Verlag, 352-361, 1993.

- [CFSV-94] T. CHEATHAM, A. FAHMY, D. C. STEFANESCU, L. G. VALIANT, "Bulk Synchronous Parallel Computing -- A Paradigm for Transportable Software", Technical Report TR-36-94, Harvard University, Computer Science Department, 1994.
- [CFSV-95] T. CHEATHAM, A. FAHMY, D. STEFANESCU, L. G. VALIANT, "Bulk Synchronous Parallel Computing -- A Paradigm for Transportable Software", *Proc. 28th Annual Hawaii Conference on System Sciences*, vol. II, IEEE Computer Society Press, Jan. 1995.
- [CT-87] F. CHIN, H. F. TING, "An Improved Algorithm for Finding the Median Distributively", *Algorithmica*, vol. 2, 235-249, 1987.
- [Ch-95] K. L. CHUNG, "Fast Median-Finding On Mesh-Connected Computers With Segmented Buses", *Nordic Journal of Computing*, vol. 2, 397-406, 1995.
- [Co-88a] R. J. COLE, "Parallel Merge Sort", *SIAM Journal on Computing*, vol. 17 (4), 770-785, 1988.
- [Co-88b] R. J. COLE, "An Optimally Efficient Selection Algorithm", *Information Processing Letters*, vol. 26 (6), 295-299, Jan. 1988.
- [CV-86] R. J. COLE, U. VISHKIN, "Deterministic coin tossing and accelerated cascades: micro and macro techniques for designing parallel algorithms", *Proc. 18th Annual ACM Symposium on the Theory of Computing*, Berkeley, California, 206-219, May 1986.
- [CY-85] R. J. COLE, C. K. YAP, "A Parallel Median Algorithm", *Information Processing Letters*, vol. 20 (3), 137-139, Apr. 1985.
- [CLR-90] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, *Introduction to Algorithms*, 2nd ed., MIT Press, 1990.
- [CKPSSSE-93] D. E. CULLER, R. M. KARP, D. A. PATTERSON, A. SAHAY, K. E. SCHAUER, E. SANTOS, R. SUBRAMONIAN, T. VON EICKEN, "LogP: Towards a Realistic Model of Parallel Computation", *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California, 1-12, May 1993.
- [DDDFK-95] F. DEHNE, X. DENG, P. DYMOND, A. FABRI, A. A. KOKHAR, "A Randomized Parallel 3D Convex Hull Algorithm for Coarse Grained Parallel Multicomputers", *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 1995.
- [DFR-93] F. DEHNE, A. FABRI, A. RAU-CHAPLIN, "Scalable Parallel Computational Geometry for Coarse Grained Multicomputers", *Proc. ACM Symposium on Computational Geometry*, 298-307, 1993.

- [DKF-94] F. DEHNE, C. KENYON, A. FABRI, "Scalable and Architecture Independent Parallel Geometric Algorithms with High Probability Optimal Time", *Proc. IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, 586-593, 1994.
- [DR-94] P. F. DIETZ, R. RAMAN, "Very Fast Optimal Parallel Algorithms for Heap Construction", *Proc. 6th IEEE Symposium on Parallel and Distributed Processing*, Oct. 1994.
- [FR-75a] R. W. FLOYD, R. L. RIVEST, "Expected Time Bounds for Selection", *Communications of the ACM*, vol. 18 (3), 165-172, 1975.
- [FR-75b] R. W. FLOYD, R. L. RIVEST, "Algorithm 489: The algorithm SELECT for finding the i th smallest of n elements", *Communications of the ACM*, vol. 18 (3), 173, Mar. 1975.
- [FV-94] T. FRACH, G. VIEHÖVER, "PowerPVM for PARIX 1.2 PowerPC", PARSYTEC Computer, Herzogenrath, Germany, Oct. 1994.
- [FM-70] W. D. FRAZER, A. C. MCKELLAR, "Samplesort: A Sampling Approach to Minimal Storage Tree Sorting", *Journal of the ACM*, vol. 17 (3), 496-507, Jul. 1970.
- [Fr-88] G. N. FREDERICKSON, "Distributed algorithms for selection in sets", *Journal of Computer and System Sciences*, vol. 37 (3), 337-348, Dec. 1988.
- [FJ-82] G. N. FREDERICKSON, D. B. JOHNSON, "The Complexity of Selection and Ranking in $X+Y$ and Matrices with Sorted Columns", *Journal of Computer and System Sciences*, vol. 24, 197-208, 1982.
- [GR-84] D. B. GANNON, J. VAN ROSENDALE, "On the Impact of Communication Complexity on the Design of Parallel Numerical Algorithms", *IEEE Transactions on Computers*, vol. 33 (12), 1180-1194, Dec. 1984.
- [GJ-79] M. R. GAREY, D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.
- [GBDJMS-93] A. GEIST, A. BEGUELIN, J. DONGARRA, W. JIANG, R. MANCHEK, V. SUNDERAM, "PVM 3 User's Guide and Reference Manual", Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1993.
- [GS-96] A. V. GERBESSIOTIS, C. J. SINIOLAKIS, "Deterministic Sorting and Randomized Median Finding on the BSP model", *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 223-232, 1996.
- [GV-92] A. V. GERBESSIOTIS, L. G. VALIANT, "Direct Bulk-Synchronous Parallel Algorithms", Technical Report TR-10-92, Harvard University, Computer Science Department, 1992.
- [GV-94] A. V. GERBESSIOTIS, L. G. VALIANT, "Direct Bulk-Synchronous Parallel Algorithms", *Journal of Parallel and Distributed Computing*, vol. 22, 251-267, 1994.

- [GG-93] M. GHOUSE, M. T. GOODRICH, "Experimental Evidence for the Power of Random Sampling in Practical Parallel Algorithms", *Proc. 7th International Parallel Processing Symposium*, 1993.
- [Go-97] S. GÖTZ, "Algorithms in CGM, BSP and BSP* Model: A Survey", Term Paper for Graduate Course "Parallel Algorithms and VLSI Implementation", Carleton University, Ottawa, Jan. 1997.
- [GL-94] W. GROPP, E. LUSK, "User's Guide for *mpich*, a Portable Implementation of MPI", Technical Report ANL-96/6, Argonne National Laboratory, 1994.
- [HBJ-95] D. R. HELMAN, D. A. BADER, J. JÁJÁ, "A Parallel Sorting Algorithm With an Experimental Study", Technical Report CS-TR-3549 and UMIACS-TR-95-102, Institute for Advanced Computer Studies, and Department of Electrical Engineering, University of Maryland, College Park, MD, Dec. 1995.
- [HBJ-96] D. R. HELMAN, D. A. BADER, J. JÁJÁ, "A Randomized Parallel Sorting Algorithm With an Experimental Study", Technical Report CS-TR-3669 and UMIACS-TR-96-53, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, Aug. 1996.
- [Ho-61] C. A. R. HOARE, "Algorithm 63 (PARTITION), Algorithm 64 (QUICKSORT), Algorithm 65 (FIND)", *Communications of the ACM*, vol. 4, 321-322, 1961.
- [Ho-62] C. A. R. HOARE, "Quicksort", *Computer Journal*, 5, 10-15, 1962.
- [HC-83] J. S. HUANG, Y. C. CHOW, "Parallel Sorting and data partitioning by sampling", *IEEE Computer Society's 7th International Computer Software and Applications Conference*, 627-631, Nov. 1983.
- [Ja-92] J. JÁJÁ, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992.
- [Kn-73] D. E. KNUTH, *The Art of Computer Programming, Vol. III: Sorting and Searching*, Addison-Wesley, Reading, 1973.
- [KNR-96] D. KRIZANC, L. NARAYANAN, R. RAMAN, "Fast Deterministic Selection on Mesh-Connected Processor Arrays", *Algorithmica*, vol. 15 (4), 319-331, Apr. 1996.
- [Le-85] T. LEIGHTON, "Tight Bounds on the Complexity of Parallel Sorting", *IEEE Transactions on Computers*, vol. c-34 (4), 344-354, Apr. 1985.
- [LLS-93] X. LI, P. LU, J. SCHAEFFER, J. SHILLINGTON, P.S. WONG, H. SHI, "On the Versatility of Parallel Sorting by Regular Sampling", *Parallel Computing*, vol. 19, 1079-1103, Oct. 1993.
- [LS-94] H. LI, K. SEVCIK, "Parallel Sorting by Overpartitioning", *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 46-56, 1994.

- [Mc-93a] W. F. MCCOLL, "Special Purpose Parallel Computing", *Proc. 1991 ALCOM Spring School on Parallel Computation*, in *Lectures on Parallel Computation*, Cambridge University Press, 261-336, 1993.
- [Mc-93b] W. F. MCCOLL, "General Purpose Parallel Computing", *Proc. 1991 ALCOM Spring School on Parallel Computation*, in *Lectures on Parallel Computation*, Cambridge University Press, 337-391, 1993.
- [Mc-94a] W. F. MCCOLL, "Scalable Parallel Computing: A Grand Unified Theory and its Practical Development", *Proc. 13th IFIP World Computer Congress*, vol. 1 (Invited Paper), Elsevier, 1994.
- [Mc-94b] W. F. MCCOLL, "BSP Programming", *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, American Mathematical Society, Princeton, May 1994.
- [Mc-94c] W. F. MCCOLL, "The BSP Approach to Architecture Independent Parallel Programming", Oxford University Computing Laboratory, Dec. 1994.
- [Mc-95] W. F. MCCOLL, "Scalable Computing", *Computer Science Today: Recent Trends and Developments*, vol. 1000 of *Lecture Notes in Computer Science*, Springer-Verlag, 46-61, 1995.
- [MPIF-94] MESSAGE PASSING INTERFACE FORUM, "MPI: A Message-Passing Interface Standard", *International Journal of Supercomputer Applications*, vol. 8 (3/4), 1994.
- [Na-92] L. NARAYANAN, "Selection, Sorting, and Routing on Mesh-Connected Processor Arrays", PhD thesis, University of Rochester, CS, May 1992.
- [NS-82] D. NASSIMI, S. SAHNI, "Parallel permutation and sorting algorithms and a new generalized connection network", *Journal of the ACM*, vol. 29 (3), 642-667, Jul. 1982.
- [NA-91] D. NUSSBAUM, A. AGARWAL, "Scalability of parallel machines", *Communications of the ACM*, vol. 34 (3), 57-61, Mar. 1991.
- [Ra-90] S. RAJASEKARAN, "Randomized parallel selection", *Foundations of Software Technology and Theoretical Computer Science*, in vol. 472 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 215-224, 1990.
- [Ra-96] S. RAJASEKARAN, "Randomized Selection on the Hypercube", *Journal of Parallel and Distributed Computing*, vol. 37, 187-193, 1996.
- [RT-92] S. RAJASEKARAN, T. TSANTILAS, "Optimal Routing Algorithms for Mesh-Connected Processor Arrays", *Algorithmica*, vol. 8, 21-38, 1992.
- [Re-85] R. REISCHUK, "Probabilistic Parallel Algorithms for Sorting and Selection", *SIAM Journal on Computing*, vol. 14 (2), 396-409, May 1985.
- [Ro-62] M. RODEH, "Finding the Median Distributively", *Journal of Computer and System Sciences*, vol. 24 (2), 162-166, 1982.

- [SSS-87] N. SANTORO, J. B. SIDNEY, S. J. SIDNEY, "On the Expected Complexity of Distributed Selection", *Proc. 4th Annual Symposium on Theoretical Aspects of Computer Science*, vol. 247 of Lecture Notes in Computer Science, Passau, Germany, Springer-Verlag, 456-467, Feb. 1987.
- [SSS-92] N. SANTORO, J. B. SIDNEY, S. J. SIDNEY, "A distributed selection algorithm and its expected communication complexity", *Theoretical Computer Science*, vol. 100 (1), 185-204, Jun. 1992.
- [Se-90] R. SEDGEWICK, *Algorithms in C*, Addison-Wesley Publishing Company, 1990.
- [Sen-90] S. SEN, "Finding an Approximate Median with High Probability in Constant Parallel Time", *Information Processing Letters*, vol. 34, 77-80, Mar. 1990.
- [ST-90] J. P. SHEU, J. S. TANG, "Efficient parallel k selection algorithm", *Information Processing Letters*, vol. 35 (6), 313-316, Sep. 1990.
- [SS-92] H. SHI, J. SCHAEFFER, "Parallel Sorting by Regular Sampling", *Journal of Parallel and Distributed Computing*, vol. 14 (4), 361-372, 1992.
- [SV-81] Y. SHILOACH, U. VISHKIN, "Finding the Maximum, Merging and Sorting in a Parallel Computation Model", *Journal of Algorithms*, vol. 2, 88-102, 1981.
- [Su-93] T. SUEL, "Optimal Deterministic Routing and Sorting on Mesh-Connected Arrays of Processors", Technical Report UTA/CS-TR-93-18, University of Texas at Austin, Department of Computer Sciences, Oct. 1993.
- [Va-75] L. G. VALIANT, "Parallelism in comparison problems", *SIAM Journal on Computing*, vol. 4 (3), 348-355, Sep. 1975.
- [Va-90] L. G. VALIANT, "A Bridging Model for Parallel Computation", *Communications of the ACM*, vol. 33 (8), 103-111, Aug. 1990.
- [Va-92] L. G. VALIANT, "A Combining Mechanism for Parallel Computers", *Proc. Symposium on Parallel Architectures and their Efficient Use*, Paderborn, Germany, 1-10, Nov. 1992.
- [Va-93] L. G. VALIANT, "Why BSP computers?", *Proc. 7th International Parallel Processing Symposium*, Newport Beach, California, 2-5, Apr. 1993.
- [Vi-94a] G. VIEHÖVER, "PARIX 1.2 PowerPC - User's Guide", PARSYTEC Computer, Herzogenrath, Germany, Aug. 1994.
- [Vi-94b] G. VIEHÖVER, "PARIX 1.2 PowerPC - Programmer's Guide", PARSYTEC Computer, Herzogenrath, Germany, Aug. 1994.
- [Wi-89] N. WIRTH, *Algoritmos e Estruturas de Dados*, Prentice-Hall do Brasil, 1989.

APÊNDICE A

LISTAGENS DOS

PROGRAMAS IMPLEMENTADOS

A.1 “alg_seq.h”: Protótipos dos algoritmos sequenciais

```
/*
 * Arquivo:      "alg_seq.h"
 *
 * Conteudo:     Prototipos dos algoritmos sequenciais basicos.
 *
 * Trabalho:     "Algoritmos de Selecao para Maquinas
 *               Paralelas com Memoria Distribuida"
 *
 * Autor:        Einar Luciano Gattoni Saukas
 *
 * Orientador:   Prof. Dr. Siang W. Song
 *
 */

/* Declaracao de tipos */

typedef double Elemento;      /* Tipo do elemento a ser selecionado */
typedef int    Indice;       /* Indice (posicao) de cada elemento */

typedef struct {
    Indice    campo_indice;
    Elemento  campo_elemento;
} Par_indice_elemento;

typedef struct {
    Indice menor;
    Indice menor_igual;
} Tam_particoes;

/* Declaracao de prototipos */

void ordena_merge_sort(Elemento *vetor, Indice l, Indice r,
                       Elemento *vetor_aux);

void ordena_insertion_sort(Elemento *vetor, Indice n);
void ordena_estrutura(Par_indice_elemento *vetor, Indice n);
void ordena_chave_conteudo(Indice *vetor_chave, Indice *vetor_conteudo,
                           Indice n);

Indice min_indice(Indice x, Indice y);
Indice max_indice(Indice x, Indice y);
Indice max_vetor_indice(Indice *vetor, Indice n);

Indice particiona_menores_seq(Elemento *vetor, Indice n, Elemento separador);
Indice particiona_menores_iguais_seq(Elemento *vetor, Indice n,
                                      Elemento separador);
Tam_particoes particiona_seq(Elemento *vetor, Indice n, Elemento separador);
void particiona_seq_simult(Elemento *vetor, Indice n, Elemento *separador,
                           Indice q, Indice delta_vet,
                           Tam_particoes *n_particoes);
Elemento seleciona_seq(Elemento *vetor, Indice n, Indice k,
                       Elemento *vetor_aux);
void seleciona_seq_simult(Elemento *vetor, Indice n, Indice *k, Indice q,
                           Indice delta_k, Elemento *vetor_aux,
                           Elemento *resultados);

/* FIM */
```


