Algoritmos BSP/CGM para Ordenação

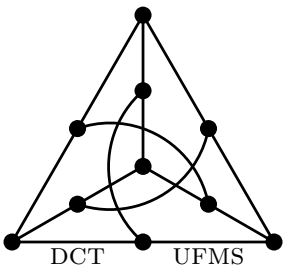
Luciano Gonda

Dissertação de Mestrado

Orientação: Prof. Dr. Henrique Mongelli

Área de Concentração: Ciência da Computação

Dissertação apresentada como requisito para a obtenção do título de mestre em Ciência da Computação.



Departamento de Computação e Estatística
Centro de Ciências Exatas e Tecnologia
Universidade Federal de Mato Grosso do Sul
20 de agosto de 2004

Algoritmos BSP/CGM para Ordenação

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

Campo Grande/MS, 20 de agosto de 2004.

Banca Examinadora:

- Prof. Dr. Henrique Mongelli (orientador) (DCT-UFMS)
- Prof. Dr. Siang Wun Song (IME-USP)
- Prof. Dr. Edson Norberto Cáceres (DCT-UFMS)

aos meus pais Jorge e Zilda

Agradecimentos

Ao meu orientador, professor Dr. Henrique Mongelli, pela dedicação e paciência. Obrigado também pelas orientações e conversas.

Aos meus pais, por todo o incentivo e apoio constante. Obrigado por terem sempre me apoiado em todos os momentos da minha vida. Sem este apoio jamais teria chegado à conclusão deste trabalho. Muito obrigado!

Ao grande amigo Erik, pela constante companhia. Obrigado também pelas sugestões, conversas, ajudas e estímulos durante todo o desenvolvimento deste trabalho.

Aos amigos da UCDB, em especial, Amaury e Danielle, pelas trocas de informações e palavras de incentivo.

Ao professor Edson, pelas sugestões e por ter possibilidado a utilização do Beowulf do IC-Unicamp.

Ao Instituto de Computação da Unicamp (IC-UNICAMP) por ter cedido o Beowulf para realização dos experimentos relacionados a este trabalho.

Aos amigos Jona, Graziela e Luciana pelos momentos de descontração e pelo apoio.

Aos professores e funcionários do DCT pelas oportunidades concedidas. Agradeço por toda ajuda e apoio durante toda minha formação.

A todas as pessoas que colaboraram direta ou indiretamente para o desenvolvimento do trabalho.

Resumo

O problema de ordenação é um assunto bastante estudado em computação. Ordenar uma seqüência $S = (a_1, a_2, \ldots, a_n)$, consiste em obter uma seqüência $S' = (a'_1, a'_2, \ldots, a'_n)$, onde $a'_i \leq a'_j$, se $i < j$. A paralelização de problemas é utilizada para reduzir o tempo de execução dos problemas que necessitam de alto poder de processamento. Neste trabalho descreveremos três algoritmos de ordenação paralelos desenvolvidos no modelo BSP/CGM, no qual cada processador possui memória de tamanho