A.2 “alg_seq.c”: Algoritmos sequenciais básicos

```
/*
 * Arquivo:      "alg_seq.c"
 *
 * Conteudo:     Algoritmos sequenciais basicos.
 *
 *               Este arquivo contem os algoritmos sequenciais de ordenacao,
 *               particionamento e selecao, que funcionam como sub-rotinas dos
 *               algoritmos paralelos e tambem sao utilizados diretamente no
 *               caso p=1.
 *
 * Trabalho:     "Algoritmos de Selecao para Maquinas
 *               Paralelas com Memoria Distribuida"
 *
 * Autor:        Einar Luciano Gattoni Saukas
 *
 * Orientador:   Prof. Dr. Siang W. Song
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "alg_seq.h"

const Indice c_algoritmo_pick = 21;      /* Constante "c" do algoritmo PICK */

/*
 * ordena_merge_sort(vetor, l, r, vetor_aux)
 *
 * Ordenacao deterministica do "vetor[l..r]" pelo algoritmo "merge sort". Este
 * e' o metodo de ordenacao deterministico mais rapido para vetores grandes,
 * pois entre os algoritmos deterministicos de complexidade  $O(n \log n)$  e' o
 * que apresenta as menores constantes de tempo (de acordo com N. Wirth).
 *
 * A area "vetor_aux[l..r]" e' utilizado como area temporaria de trabalho.
 */

void ordena_merge_sort(Elemento *vetor, Indice l, Indice r, Elemento *vetor_aux)
{
    int i, j, k, m;
    if (r > l) {
        m=(r+l)/2;
        ordena_merge_sort(vetor, l, m, vetor_aux);
        ordena_merge_sort(vetor, m+1, r, vetor_aux);
        for (i=m+1; i > l; i--)
            vetor_aux[i-1]=vetor[i-1];
        for (j=m; j < r; j++)
            vetor_aux[r+m-j]=vetor[j+1];
        for (k=l; k <= r; k++)
            vetor[k]=(vetor_aux[i] <= vetor_aux[j] ? vetor_aux[i++] :
                vetor_aux[j--]);
    }
}

/*
 * ordena_insertion_sort(vetor, n)
 *
 * Ordenacao deterministica do "vetor[1..n]" pelo algoritmo "insertion sort".
 * Este e' o metodo de ordenacao deterministico mais rapido para vetores
 * pequenos, pois embora possua complexidade  $O(n^2)$ , apresenta as menores
 */
```

```

*    constantes de tempo (de acordo com N. Wirth).
*
*    A posicao vetor[0] deve ser valida. Esta posicao e' alterada durante a
*    execucao, mas seu valor e' restaurado no final do algoritmo.
*
*/

void ordena_insertion_sort(Elemento *vetor, Indice n)
{
    Elemento zero_original=vetor[0];
    Indice i;
    for (i=2; i <= n; i++) {
        Elemento novo=vetor[0]=vetor[i];    /* sentinela */
        Indice j;
        for (j=i; vetor[j-1] > novo; j--)
            vetor[j]=vetor[j-1];
        vetor[j]=novo;
    }
    vetor[0]=zero_original;
}

/*
* ordena_estrutura(vetor, n)
*
* Ordenacao deterministica do "vetor[1..n]", mas desta vez cada elemento do
* vetor e' composto dos campos campo_indice e campo_elemento, sendo apenas
* este ultimo utilizado como chave.
*
* O algoritmo de ordenacao utilizado e' o "insertion sort", por ser o metodo
* de ordenacao deterministico mais recomendavel para vetores pequenos.
*
* A posicao vetor[0] deve ser valida. Esta posicao e' alterada durante a
* execucao, mas seu valor e' restaurado no final do algoritmo.
*
*/

void ordena_estrutura(Par_indice_elemento *vetor, Indice n)
{
    Elemento zero_original=vetor[0].campo_elemento;
    Indice i;
    for (i=2; i <= n; i++) {
        Elemento nova_chave=vetor[0].campo_elemento=vetor[i].campo_elemento;
                                                                    /* sentinela */
        Indice novo_conteudo=vetor[i].campo_indice;
        Indice j;
        for (j=i; vetor[j-1].campo_elemento > nova_chave; j--) {
            vetor[j].campo_elemento=vetor[j-1].campo_elemento;
            vetor[j].campo_indice=vetor[j-1].campo_indice;
        }
        vetor[j].campo_elemento=nova_chave;
        vetor[j].campo_indice=novo_conteudo;
    }
    vetor[0].campo_elemento=zero_original;
}

/*
* ordena_chave_conteudo(vetor_chave, vetor_conteudo, n)
*
* Ordenacao deterministica do "vetor_chave[1..n]", mas todas as trocas de
* posicao de seus elementos devem ser repetidas em "vetor_conteudo[1..n]".
*
* O algoritmo de ordenacao utilizado e' o "insertion sort", por ser o metodo
* de ordenacao deterministico mais recomendavel para vetores pequenos.
*
* A posicao vetor_chave[0] deve ser valida. Esta posicao e' alterada durante

```

```

*      a execucao, mas seu valor e' restaurado no final do algoritmo.
*
*      Trata-se basicamente de uma repeticao da rotina anterior, mas com os campos
*      chave e conteudo pertencentes a estruturas diferentes. Os templates do
*      C++ fazem bastante falta nestes casos...
*
*/

void ordena_chave_conteudo(Indice *vetor_chave, Indice *vetor_conteudo,
                           Indice n)
{
    Indice zero_original=vetor_chave[0];
    Indice i;
    for (i=2; i <= n; i++) {
        Indice nova_chave=vetor_chave[0]=vetor_chave[i];    /* sentinela */
        Indice novo_conteudo=vetor_conteudo[i];
        Indice j;
        for (j=i; vetor_chave[j-1] > nova_chave; j--) {
            vetor_chave[j]=vetor_chave[j-1];
            vetor_conteudo[j]=vetor_conteudo[j-1];
        }
        vetor_chave[j]=nova_chave;
        vetor_conteudo[j]=novo_conteudo;
    }
    vetor_chave[0]=zero_original;
}

/*
* min_indice(x, y)
*
* Retorna o elemento minimo entre dois elementos do tipo "Indice".
*
*/

Indice min_indice(Indice x, Indice y)
{
    return (x < y ? x : y);
}

/*
* max_indice(x, y)
*
* Retorna o elemento maximo entre dois elementos do tipo "Indice".
*
*/

Indice max_indice(Indice x, Indice y)
{
    return (x > y ? x : y);
}

/*
* max_vetor_indice(vetor, n)
*
* Retorna o elemento maximo de "vetor[1..n]".
*
* Se n=0, retorna 0.
*
*/

Indice max_vetor_indice(Indice *vetor, Indice n)
{
    if (n > 0) {
        Indice i;

```

```

        Indice maximo=vetor[1];
        for (i=2; i <= n; i++)
            if (maximo < vetor[i])
                maximo=vetor[i];
        return maximo;
    }
    else
        return 0;
}

/*
 * particiona_menores_seq(vetor, n, separador)
 *
 * Particionamento do "vetor[1..n]", com base em um elemento "separador",
 * dividindo o vetor em duas particoes: no inicio do vetor ficam os
 * elementos estritamente menores que o separador, no final ficam os
 * elementos maiores ou iguais ao separador.
 *
 * O algoritmo requer n > 0.
 *
 * Retorno: tamanho da 1a particao (numero de elementos estritamente menores)
 *
 * Obs: No inicio da varredura, e' preciso verificar a cada passo se o vetor
 * acabou porque todos os elementos podem ser estritamente menores que o
 * separador ou todos os elementos podem ser maiores ou iguais ao separador.
 * Depois que um elemento estritamente menor e outro elemento maior ou igual
 * sao encontrados, esta verificacao nao e' mais necessaria.
 */

Indice particiona_menores_seq(Elemento *vetor, Indice n, Elemento separador)
{
    Indice i=0;
    Indice j=n+1;
    Elemento aux;

    while (vetor[++i] < separador)
        if (i >= n)
            return n;

    while (vetor[--j] >= separador)
        if (j <= 1)
            return 0;

    while (i < j) {
        aux=vetor[i];
        vetor[i]=vetor[j];
        vetor[j]=aux;
        while (vetor[++i] < separador)
            ;
        while (vetor[--j] >= separador)
            ;
    }
    return j;
}

/*
 * particiona_menores_iguais_seq(vetor, n, separador)
 *
 * Particionamento do "vetor[1..n]", com base em um elemento "separador",
 * dividindo o vetor em duas particoes: no inicio do vetor ficam os
 * elementos menores ou iguais ao separador, no final ficam os elementos
 * estritamente maiores.
 *
 * O algoritmo requer n > 0.

```

```

*
* Retorno: tamanho da 1a particao (numero de elementos menores ou iguais)
*
* Obs: No inicio da varredura, e' preciso verificar a cada passo se o vetor
* acabou porque todos os elementos podem ser menores ou iguais ao
* separador ou todos os elementos podem ser estritamente maiores que o
* separador. Depois que um elemento menor ou igual e outro elemento
* estritamente maior sao encontrados, esta verificacao nao e' mais
* necessaria.
*
*/

Indice particiona_menores_iguais_seq(Elemento *vetor, Indice n,
                                     Elemento separador)
{
    Indice i=0;
    Indice j=n+1;
    Elemento aux;

    while (vetor[++i] <= separador)
        if (i >= n)
            return n;

    while (vetor[--j] > separador)
        if (j <= 1)
            return 0;

    while (i < j) {
        aux=vetor[i];
        vetor[i]=vetor[j];
        vetor[j]=aux;
        while (vetor[++i] <= separador)
            ;
        while (vetor[--j] > separador)
            ;
    }
    return j;
}

/*
* particiona_seq(vetor, n, separador)
*
* Particionamento do "vetor[1..n]", com base em um elemento "separador",
* dividindo o vetor em tres particoes: no inicio do vetor ficam os
* elementos estritamente menores que o separador, em seguida os elementos
* iguais, no final os elementos estritamente maiores.
*
* O algoritmo realiza uma varredura inicial no vetor de entrada, produzindo
* uma particao com elementos estritamente menores que o separador e outra
* particao com elementos iguais ou maiores. Em seguida realizada uma
* nova varredura apenas nesta segunda particao, produzindo uma particao com
* elementos iguais e outra com elementos estritamente maiores. Portanto a
* complexidade do algoritmo e' O(n).
*
* O algoritmo requer n > 0.
*
* Retorno: Campo .menor          = tamanho da 1a particao
*          Campo .menor_igual    = soma dos tamanhos da 1a e 2a particoes
*
*/

Tam_particoes particiona_seq(Elemento *vetor, Indice n, Elemento separador)
{
    Tam_particoes n_part;
    n_part.menor_igual=n_part.menor=particiona_menores_seq(vetor, n, separador);
    if (n_part.menor != n)

```

```

        n_part.menor_igual+=particiona_menores_iguais_seq(vetor+n_part.menor,
                                                            n-n_part.menor,
                                                            separador);
    return n_part;
}

/*
 * particiona_seq_simult(vetor, n, separador, q, delta_vet, n_particoes)
 *
 * Particionamento do "vetor[1..n]", com base em um conjunto de q elementos
 * "separador[1..q]", dividindo o vetor em O(q) particoes: no inicio do
 * vetor ficam os elementos estritamente menores que separador[1], em
 * seguida os elementos iguais a separador[1], depois os elementos
 * estritamente maiores que separador[1] e estritamente menores que
 * separador[2], depois os elementos iguais a separador[2], etc.
 *
 * Este algoritmo e' uma "extensao" do algoritmo de particionamento anterior,
 * adaptado para a utilizacao de diversos separadores, realizando o
 * particionamento em "forma binaria": o vetor de entrada e' inicialmente
 * particionado em relacao a separador[q/2+1], utilizando o algoritmo de
 * particionamento simples anterior, em seguida o algoritmo e' aplicado
 * recursivamente em cada particao. Portanto a complexidade deste algoritmo
 * e' O(n log q).
 *
 * O algoritmo requer n > 0 e q > 0.
 *
 * Retorno: n_particoes[i].menor      = num. de elementos < separador[i]
 *          n_particoes[i].menor_igual = num. de elementos <= separador[i]
 *          (1 <= i <= q)
 */

void particiona_seq_simult(Elemento *vetor, Indice n, Elemento *separador,
                           Indice q, Indice delta_vet,
                           Tam_particoes *n_particoes)
{
    Indice indice_medio=q/2+1;
    Indice indice_prev;
    Indice indice_prox;

    /* Particiona o conjunto de entrada com base em separador[q/2+1] */
    n_particoes[indice_medio]=particiona_seq(vetor, n, separador[indice_medio]);

    /* Aplica algoritmo recursiv. para separador[1..q/2], se necessario */
    for (indice_prev=indice_medio-1;
         indice_prev >= 1 && separador[indice_prev] == separador[indice_medio];
         indice_prev--) {
        n_particoes[indice_prev].menor=n_particoes[indice_medio].menor+delta_vet;
        n_particoes[indice_prev].menor_igual=n_particoes[indice_medio].menor_igual
                                                +delta_vet;
    }
    if (indice_prev >= 1)
        particiona_seq_simult(vetor, n_particoes[indice_medio].menor, separador,
                              indice_prev, delta_vet, n_particoes);

    /* Aplica algoritmo recursiv. para separador[q/2+2..q], se necessario */
    for (indice_prox=indice_medio+1;
         indice_prox <= q && separador[indice_prox] == separador[indice_medio];
         indice_prox++) {
        n_particoes[indice_prox].menor=n_particoes[indice_medio].menor+delta_vet;
        n_particoes[indice_prox].menor_igual=n_particoes[indice_medio].menor_igual
                                                +delta_vet;
    }
}

```

```

    if (indice_prox <= q)
        particiona_seq_simult(vetor+n_particoes[indice_medio].menor_igual,
                               n-n_particoes[indice_medio].menor_igual,
                               separador+indice_prox-1, q-indice_prox+1,
                               delta_vet+n_particoes[indice_medio].menor_igual,
                               n_particoes+indice_prox-1);

    /* Ajusta o resultado e retorna */

    n_particoes[indice_medio].menor+=delta_vet;
    n_particoes[indice_medio].menor_igual+=delta_vet;
}

/*
 * seleciona_seq(vetor, n, k, vetor_aux)
 *
 * Selecao deterministica do k-esimo elemento do "vetor[1..n]".
 *
 * O algoritmo de selecao utilizado e' o "PICK" de Blum, Floyd, Pratt, Rivest
 * e Tarjan.
 *
 * A posicao vetor[0] deve ser valida. Esta posicao e' alterada durante a
 * execucao, mas seu valor e' restaurado no final do algoritmo.
 *
 * O valor de k deve valido: 1 <= k <= n.
 *
 * A area "vetor_aux[0..n/2+1]" e' utilizado como area temporaria de trabalho.
 *
 * Retorno: valor do k-esimo elemento.
 */
Elemento seleciona_seq(Elemento *vetor, Indice n, Indice k, Elemento *vetor_aux)
{
    /* Se numero de elementos e' suficientemente pequeno, resolve diretamente */

    if (n <= c_algoritmo_pick) {
        ordena_insertion_sort(vetor, n);
        return vetor[k];
    }

    /* Caso contrario: */

    else {

        /* Arranja vetor em n/c colunas de tamanho c */

        Indice num_grupos=n/c_algoritmo_pick;
        Elemento *grupo=vetor;

        /* Determina a mediana de cada grupo usando "insertion sort" */

        Indice i;
        for (i=1; i <= num_grupos; i++) {
            ordena_insertion_sort(grupo, c_algoritmo_pick);
            vetor_aux[i]=grupo[c_algoritmo_pick/2+1];
            grupo+=c_algoritmo_pick;
        }

        /* Determina tambem a mediana dos restantes (n mod c) elementos */

        {
            Indice tam_ultimo_grupo=n%c_algoritmo_pick;
            if (tam_ultimo_grupo) {
                ordena_insertion_sort(grupo, tam_ultimo_grupo);
            }
        }
    }
}

```

```

        vetor_aux[++num_grupos]=grupo[tam_ultimo_grupo/2+1];
    }
}
{
/* Aplica algoritmo recursiv. para determinar mediana das n/c medianas */

    Elemento separador=seleciona_seq(vetor_aux, num_grupos, num_grupos/2+1,
                                     vetor_aux+num_grupos+2);

/* Particiona vetor usando a mediana das medianas como separador */

    Tam_particoes n_particoes=particiona_seq(vetor, n, separador);

/* Aplica algoritmo recursivamente apenas em uma das particoes */

    if (k <= n_particoes.menor)
        return seleciona_seq(vetor, n_particoes.menor, k, vetor_aux);
    else if (k > n_particoes.menor_igual)
        return seleciona_seq(vetor+n_particoes.menor_igual,
                             n-n_particoes.menor_igual,
                             k-n_particoes.menor_igual, vetor_aux);
    else
        return vetor[k];
}
}

/*
* seleciona_seq_simult(vetor, n, k, q, delta_k, vetor_aux, resultados)
*
* Selecao deterministica dos elementos (k[1]-delta_k)..(k[q]-delta_k) do
* "vetor[1..n]". O valor "delta_k" funciona como um "offset" dos k-esimos
* elementos, evitando a necessidade de reajustar todo o conjunto k[1..q]
* quando uma particao e' descartada.
*
* Este algoritmo e' uma "extensao" do algoritmo de selecao "PICK". A cada
* particionamento do conjunto de elementos, os valores dos q k-esimos sao
* comparados com o tamanho das particoes, possivelmente dividindo o
* conjunto de q problemas em 2 conjuntos de problemas disjuntos. Evita-se
* assim a necessidade de repetir particionamentos para os diversos
* k-esimos.
*
* A posicao vetor[0] deve ser valida. Esta posicao e' alterada durante a
* execucao, mas seu valor e' restaurado no final do algoritmo.
*
* O conjunto k[1..q] deve estar ordenado.
*
* O valor de cada k[i] deve valido: 1 <= (k[i]-delta_k) <= n (1 <= i <= q).
*
* A area "vetor_aux[0..n/2+1]" e' utilizado como area temporaria de trabalho.
*
* Retorno: resultados[i] = valor do k[i]-esimo elemento (1 <= i <= q).
*
* Obs: A divisao do conjunto k[1..q] e' feita por busca sequencial, pois
* neste trabalho emprega valores pequenos de q. Caso haja interesse em
* empregar valores maiores de q, deve-se empregar busca binaria.
*/

void seleciona_seq_simult(Elemento *vetor, Indice n, Indice *k, Indice q,
                          Indice delta_k, Elemento *vetor_aux,
                          Elemento *resultados)
{
    Indice i;

```



```

/* Se numero de elementos e' suficientemente pequeno, resolve diretamente */
if (n <= c_algoritmo_pick) {
    ordena_insertion_sort(vetor, n);
    for (i=1; i <= q; i++)
        resultados[i]=vetor[k[i]-delta_k];
}

/* Caso contrario: */
else {

    /* Arranja vetor em n/c colunas de tamanho c */

    Indice num_grupos=n/c_algoritmo_pick;
    Elemento *grupo=vetor;

    /* Determina a mediana de cada grupo usando "insertion sort" */
    for (i=1; i <= num_grupos; i++) {
        ordena_insertion_sort(grupo, c_algoritmo_pick);
        vetor_aux[i]=grupo[c_algoritmo_pick/2+1];
        grupo+=c_algoritmo_pick;
    }

    /* Determina tambem a mediana dos restantes (n mod c) elementos */
    {
        Indice tam_ultimo_grupo=n%c_algoritmo_pick;
        if (tam_ultimo_grupo) {
            ordena_insertion_sort(grupo, tam_ultimo_grupo);
            vetor_aux[++num_grupos]=grupo[tam_ultimo_grupo/2+1];
        }
    }

    /* Aplica algoritmo de selecao para determinar mediana das n/c medianas */
    Elemento separador=seleciona_seq(vetor_aux, num_grupos, num_grupos/2+1,
                                     vetor_aux+num_grupos+2);

    /* Particiona vetor usando a mediana das medianas como separador */
    Tam_particoes n_particoes=particiona_seq(vetor, n, separador);

    /* Aplica algoritmo recursiv. na particao adequada, dividindo problemas */
    /* Problemas na particao de elementos estritamente menores */
    for (i=1; i <= q && k[i]-delta_k <= n_particoes.menor; i++)
        ;
    if (i > 1)
        seleciona_seq_simult(vetor, n_particoes.menor, k, i-1, delta_k,
                             vetor_aux, resultados);

    /* Problemas na particao de elementos iguais (solucao imediata) */
    for (; i <= q && k[i]-delta_k <= n_particoes.menor_igual; i++)
        resultados[i]=separador;

    /* Problemas na particao de elementos estritamente maiores */
    if (i <= q)
        seleciona_seq_simult(vetor+n_particoes.menor_igual,
                             n-n_particoes.menor_igual, k+i-1, q-i+1,
                             delta_k+n_particoes.menor_igual, vetor_aux,
                             resultados+i-1);
}

```

```
    }  
  }  
  
/* FIM */
```

A.3 “alg_par.h”: Protótipos dos algoritmos paralelos

```
/*
 * Arquivo:      "alg_par.h"
 *
 * Conteudo:     Prototipos dos algoritmos paralelos.
 *
 * Trabalho:     "Algoritmos de Selecao para Maquinas
 *               Paralelas com Memoria Distribuida"
 *
 * Autor:        Einar Luciano Gattoni Saukas
 *
 * Orientador:   Prof. Dr. Siang W. Song
 *
 */

/* Declaracao de comandos de depuracao */

#define  DEBUG( COMANDO )      /* COMANDO */

/* Declaracao de variaveis de depuracao */

extern unsigned int  tempo_2_1, tempo_2_2, tempo_2_3, tempo_2_4, tempo_2_5,
                    tempo_2_6, tempo_2_8, tempo_2_9, tempo_3_1, tempo_3_2,
                    tempo_4_1, tempo_4_2, tempo_5,  tempo_6,  tempo_7,
                    tempo_8,  tempo_9,  tempo_11, num_rodadas,
                    num_redist,          num_restantes;

/* Declaracao de prototipos */

void zero_cronometro(void);

Elemento algoritmo1(Elemento *vetor_entrada, Indice n, Indice p, Indice k,
                    Elemento *vetor_aux, Indice proc_num, Indice c);

void algoritmo2(Elemento *vetor_entrada, Indice n, Indice p, Indice *k,
                Indice q, Elemento *vetor_aux, Elemento *vetor_aux2,
                Indice proc_num, Indice c, Elemento *resultados);

void seleciona_paralelo_simult(Elemento *vetor_entrada, Indice n, Indice p,
                               Indice *k, Indice q, Elemento *vetor_aux,
                               Elemento *vetor_aux2, Indice proc_num, Indice c,
                               Elemento *resultados, Indice *prob_distintos,
                               Elemento *vetor_local[], Indice n_local[],
                               Indice delta_k[], Indice vetor_indice_inicial[],
                               Indice vetor_indice_final[],
                               Tam_particoes result_local_tam_particoes[],
                               Indice flag_result[]);

void algoritmo3(Elemento *vetor_entrada, Indice n, Indice p,
                Elemento *vetor_aux, Elemento *vetor_aux2, Indice proc_num,
                Indice c, Elemento *resultados);

/* FIM */
```

A.4 “alg_par.c”: Algoritmos paralelos principais

```
/*
 * Arquivo:      "alg_par.c"
 *
 * Conteudo:     Algoritmos paralelos principais.
 *
 * Trabalho:     "Algoritmos de Selecao para Maquinas
 *               Paralelas com Memoria Distribuida"
 *
 * Autor:        Einar Luciano Gattoni Saukas
 *
 * Orientador:   Prof. Dr. Siang W. Song
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "alg_seq.h"
#include "alg_com.h"
#include "alg_par.h"

/* Declaracao de variaveis de depuracao */

unsigned int tempo_2_1, tempo_2_2, tempo_2_3, tempo_2_4, tempo_2_5,
             tempo_2_6, tempo_2_8, tempo_2_9, tempo_3_1, tempo_3_2,
             tempo_4_1, tempo_4_2, tempo_5,   tempo_6,   tempo_7,
             tempo_8,   tempo_9,   tempo_11, num_rodadas,
             num_redist,          num_restantes;

/*
 * zera_cronometro()
 *
 * Zera todas as variaveis de depuracao (de desempenho).
 *
 */

void zera_cronometro(void)
{
    tempo_2_1= tempo_2_2= tempo_2_3= tempo_2_4= tempo_2_5=
    tempo_2_6= tempo_2_8= tempo_2_9= tempo_3_1= tempo_3_2=
    tempo_4_1= tempo_4_2= tempo_5=   tempo_6=   tempo_7=
    tempo_8=   tempo_9=   tempo_11= num_rodadas=
    num_redist=          num_restantes=0;
}

/*
 * algoritmo1(vetor_entrada, n, p, k, vetor_aux, proc_num, c)
 *
 * Algoritmo paralelo de selecao do k-esimo elemento.
 *
 * Cada um dos p processadores armazena em sua memoria local n/p elementos
 * em "vetor_entrada[1..n/p)".
 *
 * O indice deste processador e' dado por "proc_num" (1 <= proc_num <= p).
 *
 * O parametro "c" ajusta o numero de rodadas de comunicacao.
 *
 * O valor de k deve valido: 1 <= k <= n.
 *
 * A area "vetor_aux[0..(n/p)/2+1]" e' utilizado como area temporaria de
```

```

*      trabalho (pelo algoritmo de selecao sequencial).
*
*      Retorno: valor do k-esimo elemento.
*
*/

Elemento algoritmo1(Elemento *vetor_entrada, Indice n, Indice p, Indice k,
                    Elemento *vetor_aux, Indice proc_num, Indice c)
{
    Elemento *vetor_local=vetor_entrada;
    Indice n_local=(n%p < proc_num ? n/p : n/p+1);

    Indice N;
    Indice i;

    /* 1. Iniciar N:=n */
    N=n;

    if (p != 1) {
        /* 2. Repetir rodada ate' N <= n/cp */
        while (N > n/(c*p)) {

            Tam_particoes n_local_tam_particoes;
            Tam_particoes n_total_tam_particoes;

            Elemento mediana_local;
            Elemento mediana_ponderada;

            Par_indice_elemento vetor_n_mediana_local[MAX_P+1];

            num_rodadas++;

            /* 2.1 Cada proc. i calcula a mediana m[i] de seus n[i] elementos */
            if (n_local > 0)
                mediana_local=seleciona_seq(vetor_local, n_local, n_local/2+1,
                                             vetor_aux);
            else
                mediana_local=0;

            DEBUG(marca_cronometro(&tempo_2_1));

            /* 2.2 Cada proc. i transfere m[i] e n[i] para o proc. 1 */
            comunic1_reune_medianas_locais(n_local, mediana_local,
                                           vetor_n_mediana_local);

            DEBUG(marca_cronometro(&tempo_2_2));

            /* 2.3 Proc. 1 calcula a mediana ponderada M */
            if (proc_num == 1) {
                Indice soma_total=0;
                Indice soma_parcial=0;

                ordena_estrutura(vetor_n_mediana_local, p);

                for (i=1; i <= p; i++)
                    soma_total+=vetor_n_mediana_local[i].campo_indice;
                for (i=1; i <= p; i++) {
                    soma_parcial+=vetor_n_mediana_local[i].campo_indice;
                    if (2*soma_parcial >= soma_total)
                        break;
                }
            }
        }
    }
}

```

```

        mediana_ponderada=vetor_n_mediana_local[i].campo_elemento;
    }

    DEBUG(marca_cronometro(&tempo_2_3);)

    /* 2.4 Proc. 1 irradia M para os demais processadores */
    comunic1_irradia_mediana_ponderada(&mediana_ponderada);

    DEBUG(marca_cronometro(&tempo_2_4);)

    /* 2.5 Cada proc. calcula l[i], e[i] e g[i] de M */
    if (n_local > 0)
        n_local_tam_particoes=particiona_seq(vetor_local, n_local,
                                                mediana_ponderada);
    else
        n_local_tam_particoes.menor=n_local_tam_particoes.menor_igual=0;

    DEBUG(marca_cronometro(&tempo_2_5);)

    /* 2.6 Cada proc. i transfere l[i], e[i] e g[i] para o proc. 1 */
    /* 2.7 Proc. 1 calcula L, E, G de M */
    comunic1_adiciona_tamanhos_particoes(n_local_tam_particoes,
                                          &n_total_tam_particoes);

    DEBUG(marca_cronometro(&tempo_2_6);)

    /* 2.8 Proc. 1 irradia L, E, G para os demais processadores */
    comunic1_irradia_somas_tamanhos_particoes(&n_total_tam_particoes);

    DEBUG(marca_cronometro(&tempo_2_8);)

    /* 2.9 CASE: */
    /*     se k <= L: Cada proc. i mantém apenas elem. menores do que M */

    if (k <= n_total_tam_particoes.menor) {
        N=n_total_tam_particoes.menor;
        n_local=n_local_tam_particoes.menor;
    }

    /*     se k > L+E: Cada proc. i mantém apenas elem. maiores do que M */

    else if (k > n_total_tam_particoes.menor_igual) {
        N=n_total_tam_particoes.menor_igual;
        k=n_total_tam_particoes.menor_igual;
        n_local-=n_local_tam_particoes.menor_igual;
        vetor_local+=n_local_tam_particoes.menor_igual;
    }

    /*     se L < k <= L+E: Retorna solucao M e aborta execucao */

    else
        return mediana_ponderada;

    DEBUG(marca_cronometro(&tempo_2_9);)
}

/* 3. Todos os N elementos restantes são transferidos para o proc. 1 */
comunic1_reune_elementos_restantes(n_local, vetor_local, N,
                                    vetor_entrada);

num_restantes=N;

```

```

        DEBUG(marca_cronometro(&tempo_3_2);)
    }

    /* 4. Proc. 1 termina sequencialmente a resolucao do problema */
    if (proc_num == 1) {
        Elemento solucao=seleciona_seq(vetor_entrada, N, k, vetor_aux);

        DEBUG(marca_cronometro(&tempo_4_1);)

        return solucao;
    }
    else
        return 0;
}

/*
* algoritmo2(vetor_entrada, n, p, k, q, vetor_aux, vetor_aux2, proc_num, c,
*            resultados)
*
* Algoritmo paralelo de selecao simultanea de q k-esimos elementos.
*
* Cada um dos p processadores armazena em sua memoria local n/p elementos
* em "vetor_entrada[1..n/p]".
*
* O indice deste processador e' dado por "proc_num" (1 <= proc_num <= p).
*
* O parametro "c" ajusta o numero de rodadas de comunicacao.
*
* O valor de cada k deve valido: 1 <= k[j] <= n (1 <= j <= q).
*
* A area "vetor_aux[0..(n/p)/2+1]" e' utilizado como area temporaria de
* trabalho (pelo algoritmo de selecao sequencial).
*
* A area "vetor_aux2[0..2*(n/p)+1]" e' utilizado como area temporaria de
* trabalho (para a redistribuicao dos elementos).
*
* Retorno: resultados[i] = valor do k[i]-esimo elemento (1 <= i <= q).
*/

void algoritmo2(Elemento *vetor_entrada, Indice n, Indice p, Indice *k,
                Indice q, Elemento *vetor_aux, Elemento *vetor_aux2,
                Indice proc_num, Indice c, Elemento *resultados)
{
    Indice prob_distintos;
    Elemento *vetor_local[MAX_Q+1];
    Indice n_local[MAX_Q+1];
    Indice delta_k[MAX_Q+1];
    Indice vetor_indice_inicial[MAX_Q+1];
    Indice vetor_indice_final[MAX_Q+1];

    Tam_particoes result_local_tam_particoes[MAX_Q+1];
    Indice flag_result[MAX_Q+1];

    prob_distintos=1;
    vetor_local[1]=vetor_entrada;
    n_local[1]=(n%p < proc_num ? n/p : n/p+1);
    delta_k[1]=0;
    vetor_indice_inicial[1]=1;
    vetor_indice_final[1]=q;

    seleciona_paralelo_simult(vetor_entrada, n, p, k, q, vetor_aux, vetor_aux2,
                             proc_num, c, resultados, &prob_distintos,

```

```

        vetor_local, n_local, delta_k,
        vetor_indice_inicial, vetor_indice_final,
        result_local_tam_particoes, flag_result);
    }

/*
 * seleciona_paralelo_simult(vetor_entrada, n, p, k, q, vetor_aux, vetor_aux2,
 *                           proc_num, c, resultados, prob_distintos,
 *                           vetor_local, n_local, delta_k,
 *                           vetor_indice_inicial, vetor_indice_final,
 *                           result_local_tam_particoes, flag_result)
 *
 * Sub-rotina de execucao do algoritmo paralelo de selecao simultanea de q
 * k-esimos elementos.
 *
 * Esta sub-rotina simplesmente implementa o algoritmo 2 descrito acima, mas
 * permitindo que as estruturas de dados principais sejam declaradas
 * externamente e recebidas como parametros, de forma que o algoritmo 3
 * possa tambem ter acesso a estas estruturas de dados.
 */

void seleciona_paralelo_simult(Elemento *vetor_entrada, Indice n, Indice p,
                              Indice *k, Indice q, Elemento *vetor_aux,
                              Elemento *vetor_aux2, Indice proc_num, Indice c,
                              Elemento *resultados, Indice *prob_distintos,
                              Elemento *vetor_local[], Indice n_local[],
                              Indice delta_k[], Indice vetor_indice_inicial[],
                              Indice vetor_indice_final[],
                              Tam_particoes result_local_tam_particoes[],
                              Indice flag_result[])
{
    Indice N[MAX_Q+1];
    Indice i;
    Indice j;

    Indice vetor_alocacao_problema[MAX_Q+1];
    Elemento *vetor_local2[MAX_Q+1];

    for (j=1; j <= q; j++)
        flag_result[j]=0;

    /* 1. Iniciar N[j]:=n e A[i,j]:=A[i], j=1..q */
    for (j=1; j <= *prob_distintos; j++)
        N[j]=n;

    if (p == 1)
        for (j=1; j <= *prob_distintos; j++) {
            vetor_alocacao_problema[j]=1;
            vetor_local2[j]=vetor_local[j];
        }
    else {

        /* 2. Repetir rodada ate' N[j] <= n/(c*max{q,p}), j=1..q */

        while (max_vetor_indice(N, *prob_distintos) > n/(c*max_indice(p, q))) {

            Tam_particoes n_local_tam_particoes[MAX_Q+1];
            Tam_particoes n_total_tam_particoes[MAX_Q+1];

            Elemento mediana_local[MAX_Q+1];
            Elemento mediana_ponderada[MAX_Q+1];

            Par_indice_elemento vetor_n_mediana_local[MAX_P*MAX_Q+1];

```



```

num_rodadas++;

/* 2.1 Cada proc. i calcula as medianas m[i,j] de seus n[i,j] */
/*      elementos em A[i,j] */

for (j=1; j <= *prob_distintos; j++)
    if (n_local[j] > 0)
        mediana_local[j]=seleciona_seq(vetor_local[j], n_local[j],
                                         n_local[j]/2+1, vetor_aux);
    else
        mediana_local[j]=0;

DEBUG(marca_cronometro(&tempo_2_1);)

/* 2.2 Cada proc. i transfere m[i,j] e n[i,j] para o proc. 1 */

comunic2_reune_medianas_locais(n_local, mediana_local,
                               vetor_n_mediana_local, *prob_distintos);

DEBUG(marca_cronometro(&tempo_2_2);)

/* 2.3 Proc. 1 calcula as medianas ponderadas M[j] */

if (proc_num == 1) {
    Par_indice_elemento vetor_n_mediana_local_aux[MAX_P+1];
    for (j=1; j <= *prob_distintos; j++) {
        Indice soma_total=0;
        Indice soma_parcial=0;

        for (i=1; i <= p; i++) {
            vetor_n_mediana_local_aux[i].campo_indice=
                vetor_n_mediana_local[(i-1)**prob_distintos+j].campo_indice;
            vetor_n_mediana_local_aux[i].campo_elemento=
                vetor_n_mediana_local[(i-1)**prob_distintos+j].campo_elemento;
        }

        ordena_estrutura(vetor_n_mediana_local_aux, p);

        for (i=1; i <= p; i++)
            soma_total+=vetor_n_mediana_local_aux[i].campo_indice;
        for (i=1; i <= p; i++) {
            soma_parcial+=vetor_n_mediana_local_aux[i].campo_indice;
            if (2*soma_parcial >= soma_total)
                break;
        }
        mediana_ponderada[j]=vetor_n_mediana_local_aux[i].campo_elemento;
    }
}

DEBUG(marca_cronometro(&tempo_2_3);)

/* 2.4 Proc. 1 irradia M[j] para os demais processadores */

comunic2_irradia_mediana_ponderada(mediana_ponderada, *prob_distintos);

DEBUG(marca_cronometro(&tempo_2_4);)

/* 2.5 Cada proc. calcula l[i,j], e[i,j] e g[i,j] de M[j] */

for (j=1; j <= *prob_distintos; j++)
    if (n_local[j] > 0)
        n_local_tam_particoes[j]=particiona_seq(vetor_local[j],
                                                  n_local[j],
                                                  mediana_ponderada[j]);
    else
        n_local_tam_particoes[j].menor=
            n_local_tam_particoes[j].menor_igual=0;

```

```

DEBUG(marca_cronometro(&tempo_2_5);)

/* 2.6 Cada proc. i transfere l[i,j], e[i,j] e g[i,j] para proc. 1 */
/* 2.7 Proc. 1 calcula L[j], E[j], G[j] de M[j] */

comunic2_adiciona_tamanhos_particoes(n_local_tam_particoes,
                                     n_total_tam_particoes,
                                     *prob_distintos);

DEBUG(marca_cronometro(&tempo_2_6);)

/* 2.8 Proc. 1 irradia L[j], E[j], G[j] para demais processadores */

comunic2_irradia_somas_tamanhos_particoes(n_total_tam_particoes,
                                           *prob_distintos);

DEBUG(marca_cronometro(&tempo_2_8);)

/* 2.9 Para cada grupo de problemas associados a conjuntos disjuntos */
for (j=*prob_distintos; j > 0; j--) {

    /* Se ha' problemas tanto na particao menor como na maior */
    /* entao divide o grupo de problemas em dois grupos */

    if (k[vetor_indice_inicial[j]]-delta_k[j] <=
        n_total_tam_particoes[j].menor &&
        k[vetor_indice_final[j]]-delta_k[j] >
        n_total_tam_particoes[j].menor_igual) {

        for (i=vetor_indice_inicial[j]+1;
            k[i]-delta_k[j] <= n_total_tam_particoes[j].menor; i++)
            ;

        (*prob_distintos)++;

        N[*prob_distintos]=n_total_tam_particoes[j].menor;
        n_local[*prob_distintos]=n_local_tam_particoes[j].menor;
        vetor_local[*prob_distintos]=vetor_local[j];
        delta_k[*prob_distintos]=delta_k[j];

        vetor_indice_inicial[*prob_distintos]=vetor_indice_inicial[j];
        vetor_indice_final[*prob_distintos]=i-1;
        vetor_indice_inicial[j]=i;
    }

    /* Se agora ha' problemas tanto na particao menor como na igual */
    /* entao resolve diretamente os problemas da particao igual */

    if (k[vetor_indice_final[j]]-delta_k[j] <=
        n_total_tam_particoes[j].menor_igual)
        for (; vetor_indice_final[j] >= vetor_indice_inicial[j] &&
            k[vetor_indice_final[j]]-delta_k[j] >
            n_total_tam_particoes[j].menor;
            vetor_indice_final[j]--) {
            resultados[vetor_indice_final[j]]=mediana_ponderada[j];
            flag_result[vetor_indice_final[j]]=1;

            /* Guarda numeros de elementos locais para alg. 3 passo 7 */
            result_local_tam_particoes[vetor_indice_final[j]].menor=
            n_local_tam_particoes[j].menor+(vetor_local[j]-vetor_entrada);
            result_local_tam_particoes[vetor_indice_final[j]].menor_igual=
            n_local_tam_particoes[j].menor_igual+(vetor_local[j]-vetor_entrada);
        }

    /* Se agora ha' problemas tanto na particao igual como na maior */

```

```

/*  entao resolve diretamente os problemas da particao igual  */
if (k[vetor_indice_inicial[j]]-delta_k[j] >
    n_total_tam_particoes[j].menor)
    for (;
        vetor_indice_inicial[j] <= vetor_indice_final[j] &&
        k[vetor_indice_inicial[j]]-delta_k[j] <=
            n_total_tam_particoes[j].menor_igual;
        vetor_indice_inicial[j]++) {
        resultados[vetor_indice_inicial[j]]=mediana_ponderada[j];
        flag_result[vetor_indice_inicial[j]]=1;

        /* Guarda numeros de elementos locais para alg. 3 passo 7 */
        result_local_tam_particoes[vetor_indice_inicial[j]].menor=
            n_local_tam_particoes[j].menor+(vetor_local[j]-vetor_entrada);
        result_local_tam_particoes[vetor_indice_inicial[j]].menor_igual=
            n_local_tam_particoes[j].menor_igual+(vetor_local[j]-vetor_entrada);
    }

/* Se agora nao ha' mais problemas restantes neste grupo */
/*  entao elimina este grupo de problemas  */

if (vetor_indice_inicial[j] > vetor_indice_final[j]) {
    if (j != *prob_distintos) {
        N[j]=N[*prob_distintos];
        n_local[j]=n_local[*prob_distintos];
        vetor_local[j]=vetor_local[*prob_distintos];
        delta_k[j]=delta_k[*prob_distintos];

        vetor_indice_inicial[j]=vetor_indice_inicial[*prob_distintos];
        vetor_indice_final[j]=vetor_indice_final[*prob_distintos];
    }
    j--;
    (*prob_distintos)--;
    if (prob_distintos == 0)
        return;
}

/* Se todos os problemas deste grupo estao na particao menor ou */
/*  estao na particao maior, entao elimina as demais particoes */

else
    if (k[vetor_indice_inicial[j]]-delta_k[j] <=
        n_total_tam_particoes[j].menor) {
        N[j]=n_total_tam_particoes[j].menor;
        n_local[j]=n_local_tam_particoes[j].menor;
    }
    else {
        N[j]-=n_total_tam_particoes[j].menor_igual;
        n_local[j]-=n_local_tam_particoes[j].menor_igual;
        vetor_local[j]+=n_local_tam_particoes[j].menor_igual;
        delta_k[j]+=n_total_tam_particoes[j].menor_igual;
    }
}

DEBUG(marca_cronometro(&tempo_2_9);)
}

/* 3. Os O(q) problemas sao redistribuidos entre O(p) processadores */

/* Um algoritmo "guloso" e' utilizado para determinar a alocao */
/* dos problemas entre os processadores */

{
    Indice vetor_tamanho_problema[MAX_Q+1];
    Indice vetor_indice_problema[MAX_Q+1];
    Indice carga_processador[MAX_P+1];

```

```

    for (j=1; j <= *prob_distintos; j++) {
        vetor_tamanho_problema[j]=N[j];
        vetor_indice_problema[j]=j;

        num_restantes+=N[j];
    }

    ordena_chave_conteudo(vetor_tamanho_problema, vetor_indice_problema,
                          *prob_distintos);

    for (i=1; i <= p; i++)
        carga_processador[i]=0;
    for (j=*prob_distintos; j > 0; j--) {
        indice_i_minimo=1;
        for (i=2; i <= p; i++)
            if (carga_processador[i_minimo] > carga_processador[i])
                i_minimo=i;
        carga_processador[i_minimo]+=N[vetor_indice_problema[j]];
        vetor_alocacao_problema[vetor_indice_problema[j]]=i_minimo;
    }

    DEBUG(marca_cronometro(&tempo_3_1);)

    /* Os elementos sao transferidos de acordo com a alocao calculada */
    comunic2_redist_elementos_restantes(vetor_local, n_local, N,
                                         vetor_alocacao_problema, vetor_aux2,
                                         vetor_local2, *prob_distintos);

    DEBUG(marca_cronometro(&tempo_3_2);)
}

/* 4. Cada proc. termina sequencialmente resolucao de problemas recebidos */
for (j=1; j <= *prob_distintos; j++)
    if (vetor_alocacao_problema[j] == proc_num) {
        seleciona_seq_simult(vetor_local2[j], N[j],
                             k+vetor_indice_inicial[j]-1,
                             vetor_indice_final[j]-vetor_indice_inicial[j]+1,
                             delta_k[j], vetor_aux,
                             resultados+vetor_indice_inicial[j]-1);
        for (i=vetor_indice_inicial[j]; i <= vetor_indice_final[j]; i++)
            flag_result[i]=2;
    }

    DEBUG(marca_cronometro(&tempo_4_1);)

    /* Cada proc. transfere seus resultados para o proc. 1 */
    if (p != 1) {
        comunic2_reune_resultados(resultados, flag_result, q);

        DEBUG(marca_cronometro(&tempo_4_2);)
    }
}

/*
* algoritmo3(vetor_entrada, n, p, vetor_aux, vetor_aux2, proc_num, c,
*            resultados)
*
* Algoritmo paralelo de ordenacao baseado em selecao.
*
* Cada um dos p processadores armazena em sua memoria local n/p elementos

```

```

*   em "vetor_entrada[1..n/p]".
*
*   O indice deste processador e' dado por "proc_num" ( $1 \leq \text{proc\_num} \leq p$ ).
*
*   O parametro "c" ajusta o numero de rodadas de comunicacao.
*
*   A area "vetor_aux[0..(n/p)/2+1]" e' utilizado como area temporaria de
*   trabalho (pelo algoritmo de selecao sequencial).
*
*   A area "vetor_aux2[0..2*(n/p)+1]" e' utilizado como area temporaria de
*   trabalho (para a realocacao dos elementos).
*
*   Ao final da execucao, os n/p menores elementos estarao armazenados em ordem
*   crescente no processador 1, os seguintes n/p menores elementos estarao em
*   ordem crescente no processador 2, etc.
*
*/

```

```

void algoritmo3(Elemento *vetor_entrada, Indice n, Indice p,
                Elemento *vetor_aux, Elemento *vetor_aux2, Indice proc_num,
                Indice c, Elemento *resultados)
{
    Indice n_final;

    if (p == 1)
        n_final=n;
    else {
        const Indice q=p-1;
        Indice k[MAX_P];
        Indice i;
        Indice j;

        Indice prob_distintos;
        Elemento *vetor_local[MAX_P];
        Indice n_local[MAX_P];
        Indice delta_k[MAX_P];
        Indice vetor_indice_inicial[MAX_P];
        Indice vetor_indice_final[MAX_P];

        Tam_particoes result_local_tam_particoes[MAX_P*(MAX_P-1)+1];
        Indice redist_local_tam_particoes[MAX_P*MAX_P+1];
        Indice flag_result[MAX_P];

        for (j=1; j <= n%p; j++)
            k[j]=j*(n/p+1);
        for (j=n%p+1; j <= q; j++)
            k[j]=j*(n/p);

        prob_distintos=1;
        vetor_local[1]=vetor_entrada;
        n_local[1]=(n%p < proc_num ? n/p : n/p+1);
        delta_k[1]=0;
        vetor_indice_inicial[1]=1;
        vetor_indice_final[1]=q;

        /* 1..4. Aplicar o algoritmo 2 para q=p-1 e k[j]=jn/p, j=1..q */
        seleciona_paralelo_simult(vetor_entrada, n, p, k, q, vetor_aux,
                                vetor_aux2, proc_num, c, resultados,
                                &prob_distintos, vetor_local, n_local, delta_k,
                                vetor_indice_inicial, vetor_indice_final,
                                result_local_tam_particoes+(proc_num-1)*q,
                                flag_result);

        if (prob_distintos > 0) {

```

```

/* 5. Cada proc. i comunica para os demais as solucoes de problemas */
/*      finalizados localmente (como proc. 1 ja' possui a solucao */
/*      de todos os problemas, basta irradiar esta informacao) */

comunic3_irradia_resultados(resultados, q);

DEBUG(marca_cronometro(&tempo_5);)

/* 6. Cada proc. i separa elementos iniciais em O(p) sub-conjuntos */

for (j=1; j <= prob_distintos; j++)
    if (n_local[j] > 0)
        particiona_seq_simult(vetor_local[j], n_local[j],
                               resultados+vetor_indice_inicial[j]-1,
                               vetor_indice_final[j]-vetor_indice_inicial[j]+1,
                               (vetor_local[j]-vetor_entrada),
                               result_local_tam_particoes+(proc_num-1)*q+
                               vetor_indice_inicial[j]-1);
    else
        for (i=vetor_indice_inicial[j]; i <= vetor_indice_final[j]; i++)
            result_local_tam_particoes[(proc_num-1)*q+i].menor=
                result_local_tam_particoes[(proc_num-1)*q+i].menor_igual=
                    vetor_local[j]-vetor_entrada;

    DEBUG(marca_cronometro(&tempo_6);)
}

/* 7. Cada proc. i transfere para proc. 1 numeros de elementos locais */

comunic3_reune_total_resultados(result_local_tam_particoes, q);

DEBUG(marca_cronometro(&tempo_7);)

/* 8. Proc. 1 determina divisao dos elementos iguais aos separadores */

if (proc_num == 1) {
    for (j=1; j <= q; j++) {
        Indice soma_menor_igual=0;
        for (i=1; i <= p; i++)
            soma_menor_igual+=
                result_local_tam_particoes[(i-1)*q+j].menor_igual;
        for (i=1; i <= p; i++) {
            Indice retirada=min_indice(soma_menor_igual-k[j],
                                         result_local_tam_particoes[(i-1)*q+j].menor_igual-
                                         result_local_tam_particoes[(i-1)*q+j].menor);
            redist_local_tam_particoes[(i-1)*p+j]=
                result_local_tam_particoes[(i-1)*q+j].menor_igual-retirada;
            soma_menor_igual-=retirada;
        }
    }
    for (i=1; i <= n%p; i++)
        redist_local_tam_particoes[(i-1)*p+p]=n/p+1;
    for (i=n%p+1; i <= p; i++)
        redist_local_tam_particoes[(i-1)*p+p]=n/p;

    num_redist=redist_local_tam_particoes[p]-
                redist_local_tam_particoes[1];
    for (i=2; i <= p; i++)
        num_redist+=redist_local_tam_particoes[(i-1)*p+p]-
                    (redist_local_tam_particoes[(i-1)*p+i]-
                     redist_local_tam_particoes[(i-1)*p+i-1]);
}

DEBUG(marca_cronometro(&tempo_8);)

/* 9. Proc. 1 transmite para os demais processadores esta divisao */

```

```

/* 10. Todos os elementos sao redistribuidos entre os processadores */
comunic3_redistribui_dados(redist_local_tam_particoes, vetor_entrada,
                           vetor_aux2, &n_final);

DEBUG(marca_cronometro(&tempo_9);)
}

/* 11. Cada proc. i ordena localmente seus elementos recebidos */
ordena_merge_sort(vetor_entrada, 1, n_final, vetor_aux2);

DEBUG(marca_cronometro(&tempo_11);)
}

/* FIM */

```

A.5 “alg_com.h”: Protótipos das funções de comunicação

```
/*
 * Arquivo:      "alg_com.h"
 *
 * Conteúdo:     Prototipos das funcoes de comunicacao.
 *
 *               Embora a implementacao destas funcoes seja completamente
 *               dependente da interface de comunicacao utilizada, os
 *               prototipos aqui definidos sao comuns a todas as
 *               implementacoes. Portanto os demais algoritmos sequenciais e
 *               paralelos nao sao afetados por esta dependencia.
 *
 *               Na pratica isto significa que, quando a implementacao dos
 *               algoritmos na interface PVM e' alterada para rodar na
 *               interface MPI, basta substituir o arquivo especifico das
 *               funcoes de comunicacao (no caso trocar o arquivo "alg_pvm.c"
 *               pelo arquivo "alg_mpi.c"), sem nenhuma alteracao nos demais
 *               arquivos.
 *
 * Trabalho:     "Algoritmos de Selecao para Maquinas
 *               Paralelas com Memoria Distribuida"
 *
 * Autor:        Einar Luciano Gattoni Saukas
 *
 * Orientador:   Prof. Dr. Siang W. Song
 */

/* Declaracao de tamanhos das estruturas de dados */

#define MAX_P      16          /* Numero maximo de processadores */
#define MAX_Q      128        /* Numero maximo de problemas simultaneos */

/* Declaracao de prototipos */

void marca_cronometro(unsigned int *tempo);
void erro(const char *msg);

void comunic_inicio(char *prog, int argc, char *argv[], int *p, int *proc_num);
void comunic_sincroniza(void);
void comunic_final(void);

void comunic1_reune_medianas_locais(Indice n_local, Elemento mediana_local,
                                   Par_indice_elemento vetor_n_mediana_local[]);
void comunic1_irradia_mediana_ponderada(Elemento *mediana_ponderada);
void comunic1_adiciona_tamanhos_particoes(Tam_particoes n_local_particoes,
                                           Tam_particoes *n_total_particoes);
void comunic1_irradia_somas_tamanhos_particoes(Tam_particoes *n_total_particoes);
void comunic1_reune_elementos_restantes(Indice n_local, Elemento *vetor_local,
                                       Indice n_total, Elemento *vetor_total);

void comunic2_reune_medianas_locais(Indice n_local[], Elemento mediana_local[],
                                   Par_indice_elemento vetor_n_mediana_local[],
                                   Indice q);
void comunic2_irradia_mediana_ponderada(Elemento mediana_ponderada[], Indice q);
void comunic2_adiciona_tamanhos_particoes(Tam_particoes n_local_particoes[],
                                           Tam_particoes n_total_particoes[],
                                           Indice q);
void comunic2_irradia_somas_tamanhos_particoes(Tam_particoes n_total_particoes[],
                                               Indice q);
void comunic2_redist_elementos_restantes(Elemento *vetor_local[],
                                       Indice n_local[], Indice N[],
```



```

Indice vetor_alocacao_problema[],
Elemento *vetor_aux2,
Elemento *vetor_local2[], Indice q);
void comunic2_reune_resultados(Elemento resultados[], Indice flag_result[],
Indice q);

void comunic3_irradia_resultados(Elemento resultados[], Indice q);
void comunic3_reune_total_resultados(Tam_particoes result_local_tam_particoes[],
Indice q);
void comunic3_redistribui_dados(Indice quant_redist[], Elemento *vetor_local,
Elemento *vetor_aux, Indice *n_local);

/* FIM */

```

A.6 “alg_pvm.c”: Funções de comunicação em PVM

```
/*
 * Arquivo:      "alg_pvm.c"
 *
 * Conteudo:     Funcoes de comunicacao implementadas em PVM.
 *
 *               Neste arquivo estao isoladas todas as funcoes cuja implementacao
 *               depende da interface de comunicacao escolhida.
 *
 * Trabalho:     "Algoritmos de Selecao para Maquinas
 *               Paralelas com Memoria Distribuida"
 *
 * Autor:        Einar Luciano Gattoni Saukas
 *
 * Orientador:   Prof. Dr. Siang W. Song
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "pvm3.h"
#include "alg_seq.h"
#include "alg_com.h"

/* Variaveis globais (utilizadas apenas no escopo deste arquivo) */

int comunic_p=0;                /* Numero de processadores */
int comunic_proc_num=0;        /* Indice deste processador */
int proc_tid=0;                /* Identificacao deste processador */
int *tids=0;                   /* Vetor de identificadores */

/*
 * marca_cronometro(tempo)
 *
 * Calcula o intervalo de tempo decorrido desde a ultima chamada a esta funcao,
 * somando este valor ao parametro "tempo".
 *
 * Como esta rotina determina intervalos de tempo entre chamadas consecutivas
 * (ao inves de tempos absolutos), e' necessario realizar uma primeira
 * chamada para "iniciar" a rotina.
 *
 * Se for utilizado um ponteiro nulo no parametro "tempo", a rotina e' iniciada
 * sem retornar o intervalo de tempo.
 *
 * O exemplo a seguir ilustra a utilizacao desta funcao na cronometragem de um
 * conjunto de rotinas A, B e C:
 *
 *
 * unsigned int tempo1=0;
 * unsigned int tempo2=0;
 * ...
 * marca_cronometro(0);          // inicia a rotina
 * A();
 * marca_cronometro(&tempo1)     // tempo1= <duracao de A>
 * B();
 * marca_cronometro(&tempo2);    // tempo2= <duracao de B>
 * C();
 * marca_cronometro(&tempo1);    // tempo1= <duracao de A> + <duracao de C>
 *
 */
```

```

void marca_cronometro(unsigned int *tempo)
{
    static unsigned int tempo_anterior=0;
    unsigned int tempo_atual=TimeNow();
    if (tempo)
        *tempo+=(tempo_atual-tempo_anterior);
    tempo_anterior=tempo_atual;
}

/*
 * erro(msg)
 *
 * Imprime a mensagem de erro "msg" e aborta a execucao do programa.
 *
 */

void erro(const char *msg)
{
    fprintf(stderr, "ERRO: %s\n", msg);
    exit(-1);
}

/*
 * comunic_inicio(prog, argc, argv, p, proc_num)
 *
 * Inicia o programa nos diversos processadores e, em cada processador,
 * determina o indice deste processador e o numero total de processadores.
 *
 * Esta rotina deve ser chamada no inicio do programa, antes de qualquer outra
 * rotina de comunicacao.
 *
 * O parametro "prog" contem o nome do executavel a ser carregado em cada
 * processador.
 *
 * O primeiro argumento da linha de comando deve conter o numero de
 * processadores desejado.
 *
 * Os parametros "argc" e "argv" sao os mesmos recebidos pela funcao "main()",
 * o que garante que todos os processadores sejam iniciados com os mesmos
 * argumentos da linha de comando.
 *
 * Retorno: p          = numero de processadores (p >= 1)
 *          proc_num = indice deste processador (1 <= proc_num <= p)
 *
 */

void comunic_inicio(char *prog, int argc, char *argv[], int *p, int *proc_num)
{
    comunic_p=(argc > 1 ? atoi(argv[1]) : 0);

    if (comunic_p != 1) {
        tids=(int *)malloc(comunic_p*sizeof(int));
        if (!tids)
            erro("malloc");

        proc_tid=pvm_mytid();
        tids[0]=pvm_parent();
        if (tids[0] < 0) {
            tids[0]=proc_tid;
            comunic_proc_num=1;
            pvm_spawn(prog, &argv[1], 0, "", comunic_p-1, &tids[1]);
            pvm_initsend(PvmDataDefault);
            pvm_pkint(tids, comunic_p, 1);
            pvm_mcast(&tids[1], comunic_p-1, 0);
        }
    }
}

```

```

    }
    else {
        Indice i;
        pvm_recv(tids[0], 0);
        pvm_upkint(tids, comunic_p, 1);
        for (i=2; i <= comunic_p; i++)
            if (proc_tid == tids[i-1]) {
                comunic_proc_num=i;
                break;
            }
    }
}
else
    comunic_proc_num=1;
*p=comunic_p;
*proc_num=comunic_proc_num;
}

/*
 * comunic_sincroniza()
 *
 * Sincroniza todos os processadores.
 *
 * Esta rotina e' chamada para garantir que todos os processadores estejam
 * prontos para iniciar a execucao dos algoritmos.
 */

void comunic_sincroniza(void)
{
    if (comunic_p != 1)
        if (comunic_proc_num == 1) {
            Indice i;
            for (i=2; i <= comunic_p; i++)
                pvm_recv(tids[i-1], 0);
            pvm_initsend(PvmDataDefault);
            pvm_pkint(&comunic_proc_num, 1, 1);
            pvm_mcast(&tids[1], comunic_p-1, 0);
        }
        else {
            pvm_initsend(PvmDataDefault);
            pvm_pkint(&comunic_proc_num, 1, 1);
            pvm_send(tids[0], 0);
            pvm_recv(tids[0], 0);
        }
}

/*
 * comunic_final()
 *
 * Finaliza o programa, liberando os recursos alocados para as rotinas de
 * comunicacao.
 *
 * Esta rotina deve ser chamada no final do programa, apos a execucao das
 * demais rotinas de comunicacao.
 */

void comunic_final(void)
{
    if (comunic_p != 1) {
        pvm_exit();
        free(tids);
    }
}

```

```

/*
 * comunic1_reune_medianas_locais(n_local, mediana_local, vetor_n_mediana_local)
 *
 * Executa o passo 2.2 do algoritmo 1:
 *
 * Cada processador envia, para o processador 1, sua mediana local e numero de
 * elementos restantes.
 *
 * Retorno (apenas no processador 1):
 *     vetor_n_mediana_local[i].campo_indice = numero de elem. no proc. i
 *     vetor_n_mediana_local[i].campo_elemento = mediana local do proc. i
 *                                           (1 <= i <= p)
 */

void comunic1_reune_medianas_locais(Indice n_local, Elemento mediana_local,
                                     Par_indice_elemento vetor_n_mediana_local[])
{
    if (comunic_proc_num != 1) {
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&n_local, 1, 1);
        if (n_local > 0)
            pvm_pkdouble(&mediana_local, 1, 1);
        pvm_send(tids[0], 0);
    }
    else {
        Indice i;

        vetor_n_mediana_local[1].campo_indice=n_local;
        vetor_n_mediana_local[1].campo_elemento=mediana_local;

        for (i=2; i <= comunic_p; i++) {
            pvm_recv(tids[i-1], 0);
            pvm_upkint(&vetor_n_mediana_local[i].campo_indice, 1, 1);
            if (vetor_n_mediana_local[i].campo_indice > 0)
                pvm_upkdouble(&vetor_n_mediana_local[i].campo_elemento, 1, 1);
            else
                vetor_n_mediana_local[i].campo_elemento=0;
        }
    }
}

/*
 * comunic1_irradia_mediana_ponderada(mediana_ponderada)
 *
 * Executa o passo 2.4 do algoritmo 1:
 *
 * O processador 1 envia a mediana ponderada para todos os demais
 * processadores.
 *
 * Retorno (em todos os processadores):
 *     mediana_ponderada = mediana ponderada dos elementos restantes
 */

void comunic1_irradia_mediana_ponderada(Elemento *mediana_ponderada)
{
    if (comunic_proc_num == 1) {
        pvm_initsend(PvmDataDefault);
        pvm_pkdouble(mediana_ponderada, 1, 1);
        pvm_mcast(&tids[1], comunic_p-1, 0);
    }
    else {
        pvm_recv(tids[0], 0);
    }
}

```

```

        pvm_upkdouble(mediana_ponderada, 1, 1);
    }
}

/*
 * comunic1_adiciona_tamanhos_particoes(n_local_particoes, n_total_particoes)
 *
 * Executa os passos 2.6 e 2.7 do algoritmo 1:
 *
 * Cada processador envia os tamanhos de suas particoes locais (numero de
 * elementos locais menores, iguais e maiores que a mediana ponderada), o
 * processador 1 calcula as suas somatorias.
 *
 * Na verdade, a implementacao do algoritmo trabalha apenas com o numero de
 * elementos locais menores e o numero de elementos locais menores ou iguais,
 * o que apresenta algumas vantagens do ponto de vista pratico.
 *
 * Retorno (apenas no processador 1):
 *      n_total_particoes->menor          = numero de elementos estritamente
 *                                         menores que a mediana ponderada
 *      n_total_particoes->menor_igual    = numero de elementos menores ou
 *                                         iguais que a mediana ponderada
 */

void comunic1_adiciona_tamanhos_particoes(Tam_particoes n_local_particoes,
                                          Tam_particoes *n_total_particoes)
{
    if (comunic_proc_num != 1) {
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&n_local_particoes.menor, 1, 1);
        pvm_pkint(&n_local_particoes.menor_igual, 1, 1);
        pvm_send(tids[0], 0);
    }
    else {
        Indice i;
        n_total_particoes->menor=n_local_particoes.menor;
        n_total_particoes->menor_igual=n_local_particoes.menor_igual;
        for (i=2; i <= comunic_p; i++) {
            Indice tam;
            pvm_recv(tids[i-1], 0);
            pvm_upkint(&tam, 1, 1);
            n_total_particoes->menor+=tam;
            pvm_upkint(&tam, 1, 1);
            n_total_particoes->menor_igual+=tam;
        }
    }
}

/*
 * comunic1_irradia_somas_tamanhos_particoes(n_total_particoes)
 *
 * Executa o passo 2.8 do algoritmo 1:
 *
 * O processador 1 envia o tamanho total das particoes para todos os demais
 * processadores.
 *
 * Retorno (em todos os processadores):
 *      n_total_particoes = tamanhos totais das particoes
 */

void comunic1_irradia_somas_tamanhos_particoes(Tam_particoes *n_total_particoes)
{
    if (comunic_proc_num == 1) {

```

```

        pvm_initsend(PvmDataDefault);
        pvm_pkint(&n_total_particoes->menor, 1, 1);
        pvm_pkint(&n_total_particoes->menor_igual, 1, 1);
        pvm_mcast(&tids[1], comunic_p-1, 0);
    }
    else {
        pvm_recv(tids[0], 0);
        pvm_upkint(&n_total_particoes->menor, 1, 1);
        pvm_upkint(&n_total_particoes->menor_igual, 1, 1);
    }
}

/*
 * comunic1_reune_elementos_restantes(n_local, vetor_local, n_total,
 *                                     vetor_total)
 *
 * Executa o passo 3 do algoritmo 1:
 *
 * Cada processador envia todos os seus elementos restantes (em
 *   vetor_local[1..n_local]) para o processador 1.
 *
 * Retorno (apenas no processador 1):
 *   vetor_total[1..n_total] = elementos restantes de todos os proc.
 */

void comunic1_reune_elementos_restantes(Indice n_local, Elemento *vetor_local,
                                       Indice n_total, Elemento *vetor_total)
{
    if (comunic_proc_num != 1) {
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&n_local, 1, 1);
        if (n_local > 0)
            pvm_pkdouble(&vetor_local[1], n_local, 1);
        pvm_send(tids[0], 0);
    }
    else {
        Indice i;
        if (vetor_total != vetor_local)
            for (i=1; i <= n_local; i++)
                vetor_total[i]=vetor_local[i];

        for (i=2; i <= comunic_p; i++) {
            Indice tam;
            pvm_recv(tids[i-1], 0);
            pvm_upkint(&tam, 1, 1);
            if (tam > 0) {
                pvm_upkdouble(&vetor_total[n_local+1], tam, 1);
                n_local+=tam;
            }
        }
    }
}

/*
 * comunic2_reune_medianas_locais(n_local, mediana_local, vetor_n_mediana_local,
 *                                 q)
 *
 * Executa o passo 2.2 do algoritmo 2:
 *
 * Cada processador envia, para o processador 1, a mediana local e número
 *   de elementos restantes correspondentes a cada um dos q conjuntos de
 *   problemas disjuntos.
 *
 * Retorno (apenas no processador 1):

```

```

*      vetor_n_mediana_local[(i-1)*q+j].campo_indice = numero de elem. do
*      proc. i e problema j (1 <= i <= p, 1 <= j <= q)
*      vetor_n_mediana_local[(i-1)*q+j].campo_elemento = mediana local do
*      proc. i e problema j (1 <= i <= p, 1 <= j <= q)
*
*/

void comunic2_reune_medianas_locais(Indice n_local[], Elemento mediana_local[],
                                     Par_indice_elemento vetor_n_mediana_local[],
                                     Indice q)
{
    Indice j;
    if (comunic_proc_num != 1) {
        pvm_initsend(PvmDataDefault);
        for (j=1; j <= q; j++) {
            pvm_pkint(&n_local[j], 1, 1);
            if (n_local[j] > 0)
                pvm_pkdouble(&mediana_local[j], 1, 1);
        }
        pvm_send(tids[0], 0);
    }
    else {
        Indice i;
        for (j=1; j <= q; j++) {
            vetor_n_mediana_local[j].campo_indice=n_local[j];
            vetor_n_mediana_local[j].campo_elemento=mediana_local[j];
        }
        for (i=2; i <= comunic_p; i++) {
            pvm_recv(tids[i-1], 0);
            for (j=1; j <= q; j++) {
                pvm_upkint(&vetor_n_mediana_local[(i-1)*q+j].campo_indice, 1, 1);
                if (vetor_n_mediana_local[(i-1)*q+j].campo_indice > 0)
                    pvm_upkdouble(&vetor_n_mediana_local[(i-1)*q+j].campo_elemento, 1,
1);
                else
                    vetor_n_mediana_local[(i-1)*q+j].campo_elemento=0;
            }
        }
    }
}

/*
* comunic2_irradia_mediana_ponderada(mediana_ponderada, q)
*
* Executa o passo 2.4 do algoritmo 2:
*
* O processador 1 envia as medianas ponderadas de cada um dos q conjuntos de
* problemas disjuntos para todos os demais processadores.
*
* Retorno (em todos os processadores):
*      mediana_ponderada[j] = mediana ponderada do conjunto j de problemas
*                          (1 <= j <= q)
*
*/

void comunic2_irradia_mediana_ponderada(Elemento mediana_ponderada[], Indice q)
{
    if (comunic_proc_num == 1) {
        pvm_initsend(PvmDataDefault);
        pvm_pkdouble(&mediana_ponderada[1], q, 1);
        pvm_mcast(&tids[1], comunic_p-1, 0);
    }
    else {
        pvm_recv(tids[0], 0);
        pvm_upkdouble(&mediana_ponderada[1], q, 1);
    }
}

```



```

}

/*
 * comunic2_adiciona_tamanhos_particoes(n_local_particoes, n_total_particoes, q)
 *
 * Executa os passos 2.6 e 2.7 do algoritmo 2:
 *
 * Cada processador envia os tamanhos de suas particoes locais (numero de
 * elementos locais menores, iguais e maiores que a mediana ponderada) de
 * cada um dos q conjuntos de problemas disjuntos, o processador 1 calcula
 * as suas somatorias.
 *
 * Na verdade, a implementacao do algoritmo trabalha apenas com o numero de
 * elementos locais menores e o numero de elementos locais menores ou iguais,
 * o que apresenta algumas vantagens do ponto de vista pratico.
 *
 * Retorno (apenas no processador 1):
 *     n_total_particoes[j].menor           = numero de elementos estritamente
 *     menores que a mediana ponderada do conjunto de problemas j
 *     n_total_particoes[j].menor_igual = numero de elementos menores ou
 *     iguais que a mediana ponderada do conjunto de problemas j
 *                                     (1 <= j <= q)
 */

void comunic2_adiciona_tamanhos_particoes(Tam_particoes n_local_particoes[],
                                          Tam_particoes n_total_particoes[],
                                          Indice q)
{
    Indice j;
    if (comunic_proc_num != 1) {
        pvm_initsend(PvmDataDefault);
        for (j=1; j <= q; j++) {
            pvm_pkint(&n_local_particoes[j].menor, 1, 1);
            pvm_pkint(&n_local_particoes[j].menor_igual, 1, 1);
        }
        pvm_send(tids[0], 0);
    }
    else {
        Indice i;
        for (j=1; j <= q; j++) {
            n_total_particoes[j].menor=n_local_particoes[j].menor;
            n_total_particoes[j].menor_igual=n_local_particoes[j].menor_igual;
        }
        for (i=2; i <= comunic_p; i++) {
            Indice tam;
            pvm_recv(tids[i-1], 0);
            for (j=1; j <= q; j++) {
                pvm_upkint(&tam, 1, 1);
                n_total_particoes[j].menor+=tam;
                pvm_upkint(&tam, 1, 1);
                n_total_particoes[j].menor_igual+=tam;
            }
        }
    }
}

/*
 * comunic2_irradia_somas_tamanhos_particoes(n_total_particoes, q)
 *
 * Executa o passo 2.8 do algoritmo 2:
 *
 * O processador 1 envia o tamanho total das particoes de cada um dos q
 * conjuntos de problemas disjuntos para todos os demais processadores.
 */

```

```

* Retorno (em todos os processadores):
*     n_total_particoes[j] = tamanhos totais das particoes do conjunto de
*                             problemas j (1 <= j <= q)
*
*/

void comunic2_irradia_somas_tamanhos_particoes(Tam_particoes n_total_particoes[],
                                                Indice q)
{
    if (comunic_proc_num == 1) {
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&n_total_particoes[1].menor, 2*q, 1);
        pvm_mcast(&tids[1], comunic_p-1, 0);
    }
    else {
        pvm_recv(tids[0], 0);
        pvm_upkint(&n_total_particoes[1].menor, 2*q, 1);
    }
}

/*
* comunic2_redist_elementos_restantes(vetor_local, n_local, N,
*                                     vetor_alocacao_problema, vetor_aux2,
*                                     vetor_local2, q)
*
* Executa o passo 3 do algoritmo 2:
*
* Os elementos restantes de cada conjunto de problemas disjuntos sao
* remanejados entre os processadores. Para cada conjunto de problemas
* disjuntos j (1 <= j <= q), cada processador envia os seus elementos de
* "vetor_local[j][1..n_local[j]]" para o processador destino indicado por
* "vetor_alocacao_problema[j]", neste processador destino os problemas sao
* armazenados em areas contiguas de "vetor_aux2", indicadas por
* "vetor_local2[j][1..N[j]]".
*
* Retorno (apenas no processador vetor_alocacao_problema[j]):
*     vetor_local2[j][1..N[j]] = elementos restantes do conjunto de problemas j
*                                     (1 <= j <= q)
*
*/

void comunic2_redist_elementos_restantes(Elemento *vetor_local[],
                                          Indice n_local[], Indice N[],
                                          Indice vetor_alocacao_problema[],
                                          Elemento *vetor_aux2,
                                          Elemento *vetor_local2[], Indice q)
{
    Indice n_local2[MAX_Q+1];

    Elemento *vetor_final=vetor_aux2;
    Indice flag_receber=0;

    Indice i;
    for (i=1; i <= comunic_p; i++) {
        Indice flag_enviar=0;
        Indice j;
        for (j=1; j <= q; j++)
            if (vetor_alocacao_problema[j] == i)
                if (i == comunic_proc_num) {
                    Indice h;
                    flag_receber=1;
                    for (h=1; h <= n_local[j]; h++)
                        vetor_final[h]=vetor_local[j][h];
                    vetor_local2[j]=vetor_final;
                    n_local2[j]=n_local[j];
                    vetor_final+=N[j];
                }
        if (flag_enviar) {
            Indice k;
            for (k=1; k <= n_local[i]; k++)
                pvm_pkint(vetor_local[i][k], 1, 1);
            pvm_mcast(&tids[i], comunic_p-i, 0);
        }
    }

    if (flag_receber) {
        Indice k;
        for (k=1; k <= q; k++)
            pvm_recv(tids[k], N[k]);
    }
}

```

```

    }
    else {
        if (flag_enviar == 0) {
            pvm_initsend(PvmDataDefault);
            flag_enviar=1;
        }
        pvm_pkint(&n_local[j], 1, 1);
        if (n_local[j] > 0)
            pvm_pkdouble(&vetor_local[j][1], n_local[j], 1);
    }
    if (flag_enviar == 1)
        pvm_send(tids[i-1], 0);
}

if (flag_receber == 1)
    for (i=1; i <= comunic_p; i++)
        if (i != comunic_proc_num) {
            Indice j;
            pvm_recv(tids[i-1], 0);
            for (j=1; j <= q; j++)
                if (vetor_alocacao_problema[j] == comunic_proc_num) {
                    Indice tam;
                    pvm_upkint(&tam, 1, 1);
                    if (tam > 0) {
                        pvm_upkdouble(&vetor_local2[j][n_local2[j]+1], tam, 1);
                        n_local2[j]+=tam;
                    }
                }
        }
}

/*
* comunic2_reune_resultados(resultados, flag_result, q)
*
* Executado apos o passo 4 do algoritmo 2:
*
* Reune no processador 1 o resultado final dos q problemas de selecao
* finalizados em diversos processadores.
*
* O vetor "flag_result[j]" indica a validade de "resultados[j]" em cada
* processador j.
*
* Retorno (apenas no processador 1):
* resultados[j] = valor do k[j]-esimo elemento (1 <= j <= q).
*
*/

void comunic2_reune_resultados(Elemento resultados[], Indice flag_result[],
                               Indice q)
{
    Par_indice_elemento flag_resultados_aux[2*MAX_Q+1];

    if (comunic_proc_num != 1) {
        Indice j;
        pvm_initsend(PvmDataDefault);
        for (j=1; j <= q; j++) {
            pvm_pkdouble(&resultados[j], 1, 1);
            pvm_pkint(&flag_result[j], 1, 1);
        }
        pvm_send(tids[0], 0);
    }
    else {
        Indice i;
        for (i=2; i <= comunic_p; i++) {
            Indice j;
            pvm_recv(tids[i-1], 0);

```

```

        for (j=1; j <= q; j++) {
            pvm_upkdouble(&flag_resultados_aux[j].campo_elemento, 1, 1);
            pvm_upkint(&flag_resultados_aux[j].campo_indice, 1, 1);
            if (flag_resultados_aux[j].campo_indice != 0)
                resultados[j]=flag_resultados_aux[j].campo_elemento;
        }
    }
}

/*
 * comunic3_irradia_resultados(resultados, q)
 *
 * Executa o passo 5 do algoritmo 3:
 *
 * Cada processador i (1 <= i <= p) comunica as solucoes de seus problemas
 * finalizados localmente, de forma que todos os processadores conhecam
 * todos os k[j]-esimos elementos.
 *
 * Como o processador 1 ja' possui estas informacoes (pois a implementacao do
 * algoritmo 2 ja' realiza a comunicacao dos resultados para o processador 1
 * ao final da execucao). basta irradiar estes resultados a partir do
 * processador 1 para os demais processadores.
 *
 * Retorno (em todos os processadores):
 * resultados[j] = valor do k[j]-esimo elemento (1 <= j <= q).
 */

void comunic3_irradia_resultados(Elemento resultados[], Indice q)
{
    if (comunic_proc_num == 1) {
        pvm_initsend(PvmDataDefault);
        pvm_pkdouble(&resultados[1], q, 1);
        pvm_mcast(&tids[1], comunic_p-1, 0);
    }
    else {
        pvm_recv(tids[0], 0);
        pvm_upkdouble(&resultados[1], q, 1);
    }
}

/*
 * comunic3_reune_total_resultados(result_local_tam_particoes, q)
 *
 * Executa o passo 7 do algoritmo 3:
 *
 * Cada processador transfere para o processador 1 seu numero de elementos
 * locais menores, iguais e maiores do que cada k[j]-esimo elemento.
 *
 * Retorno (apenas no processador 1):
 * result_local_tam_particoes[j] = numero de elementos locais menores,
 * iguais e maiores ao k[j]-esimo elemento (1 <= j <= q).
 */

void comunic3_reune_total_resultados(Tam_particoes result_local_tam_particoes[],
                                     Indice q)
{
    if (comunic_proc_num != 1) {
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&result_local_tam_particoes[(comunic_proc_num-1)*q+1].menor,
                  2*q, 1);
        pvm_send(tids[0], 0);
    }
}

```

```

    }
    else {
        Indice i;
        for (i=2; i <= comunic_p; i++) {
            pvm_recv(tids[i-1], 0);
            pvm_upkint(&result_local_tam_particoes[(i-1)*q+1].menor, 2*q, 1);
        }
    }
}

/*
 * comunic3_redistribui_dados(quant_redist, vetor_local, vetor_aux, n_local)
 *
 * Executa os passos 9 e 10 do algoritmo 3:
 *
 * O processador 1 inicialmente irradia para os demais processadores as
 * informações acerca de quantos elementos serão realocados em cada caso,
 * em seguida estes elementos são realocados entre os processadores.
 *
 * Cada processador i envia para o processador j os elementos
 * "vetor_local[quant_redist[(i-1)*q+j-1]+1..quant_redist[(i-1)*q+j]]".
 *
 * Retorno (em cada processador j):
 *     vetor_local[1..n_local[j]] = elementos realocados recebidos, que
 *                                   serão ordenados localmente (1 <= j <= q).
 */

void comunic3_redistribui_dados(Indice quant_redist[], Elemento *vetor_local,
                                Elemento *vetor_aux, Indice *n_local)
{
    Indice i;
    Indice j;
    Indice quant_anterior;

    if (comunic_proc_num == 1)
        for (i=2; i <= comunic_p; i++) {
            pvm_initsend(PvmDataDefault);
            pvm_pkint(&quant_redist[(i-1)*comunic_p+1], comunic_p, 1);
            pvm_send(tids[i-1], 0);
        }
    else {
        pvm_recv(tids[0], 0);
        pvm_upkint(&quant_redist[(comunic_proc_num-1)*comunic_p+1], comunic_p, 1);
    }

    /* Cada proc. i envia os elementos alocados para outro processador j */

    quant_anterior=0;
    for (j=1; j <= comunic_p; j++) {
        Indice quant_atual=quant_redist[(comunic_proc_num-1)*comunic_p+j];
        Indice delta_quant=quant_atual-quant_anterior;
        if (j != comunic_proc_num) {
            pvm_initsend(PvmDataDefault);
            pvm_pkint(&delta_quant, 1, 1);
            if (delta_quant > 0)
                pvm_pkdouble(&vetor_local[quant_anterior+1], delta_quant, 1);
            pvm_send(tids[j-1], 0);
        }
        else
            *n_local=delta_quant;
        quant_anterior=quant_atual;
    }

    /* Cada proc. i move os elementos alocados no mesmo processador p/ início */

```

```

if (comunic_proc_num > 1) {
    Indice desl=
        quant_redist[(comunic_proc_num-1)*comunic_p+comunic_proc_num-1];
    if (desl > 0)
        for (i=1; i <= *n_local; i++)
            vetor_local[i]=vetor_local[desl+i];
}

/* Cada proc. i recebe os elementos alocados de outro processador */

for (i=1; i <= comunic_p; i++)
    if (i != comunic_proc_num) {
        Indice tam;
        pvm_recv(tids[i-1], 0);
        pvm_upkint(&tam, 1, 1);
        if (tam > 0) {
            pvm_upkdoube(&vetor_local[(*n_local)+1], tam, 1);
            *n_local+=tam;
        }
    }
}

/* FIM */

```

A.7 “alg_mpi.c”: Funções de comunicação em MPI

```
/*
 * Arquivo:      "alg_mpi.c"
 *
 * Conteudo:     Funcoes de comunicacao implementadas em MPI.
 *
 *               Neste arquivo estao isoladas todas as funcoes cuja implementacao
 *               depende da interface de comunicacao escolhida.
 *
 * Trabalho:     "Algoritmos de Selecao para Maquinas
 *               Paralelas com Memoria Distribuida"
 *
 * Autor:        Einar Luciano Gattoni Saukas
 *
 * Orientador:   Prof. Dr. Siang W. Song
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"
#include "alg_seq.h"
#include "alg_com.h"

/* Variaveis globais (utilizadas apenas no escopo deste arquivo) */

int comunic_p=0;                /* Numero de processadores */
int comunic_proc_num=0;         /* Indice deste processador */
MPI_Datatype comunic_tipo_indice_elemento; /* Tipo "Par_indice_elemento" */
MPI_Op comunic_op_reducao_resultado; /* Funcao "comunic_reducao_resultado()" */

/*
 * marca_cronometro(tempo)
 *
 * Calcula o intervalo de tempo decorrido desde a ultima chamada a esta funcao,
 * somando este valor ao parametro "tempo".
 *
 * Como esta rotina determina intervalos de tempo entre chamadas consecutivas
 * (ao inves de tempos absolutos), e' necessario realizar uma primeira
 * chamada para "iniciar" a rotina.
 *
 * Se for utilizado um ponteiro nulo no parametro "tempo", a rotina e' iniciada
 * sem retornar o intervalo de tempo.
 *
 * O exemplo a seguir ilustra a utilizacao desta funcao na cronometragem de um
 * conjunto de rotinas A, B e C:
 *
 * unsigned int tempo1=0;
 * unsigned int tempo2=0;
 * ...
 * marca_cronometro(0);           // inicia a rotina
 * A();
 * marca_cronometro(&tempo1)      // tempo1= <duracao de A>
 * B();
 * marca_cronometro(&tempo2);     // tempo2= <duracao de B>
 * C();
 * marca_cronometro(&tempo1);     // tempo1= <duracao de A> + <duracao de C>
 */
```

```

void marca_cronometro(unsigned int *tempo)
{
    static double tempo_anterior=0;
    double tempo_atual=MPI_Wtime();
    if (tempo)
        *tempo+=(unsigned)((tempo_atual-tempo_anterior)*1e6);
    tempo_anterior=tempo_atual;
}

/*
 * erro(msg)
 *
 * Imprime a mensagem de erro "msg" e aborta a execucao do programa.
 */

void erro(const char *msg)
{
    fprintf(stderr, "ERRO: %s\n", msg);
    exit(-1);
}

/*
 * comunic_reducao_resultado(entrada, saida, quant, tipo)
 *
 * Operacao binaria de "combinacao" de resultados, utilizada pela rotina
 * comunic2_reune_resultados(..) para reunir em um unico vetor do proc. 1
 * todos os resultados calculados por processadores diferentes.
 */

void comunic_reducao_resultado(void *entrada, void *saida, int *quant,
                               MPI_Datatype *tipo)
{
    int i;
    for (i=0; i < *quant; i++)
        if (((Par_indice_elemento *)entrada)[i].campo_indice != 0) {
            ((Par_indice_elemento *)saida)[i].campo_indice=
                ((Par_indice_elemento *)entrada)[i].campo_indice;
            ((Par_indice_elemento *)saida)[i].campo_elemento=
                ((Par_indice_elemento *)entrada)[i].campo_elemento;
        }
}

/*
 * comunic_inicio(prog, argc, argv, p, proc_num)
 *
 * Inicia o programa nos diversos processadores e, em cada processador,
 * determina o indice deste processador e o numero total de processadores.
 *
 * Esta rotina deve ser chamada no inicio do programa, antes de qualquer outra
 * rotina de comunicacao.
 *
 * O parametro "prog" deveria conter o nome do executavel a ser carregado em
 * cada processador. No caso da interface MPI, este parametro nao e'
 * necessario, sendo aqui mantido apenas para compatibilidade com as outras
 * implementacoes.
 *
 * Os parametros "argc" e "argv" sao os mesmos recebidos pela funcao "main()",
 * o que garante que todos os processadores sejam iniciados com os mesmos
 * argumentos da linha de comando.
 *
 * Retorno: p          = numero de processadores (p >= 1)
 */

```



```

*           proc_num = indice deste processador (1 <= proc_num <= p)
*
*/

void comunic_inicio(char *prog, int argc, char *argv[], int *p, int *proc_num)
{
    int          comp[3];
    MPI_Aint     desl[3];
    MPI_Datatype tipo[3];

    Par_indice_elemento aux;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &comunic_proc_num);
    comunic_proc_num++;
    MPI_Comm_size(MPI_COMM_WORLD, &comunic_p);

    comp[0]=1;
    desl[0]=0;
    tipo[0]=MPI_INT;
    comp[1]=1;
    desl[1]=((char *)&aux.campo_elemento)-((char *)&aux.campo_indice);
    tipo[1]=MPI_DOUBLE;
    comp[2]=1;
    desl[2]=sizeof(aux);
    tipo[2]=MPI_UB;

    MPI_Type_struct(3, comp, desl, tipo, &comunic_tipo_indice_elemento);
    MPI_Type_commit(&comunic_tipo_indice_elemento);

    MPI_Op_create(comunic_reducao_resultado, 1, &comunic_op_reducao_resultado);

    *p=comunic_p;
    *proc_num=comunic_proc_num;
}

/*
* comunic_sincroniza()
*
*   Sincroniza todos os processadores.
*
*   Esta rotina e' chamada para garantir que todos os processadores estejam
*   prontos para iniciar a execucao dos algoritmos.
*
*/

void comunic_sincroniza(void)
{
    MPI_Barrier(MPI_COMM_WORLD);
}

/*
* comunic_final()
*
*   Finaliza o programa, liberando os recursos alocados para as rotinas de
*   comunicacao.
*
*   Esta rotina deve ser chamada no final do programa, apos a execucao das
*   demais rotinas de comunicacao.
*
*/

void comunic_final(void)
{
    MPI_Op_free(&comunic_op_reducao_resultado);
}

```

```

MPI_Type_free(&comunic_tipo_indice_elemento);
MPI_Finalize();
}

/*
 * comunic1_reune_medianas_locais(n_local, mediana_local, vetor_n_mediana_local)
 *
 * Executa o passo 2.2 do algoritmo 1:
 *
 * Cada processador envia, para o processador 1, sua mediana local e numero de
 * elementos restantes.
 *
 * Retorno (apenas no processador 1):
 *     vetor_n_mediana_local[i].campo_indice = numero de elem. no proc. i
 *     vetor_n_mediana_local[i].campo_elemento = mediana local do proc. i
 *                                           (1 <= i <= p)
 */
void comunic1_reune_medianas_locais(Indice n_local, Elemento mediana_local,
                                     Par_indice_elemento vetor_n_mediana_local[])
{
    vetor_n_mediana_local[comunic_proc_num].campo_indice=n_local;
    vetor_n_mediana_local[comunic_proc_num].campo_elemento=mediana_local;
    MPI_Gather(&vetor_n_mediana_local[comunic_proc_num], 1,
              comunic_tipo_indice_elemento, &vetor_n_mediana_local[1], 1,
              comunic_tipo_indice_elemento, 0, MPI_COMM_WORLD);
}

/*
 * comunic1_irradia_mediana_ponderada(mediana_ponderada)
 *
 * Executa o passo 2.4 do algoritmo 1:
 *
 * O processador 1 envia a mediana ponderada para todos os demais
 * processadores.
 *
 * Retorno (em todos os processadores):
 *     mediana_ponderada = mediana ponderada dos elementos restantes
 */
void comunic1_irradia_mediana_ponderada(Elemento *mediana_ponderada)
{
    MPI_Bcast(mediana_ponderada, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

/*
 * comunic1_adiciona_tamanhos_particoes(n_local_particoes, n_total_particoes)
 *
 * Executa os passos 2.6 e 2.7 do algoritmo 1:
 *
 * Cada processador envia os tamanhos de suas particoes locais (numero de
 * elementos locais menores, iguais e maiores que a mediana ponderada), o
 * processador 1 calcula as suas somatorias.
 *
 * Na verdade, a implementacao do algoritmo trabalha apenas com o numero de
 * elementos locais menores e o numero de elementos locais menores ou iguais,
 * o que apresenta algumas vantagens do ponto de vista pratico.
 *
 * Retorno (apenas no processador 1):
 *     n_total_particoes->menor          = numero de elementos estritamente
 *                                     menores que a mediana ponderada
 *     n_total_particoes->menor_igual    = numero de elementos menores ou
 */

```

```

*                                     iguais que a mediana ponderada
*
*/

void comunic1_adiciona_tamanhos_particoes(Tam_particoes n_local_particoes,
                                           Tam_particoes *n_total_particoes)
{
    MPI_Reduce(&n_local_particoes, n_total_particoes, 2, MPI_INT, MPI_SUM, 0,
              MPI_COMM_WORLD);
}

/*
* comunic1_irradia_somas_tamanhos_particoes(n_total_particoes)
*
* Executa o passo 2.8 do algoritmo 1:
*
* O processador 1 envia o tamanho total das particoes para todos os demais
* processadores.
*
* Retorno (em todos os processadores):
*     n_total_particoes = tamanhos totais das particoes
*
*/

void comunic1_irradia_somas_tamanhos_particoes(Tam_particoes *n_total_particoes)
{
    MPI_Bcast(n_total_particoes, 2, MPI_INT, 0, MPI_COMM_WORLD);
}

/*
* comunic1_reune_elementos_restantes(n_local, vetor_local, n_total,
*                                   vetor_total)
*
* Executa o passo 3 do algoritmo 1:
*
* Cada processador envia todos os seus elementos restantes (em
* "vetor_local[1..n_local]") para o processador 1.
*
* Retorno (apenas no processador 1):
*     vetor_total[1..n_total] = elementos restantes de todos os proc.
*
*/

void comunic1_reune_elementos_restantes(Indice n_local, Elemento *vetor_local,
                                         Indice n_total, Elemento *vetor_total)
{
    if (comunic_proc_num != 1)
        MPI_Send(&vetor_local[1], n_local, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    else {
        Indice i;
        if (vetor_total != vetor_local)
            for (i=1; i <= n_local; i++)
                vetor_total[i]=vetor_local[i];
        for (i=2; i <= comunic_p; i++) {
            MPI_Status status;
            int tam;
            MPI_Recv(&vetor_total[n_local+1], n_total-n_local, MPI_DOUBLE, i-1, 0,
                    MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_DOUBLE, &tam);
            n_local+=tam;
        }
    }
}

```

```

/*
 * comunic2_reune_medianas_locais(n_local, mediana_local, vetor_n_mediana_local,
 *                                q)
 *
 * Executa o passo 2.2 do algoritmo 2:
 *
 * Cada processador envia, para o processador 1, a mediana local e numero
 * de elementos restantes correspondentes a cada um dos q conjuntos de
 * problemas disjuntos.
 *
 * Retorno (apenas no processador 1):
 *     vetor_n_mediana_local[(i-1)*q+j].campo_indice = numero de elem. do
 *     proc. i e problema j (1 <= i <= p, 1 <= j <= q)
 *     vetor_n_mediana_local[(i-1)*q+j].campo_elemento = mediana local do
 *     proc. i e problema j (1 <= i <= p, 1 <= j <= q)
 */

void comunic2_reune_medianas_locais(Indice n_local[], Elemento mediana_local[],
                                    Par_indice_elemento vetor_n_mediana_local[],
                                    Indice q)
{
    Indice j;
    for (j=1; j <= q; j++) {
        vetor_n_mediana_local[(comunic_proc_num-1)*q+j].campo_indice=n_local[j];
        vetor_n_mediana_local[(comunic_proc_num-1)*q+j].campo_elemento=
            mediana_local[j];
    }
    MPI_Gather(&vetor_n_mediana_local[(comunic_proc_num-1)*q+1], q,
              comunic_tipo_indice_elemento, &vetor_n_mediana_local[1], q,
              comunic_tipo_indice_elemento, 0, MPI_COMM_WORLD);
}

/*
 * comunic2_irradia_mediana_ponderada(mediana_ponderada, q)
 *
 * Executa o passo 2.4 do algoritmo 2:
 *
 * O processador 1 envia as medianas ponderadas de cada um dos q conjuntos de
 * problemas disjuntos para todos os demais processadores.
 *
 * Retorno (em todos os processadores):
 *     mediana_ponderada[j] = mediana ponderada do conjunto j de problemas
 *                           (1 <= j <= q)
 */

void comunic2_irradia_mediana_ponderada(Elemento mediana_ponderada[], Indice q)
{
    MPI_Bcast(&mediana_ponderada[1], q, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

/*
 * comunic2_adiciona_tamanhos_particoes(n_local_particoes, n_total_particoes, q)
 *
 * Executa os passos 2.6 e 2.7 do algoritmo 2:
 *
 * Cada processador envia os tamanhos de suas particoes locais (numero de
 * elementos locais menores, iguais e maiores que a mediana ponderada) de
 * cada um dos q conjuntos de problemas disjuntos, o processador 1 calcula
 * as suas somatorias.
 *
 * Na verdade, a implementacao do algoritmo trabalha apenas com o numero de
 * elementos locais menores e o numero de elementos locais menores ou iguais,
 * o que apresenta algumas vantagens do ponto de vista pratico.
 */

```

```

*
* Retorno (apenas no processador 1):
*     n_total_particoes[j].menor           = numero de elementos estritamente
*     menores que a mediana ponderada do conjunto de problemas j
*     n_total_particoes[j].menor_igual = numero de elementos menores ou
*     iguais que a mediana ponderada do conjunto de problemas j
*                                     (1 <= j <= q)
*
*/

void comunic2_adiciona_tamanhos_particoes(Tam_particoes n_local_particoes[],
                                          Tam_particoes n_total_particoes[],
                                          Indice q)
{
    MPI_Reduce(&n_local_particoes[1], &n_total_particoes[1], 2*q, MPI_INT,
              MPI_SUM, 0, MPI_COMM_WORLD);
}

/*
* comunic2_irradia_somas_tamanhos_particoes(n_total_particoes, q)
*
* Executa o passo 2.8 do algoritmo 2:
*
* O processador 1 envia o tamanho total das particoes de cada um dos q
* conjuntos de problemas disjuntos para todos os demais processadores.
*
* Retorno (em todos os processadores):
*     n_total_particoes[j] = tamanhos totais das particoes do conjunto de
*     problemas j (1 <= j <= q)
*
*/

void comunic2_irradia_somas_tamanhos_particoes(Tam_particoes n_total_particoes[],
                                              Indice q)
{
    MPI_Bcast(&n_total_particoes[1], 2*q, MPI_INT, 0, MPI_COMM_WORLD);
}

/*
* comunic2_redist_elementos_restantes(vetor_local, n_local, N,
*                                     vetor_alocacao_problema, vetor_aux2,
*                                     vetor_local2, q)
*
* Executa o passo 3 do algoritmo 2:
*
* Os elementos restantes de cada conjunto de problemas disjuntos sao
* remanejados entre os processadores. Para cada conjunto de problemas
* disjuntos j (1 <= j <= q), cada processador envia os seus elementos de
* "vetor_local[j][1..n_local[j]]" para o processador destino indicado por
* "vetor_alocacao_problema[j]", neste processador destino os problemas sao
* armazenados em areas contiguas de "vetor_aux2", indicadas por
* "vetor_local2[j][1..N[j]]".
*
* Retorno (apenas no processador vetor_alocacao_problema[j]):
*     vetor_local2[j][1..N[j]] = elementos restantes do conjunto de problemas j
*                                     (1 <= j <= q)
*
*/

void comunic2_redist_elementos_restantes(Elemento *vetor_local[],
                                          Indice n_local[], Indice N[],
                                          Indice vetor_alocacao_problema[],
                                          Elemento *vetor_aux2,
                                          Elemento *vetor_local2[], Indice q)
{

```

```

int comp_tot[MAX_P*MAX_Q+1];
int comp[MAX_P+1];
int desl[MAX_P+1];

Elemento *vetor_final=vetor_aux2;
Indice i;
Indice j;

MPI_Allgather(&n_local[1], q, MPI_INT, &comp_tot[1], q, MPI_INT,
              MPI_COMM_WORLD);
for (j=1; j <= q; j++) {
    if (vetor_alocacao_problema[j] == comunic_proc_num) {
        desl[1]=0;
        comp[1]=comp_tot[j];
        for (i=2; i <= comunic_p; i++) {
            comp[i]=comp_tot[(i-1)*q+j];
            desl[i]=desl[i-1]+comp[i-1];
        }
        vetor_local2[j]=vetor_final;
        vetor_final+=N[j];
    }
    MPI_Gatherv(&vetor_local[j][1], n_local[j], MPI_DOUBLE,
               &vetor_local2[j][1], &comp[1], &desl[1], MPI_DOUBLE,
               vetor_alocacao_problema[j]-1, MPI_COMM_WORLD);
}
}

/*
 * comunic2_reune_resultados(resultados, flag_result, q)
 *
 * Executado apos o passo 4 do algoritmo 2:
 *
 * Reune no processador 1 o resultado final dos q problemas de selecao
 * finalizados em diversos processadores.
 *
 * O vetor "flag_result[j]" indica a validade de "resultados[j]" em cada
 * processador j.
 *
 * Retorno (apenas no processador 1):
 * resultados[j] = valor do k[j]-esimo elemento (1 <= j <= q).
 */

void comunic2_reune_resultados(Elemento resultados[], Indice flag_result[],
                               Indice q)
{
    Par_indice_elemento flag_resultados_aux[2*MAX_Q+1];

    Indice j;
    for (j=1; j <= q; j++) {
        flag_resultados_aux[j].campo_indice=flag_result[j];
        flag_resultados_aux[j].campo_elemento=resultados[j];
    }
    MPI_Reduce(&flag_resultados_aux[1], &flag_resultados_aux[q+1], q,
               comunic_tipo_indice_elemento, comunic_op_reducao_resultado, 0,
               MPI_COMM_WORLD);
    for (j=1; j <= q; j++)
        if (flag_resultados_aux[q+j].campo_indice != 0)
            resultados[j]=flag_resultados_aux[q+j].campo_elemento;
}

/*
 * comunic3_irradia_resultados(resultados, q)
 *
 * Executa o passo 5 do algoritmo 3:

```

```

*
* Cada processador i ( $1 \leq i \leq p$ ) comunica as solucoes de seus problemas
* finalizados localmente, de forma que todos os processadores conhecam
* todos os  $k[j]$ -esimos elementos.
*
* Como o processador 1 ja' possui estas informacoes (pois a implementacao do
* algoritmo 2 ja' realiza a comunicacao dos resultados para o processador 1
* ao final da execucao). basta irradiar estes resultados a partir do
* processador 1 para os demais processadores.
*
* Retorno (em todos os processadores):
* resultados[j] = valor do  $k[j]$ -esimo elemento ( $1 \leq j \leq q$ ).
*
*/

void comunic3_irradia_resultados(Elemento resultados[], Indice q)
{
    MPI_Bcast(&resultados[1], q, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

/*
* comunic3_reune_total_resultados(result_local_tam_particoes, q)
*
* Executa o passo 7 do algoritmo 3:
*
* Cada processador transfere para o processador 1 seu numero de elementos
* locais menores, iguais e maiores do que cada  $k[j]$ -esimo elemento.
*
* Retorno (apenas no processador 1):
* result_local_tam_particoes[j] = numero de elementos locais menores,
* iguais e maiores ao  $k[j]$ -esimo elemento ( $1 \leq j \leq q$ ).
*
*/

void comunic3_reune_total_resultados(Tam_particoes result_local_tam_particoes[],
                                     Indice q)
{
    MPI_Gather(&result_local_tam_particoes[(comunic_proc_num-1)*q+1].menor, 2*q,
              MPI_INT, &result_local_tam_particoes[1].menor, 2*q, MPI_INT, 0,
              MPI_COMM_WORLD);
}

/*
* comunic3_redistribui_dados(quant_redist, vetor_local, vetor_aux, n_local)
*
* Executa os passos 9 e 10 do algoritmo 3:
*
* O processador 1 inicialmente irradia para os demais processadores as
* informacoes acerca de quantos elementos serao realocados em cada caso,
* em seguida estes elementos sao realocados entre os processadores.
*
* Cada processador i envia para o processador j os elementos
* "vetor_local[quant_redist[(i-1)*q+j-1]+1..quant_redist[(i-1)*q+j]]".
*
* Retorno (em cada processador j):
* vetor_local[1..n_local[j]] = elementos realocados recebidos, que
* serao ordenados localmente ( $1 \leq j \leq q$ ).
*
*/

void comunic3_redistribui_dados(Indice quant_redist[], Elemento *vetor_local,
                                Elemento *vetor_aux, Indice *n_local)
{
    Indice i;
    Indice j;

```

```

int envia_comp[MAX_P+1];
int envia_dest[MAX_P+1];
int recebe_comp[MAX_P+1];
int recebe_desl[MAX_P+1];

Indice quant_anterior;
Indice quant_atual;
Indice delta_quant;

/* Proc. 1 transmite para os demais processadores a divisao dos elementos */
MPI_Bcast(&quant_redist[1], comunic_p*comunic_p, MPI_INT, 0, MPI_COMM_WORLD);

/* Cada proc. i envia os elementos alocados para outro processador j: */
/* Calcula informacoes de elementos a serem transmitidos para outros proc. */

quant_anterior=0;
for (j=1; j <= comunic_p; j++) {
    quant_atual=quant_redist[(comunic_proc_num-1)*comunic_p+j];
    delta_quant=quant_atual-quant_anterior;
    envia_comp[j]=delta_quant;
    envia_dest[j]=quant_anterior;
    quant_anterior=quant_atual;
}

/* Calcula informacoes de elementos a serem recebidos de outros proc. */

*n_local=0;
for (i=1; i <= comunic_p; i++) {
    if (comunic_proc_num > 1)
        quant_anterior=quant_redist[(i-1)*comunic_p+comunic_proc_num-1];
    else
        quant_anterior=0;
    quant_atual=quant_redist[(i-1)*comunic_p+comunic_proc_num];
    delta_quant=quant_atual-quant_anterior;
    recebe_comp[i]=delta_quant;
    recebe_desl[i]=*n_local;
    (*n_local)+=delta_quant;
}

/* Realoca elementos, copiando-os para area temporaria do proc. destino */
MPI_Alltoallv(&vetor_local[1], &envia_comp[1], &envia_dest[1], MPI_DOUBLE,
              &vetor_aux[1], &recebe_comp[1], &recebe_desl[1], MPI_DOUBLE,
              MPI_COMM_WORLD);

/* Copia elementos recebidos na area temporaria novamente para vetor_local */

for (i=1; i <= *n_local; i++)
    vetor_local[i]=vetor_aux[i];
}

/* FIM */

```


A.8 “alg_tst1.c”: Exemplo de utilização do algoritmo 1

```
/*
 * Arquivo:      "alg_tst1.c"
 *
 * Conteudo:     Programa exemplo de utilizacao do algoritmo 1.
 *
 * Trabalho:     "Algoritmos de Selecao para Maquinas
 *               Paralelas com Memoria Distribuida"
 *
 * Autor:        Einar Luciano Gattoni Saukas
 *
 * Orientador:   Prof. Dr. Siang W. Song
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "alg_seq.h"
#include "alg_com.h"
#include "alg_par.h"

/*
 * main(argc, argv)
 *
 * Exemplo de utilizacao do algoritmo 1.
 *
 * Cria um vetor de entrada com 1M elementos escolhidos aleatoriamente
 * (distribuicao uniforme dos elementos de entrada, que corresponde ao
 * "Caso Medio"), executa o algoritmo 1 de selecao e imprime o tempo total
 * de execucao.
 */

void main(int argc, char *argv[])
{
    Indice n=1048576;
    Indice c=100;
    Indice k=n/3+1;

    Indice n_local;
    Elemento *vetor_entrada;
    Elemento *vetor_aux;

    int p;
    int proc_num;

    unsigned int tempo;

    Elemento resultado;
    Indice i;

    comunic_inicio("alg1", argc, argv, &p, &proc_num);
    if (p < 1 || p > MAX_P)
        erro("p");

    n_local=n/p;
    if (n%p >= proc_num)
        n_local++;

    vetor_entrada=(Elemento *)malloc((n_local+1)*sizeof(Elemento));
    vetor_aux=(Elemento *)malloc((n_local/2+2)*sizeof(Elemento));
```

```

if (!vetor_entrada || !vetor_aux)
    erro("malloc");

for (i=1; i <= n_local; i++)
    vetor_entrada[i]=(Elemento)rand();

zera_cronometro();
tempo=0;
comunic_sincroniza();
marca_cronometro(0);
resultado=algoritmo1(vetor_entrada, n, p, k, vetor_aux, proc_num, c);
marca_cronometro(&tempo);

if (proc_num == 1)
    printf("\nAlgoritmo 1: Analise de \"Caso Medio\", "
           "N=%8d, P=%d, K=N/3+1=%8d, C=%d\n"
           "Resultado=%g, Tempo=%10u\n", n, p, k, c, resultado, tempo);

    comunic_final();
}

/* FIM */

```

A.9 “alg_tst2.c”: Exemplo de utilização do algoritmo 2

```
/*
 * Arquivo:      "alg_tst2.c"
 *
 * Conteudo:     Programa exemplo de utilizacao do algoritmo 2.
 *
 * Trabalho:     "Algoritmos de Selecao para Maquinas
 *               Paralelas com Memoria Distribuida"
 *
 * Autor:        Einar Luciano Gattoni Saukas
 *
 * Orientador:   Prof. Dr. Siang W. Song
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "alg_seq.h"
#include "alg_com.h"
#include "alg_par.h"

/*
 * main(argc, argv)
 *
 * Exemplo de utilizacao do algoritmo 2.
 *
 * Cria um vetor de entrada com 1M elementos escolhidos aleatoriamente
 * (distribuicao uniforme dos elementos de entrada, que corresponde ao
 * "Caso Medio"), executa o algoritmo 2 de selecao e imprime o tempo total
 * de execucao.
 */

void main(int argc, char *argv[])
{
    Indice n=1048576;
    Indice c=100;
    Indice q=16;
    Indice k[MAX_Q+1];

    Indice n_local;
    Elemento *vetor_entrada;
    Elemento *vetor_aux;
    Elemento *vetor_aux2;

    int p;
    int proc_num;

    unsigned int tempo;

    Elemento resultados[MAX_Q+1];
    Indice i;
    Indice j;

    comunic_inicio("alg2", argc, argv, &p, &proc_num);
    if (p < 1 || p > MAX_P)
        erro("p");

    n_local=n/p;
    if (n%p >= proc_num)
        n_local++;
}
```

```

vetor_entrada=(Elemento *)malloc((n_local+1)*sizeof(Elemento));
vetor_aux=(Elemento *)malloc((n_local/2+2)*sizeof(Elemento));
vetor_aux2=(Elemento *)malloc((2*(n_local+1))*sizeof(Elemento));
if (!vetor_entrada || !vetor_aux || !vetor_aux2)
    erro("malloc");

for (j=1; j <= q; j++)
    k[j]=2*j*n/(3*q);

for (i=1; i <= n_local; i++)
    vetor_entrada[i]=(Elemento)rand();

zera_cronometro();
tempo=0;
comunic_sincroniza();
marca_cronometro(0);
algoritmo2(vetor_entrada, n, p, k, q, vetor_aux, vetor_aux2, proc_num, c,
    resultados);
marca_cronometro(&tempo);

if (proc_num == 1)
    printf("\nAlgoritmo 2: Analise de \"Caso Medio\", "
        "N=%8d, P=%d, Q=%d, K=j2N/3Q, C=%d\n"
        "Tempo=%10u\n", n, p, q, c, tempo);

    comunic_final();
}

/* FIM */

```

A.10 “alg_tst3.c”: Exemplo de utilização do algoritmo 3

```
/*
 * Arquivo:      "alg_tst3.c"
 *
 * Conteudo:     Programa exemplo de utilizacao do algoritmo 3.
 *
 * Trabalho:     "Algoritmos de Selecao para Maquinas
 *               Paralelas com Memoria Distribuida"
 *
 * Autor:        Einar Luciano Gattoni Saukas
 *
 * Orientador:   Prof. Dr. Siang W. Song
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "alg_seq.h"
#include "alg_com.h"
#include "alg_par.h"

/*
 * main(argc, argv)
 *
 * Exemplo de utilizacao do algoritmo 3.
 *
 * Cria um vetor de entrada com 1M elementos escolhidos aleatoriamente
 * (distribuicao uniforme dos elementos de entrada, que corresponde ao
 * "Caso Medio"), executa o algoritmo 3 de ordenacao e imprime o tempo total
 * de execucao.
 *
 */

void main(int argc, char *argv[])
{
    Indice n=1048576;
    Indice c=100;

    Indice n_local;
    Elemento *vetor_entrada;
    Elemento *vetor_aux;
    Elemento *vetor_aux2;

    int p;
    int proc_num;

    unsigned int tempo;

    Elemento resultados[MAX_P];
    Indice i;

    comunic_inicio("alg2", argc, argv, &p, &proc_num);
    if (p < 1 || p > MAX_P)
        erro("p");

    n_local=n/p;
    if (n%p >= proc_num)
        n_local++;

    vetor_entrada=(Elemento *)malloc((n_local+1)*sizeof(Elemento));
    vetor_aux=(Elemento *)malloc((n_local/2+2)*sizeof(Elemento));
```

```

vetor_aux2=(Elemento *)malloc((2*(n_local+1))*sizeof(Elemento));
if (!vetor_entrada || !vetor_aux || !vetor_aux2)
    erro("malloc");

for (i=1; i <= n_local; i++)
    vetor_entrada[i]=(Elemento)rand();

zera_cronometro();
tempo=0;
comunic_sincroniza();
marca_cronometro(0);
algoritmo3(vetor_entrada, n, p, vetor_aux, vetor_aux2, proc_num, c,
    resultados);
marca_cronometro(&tempo);

if (proc_num == 1)
    printf("\nAlgoritmo 3: Analise de \"Caso Medio\", N=%8d, P=%d, C=%d\n"
        "Tempo=%10u\n", n, p, c, tempo);

comunic_final();
}

/* FIM */

```

APÊNDICE B

GRÁFICOS DE DESEMPENHO

Alg. 1 PVM: Caso Médio, N=1048576

Alg. 1 PVM: Caso Especial, N=1048576

Alg. 1 PVM: Pior Caso, N=1048576

Alg. 1 PVM: Pior Caso Randômico, N=1048576

Fig. 1: Tempo total em função do parâmetro C, para N fixo (pior tempo em 20 execuções).

Alg. 1 PVM: Caso Médio, N=1048576

Alg. 1 PVM: Caso Especial, N=1048576

Alg. 1 PVM: Pior Caso, N=1048576

Alg. 1 PVM: Pior Caso Randômico, N=1048576

Fig. 2: Número de elementos restantes (ao final do passo 2) em função do parâmetro C, para N fixo.

Alg. 1 PVM: Caso Médio, N=1048576

Alg. 1 PVM: Caso Especial, N=1048576

Alg. 1 PVM: Pior Caso, N=1048576

Alg. 1 PVM: Pior Caso Randômico, N=1048576

Fig. 3: Número de rodadas de execução em função do parâmetro C, para N fixo.

Alg. 1 PVM: Caso Médio, P=2, N=1048576

Alg. 1 PVM: Caso Especial, P=2, N=1048576

Alg. 1 PVM: Pior Caso, P=2, N=1048576

Alg. 1 PVM: Pior Caso Randômico, P=2, N=1048576

Fig. 4: Tempos parciais em função do parâmetro C para P=2.

Alg. 1 PVM: Caso Médio, P=16, N=1048576

Alg. 1 PVM: Caso Especial, P=16, N=1048576

Alg. 1 PVM: Pior Caso, P=16, N=1048576

Alg. 1 PVM: Pior Caso Randômico, P=16, N=1048576

Fig. 5: Tempos parciais em função do parâmetro C para P=16.

Alg. 1 PVM: P=2, N=1048576

Alg. 1 PVM: P=16, N=1048576

Alg. 1 PVM: P=2, N=131072

Alg. 1 PVM: P=16, N=131072

Fig. 6: Tempo total em função do parâmetro C, para diversos valores de N e P (pior tempo em 20 execuções).

Alg. 1 PVM: Caso Médio, C=100

Alg. 1 PVM: Caso Especial, C=100

Alg. 1 PVM: Pior Caso, C=100

Alg. 1 PVM: Pior Caso Randômico, C=100

Fig. 7: Tempo total em função do número de processadores, para diversos valores de N (pior tempo em 20 execuções).

Fig. 8: Tempo total em função do número de processadores, para diversos valores de N (pior tempo em 20 execuções).

Alg. 1 PVM: Caso Médio, C=100

Alg. 1 PVM: Caso Especial, C=100

Alg. 1 PVM: Pior Caso, C=100

Alg. 1 PVM: Pior Caso Randômico, C=100

Fig. 9: Speedup em função do número de processadores, para diversos valores de N (pior tempo em 20 execuções).

Alg. 1 PVM: Caso Médio, C=100

Alg. 1 PVM: Caso Especial, C=100

Alg. 1 PVM: Pior Caso, C=100

Alg. 1 PVM: Pior Caso Randômico, C=100

Fig. 10: Tempo total em função do número de elementos (pior tempo em 20 execuções).

Alg. 2 PVM: Caso Médio, C=100, Q=16

Alg. 2 PVM: Caso Especial, C=100, Q=16

Alg. 2 PVM: Pior Caso, C=100, Q=16

Alg. 2 PVM: Pior Caso Randômico, C=100, Q=16

Fig. 11: Tempo total em função do número de processadores, para diversos valores de N (pior tempo em 20 execuções).

Alg. 2 PVM: Caso Médio, C=100, Q=16

Alg. 2 PVM: Caso Especial, C=100, Q=16

Alg. 2 PVM: Pior Caso, C=100, Q=16

Alg. 2 PVM: Pior Caso Randômico, C=100, Q=16

Fig. 12: Tempo total em função do número de processadores, para diversos valores de N (pior tempo em 20 execuções).

Alg. 2 PVM: Caso Médio, C=100, Q=16

Alg. 2 PVM: Caso Especial, C=100, Q=16

Alg. 2 PVM: Pior Caso, C=100, Q=16

Alg. 2 PVM: Pior Caso Randômico, C=100, Q=16

Fig. 13: Speedup em função do número de processadores, para diversos valores de N (pior tempo em 20 execuções).

Alg. 2 PVM: Caso Médio, C=100, Q=16

Alg. 2 PVM: Caso Especial, C=100, Q=16

Alg. 2 PVM: Pior Caso, C=100, Q=16

Alg. 2 PVM: Pior Caso Randômico, C=100, Q=16

Fig. 14: Tempo total em função do número de elementos (pior tempo em 20 execuções).

Alg. 2 PVM: Caso Médio, C=100, N=1048576

Alg. 2 PVM: Caso Especial, C=100, N=1048576

Alg. 2 PVM: Pior Caso, C=100, N=1048576

Alg 2 PVM: Pior Caso Randômico, C=100, N=1048576

Fig. 15: Tempo total em função do número de processadores, para diversos valores de Q (pior tempo em 20 execuções).

Alg. 2 PVM: Caso Médio, C=100, N=1048576

Alg. 2 PVM: Caso Especial, C=100, N=1048576

Alg. 2 PVM: Pior Caso, C=100, N=1048576

Alg. 2 PVM: Pior Caso Randômico, C=100, N=1048576

Fig. 16: Tempo total em função do número de processadores, para diversos valores de Q (pior tempo em 20 execuções).

Alg. 2 PVM: Caso Médio, C=100, N=1048576

Alg. 2 PVM: Caso Especial, C=100, N=1048576

Alg. 2 PVM: Pior Caso, C=100, N=1048576

Alg. 2 PVM: Pior Caso Randômico, C=100, N=1048576

Fig. 17: Speedup em função do número de processadores, para diversos valores de N (pior tempo em 20 execuções).

Fig. 18: Tempo total em função do número de problemas (pior tempo em 20 execuções).

Alg. 3 PVM: Caso Médio, C=100

Alg. 3 PVM: Caso Especial, C=100

Alg. 3 PVM: Pior Caso, C=100

Alg. 3 PVM: Pior Caso Randômico, C=100

Fig. 19: Tempo total em função do número de processadores, para diversos valores de N (pior tempo em 20 execuções).

Alg. 3 PVM: Caso Médio, C=100

Alg. 3 PVM: Caso Especial, C=100

Alg. 3 PVM: Pior Caso, C=100

Alg. 3 PVM: Pior Caso Randômico, C=100

Fig. 20: Tempo total em função do número de processadores, para diversos valores de N (pior tempo em 20 execuções).

Fig. 21: Número médio de elementos transmitidos por cada processador no passo 10, para diversos valores de N.

Fig. 22: Speedup em função do número de processadores, para diversos valores de N (pior tempo em 20 execuções).

Alg. 3 PVM: Caso Médio, C=100

Alg. 3 PVM: Caso Especial, C=100

Alg. 3 PVM: Pior Caso, C=100

Alg. 3 PVM: Pior Caso Randômico, C=100

Fig. 23: Tempo total em função do número de elementos (pior tempo em 20 execuções).

Alg. 3 PVM: C=100, P=2, N=1048576

Alg. 3 PVM: C=100, P=16, N=1048576

Alg. 3 PVM: C=100, P=2, N=131072

Alg. 3 PVM: C=100, P=16, N=131072

Fig. 24: Tempos parciais para diversos valores de N e P.

Alg. 1 MPI: Caso Médio, N=8388608

Alg. 1 MPI: Caso Especial, N=8388608

Alg. 1 MPI: Pior Caso, N=8388608

Alg. 1 MPI: Pior Caso Randômico, N=8388608

Fig. 25: Tempo total em função do parâmetro C, para N fixo.

Alg. 1 MPI: Caso Médio, N=8388608

Alg. 1 MPI: Caso Especial, N=8388608

Alg. 1 MPI: Pior Caso, N=8388608

Alg. 1 MPI: Pior Caso Randômico, N=8388608

Fig. 26: Número de elementos restantes (ao final do passo 2) em função do parâmetro C, para N fixo.

Alg. 1 MPI: Caso Médio, N=8388608

Alg. 1 MPI: Caso Especial, N=8388608

Alg. 1 MPI: Pior Caso, N=8388608

Alg. 1 MPI: Pior Caso Randômico, N=8388608

Fig. 27: Número de rodadas de execução em função do parâmetro C, para N fixo.

Alg. 1 MPI: Caso Médio, P=2, N=8388608

Alg. 1 MPI: Caso Especial, P=2, N=8388608

Alg. 1 MPI: Pior Caso, P=2, N=8388608

Alg. 1 MPI: Pior Caso Randômico, P=2, N=8388608

Fig. 28: Tempos parciais em função do parâmetro C para P=2.

Alg. 1 MPI: Caso Médio, P=8, N=8388608

Alg. 1 MPI: Caso Especial, P=8, N=8388608

Alg. 1 MPI: Pior Caso, P=8, N=8388608

Alg. 1 MPI: Pior Caso Randômico, P=8, N=8388608

Fig. 29: Tempos parciais em função do parâmetro C para P=16.

Alg. 1 MPI: P=2, N=8388608

Alg. 1 MPI: P=8, N=8388608

Alg. 1 MPI: P=2, N=131072

Alg. 1 MPI: P=8, N=131072

Fig. 30: Tempo total em função do parâmetro C, para diversos valores de N e P.

Fig. 31: Tempo total em função do número de processadores, para diversos valores de N.

Fig. 32: Tempo total em função do número de processadores, para diversos valores de N.

Fig. 33: Speedup em função do número de processadores, para diversos valores de N.

Fig. 34: Tempo total em função do número de elementos.

Alg. 2 MPI: Caso Médio, C=100, Q=16

Alg. 2 MPI: Caso Especial, C=100, Q=16

Alg. 2 MPI: Pior Caso, C=100, Q=16

Alg. 2 MPI: Pior Caso Randômico, C=100, Q=16

Fig. 35: Tempo total em função do número de processadores, para diversos valores de N.

Fig. 36: Tempo total em função do número de processadores, para diversos valores de N.

Fig. 37: Speedup em função do número de processadores, para diversos valores de N.

Alg. 2 MPI: Caso Médio, C=100, Q=16

Alg. 2 MPI: Caso Especial, C=100, Q=16

Alg. 2 MPI: Pior Caso, C=100, Q=16

Alg. 2 MPI: Pior Caso Randômico, C=100, Q=16

Fig. 38: Tempo total em função do número de elementos.

Fig. 39: Tempo total em função do número de processadores, para diversos valores de Q.

Alg. 2 MPI: Caso Médio, C=100, N=4194304

Alg. 2 MPI: Caso Especial, C=100, N=4194304

Alg. 2 MPI: Pior Caso, C=100, N=4194304

Alg. 2 MPI: Pior Caso Randômico, C=100, N=4194304

Fig. 40: Tempo total em função do número de processadores, para diversos valores de Q.

Alg. 2 MPI: Caso Especial, C=100, N=4194304

Alg. 2 MPI: Caso Médio, C=100, N=4194304

Alg. 2 MPI: Pior Caso Randômico, C=100, N=4194304

Alg. 2 MPI: Pior Caso, C=100, N=4194304

Fig. 41: Speedup em função do número de processadores, para diversos valores de N.

Fig. 42: Tempo total em função do número de problemas.

Alg. 3 MPI: Caso Médio, C=100

Alg. 3 MPI: Caso Especial, C=100

Alg. 3 MPI: Pior Caso, C=100

Alg. 3 MPI: Pior Caso Randômico, C=100

Fig. 43: Tempo total em função do número de processadores, para diversos valores de N.

Fig. 44: Tempo total em função do número de processadores, para diversos valores de N.

Fig. 45: Número médio de elementos transmitidos por cada processador no passo 10, para diversos valores de N.

Alg. 3 MPI: Pior Caso Randômico, C=100

Alg. 3 MPI: Pior Caso, C=100

Fig. 46: Speedup em função do número de processadores, para diversos valores de N.

Alg. 3 MPI: Caso Médio, C=100

Alg. 3 MPI: Caso Especial, C=100

Alg. 3 MPI: Pior Caso, C=100

Alg. 3 MPI: Pior Caso Randômico, C=100

Fig. 47: Tempo total em função do número de elementos.

Fig. 48: Tempos parciais para diversos valores de N e P.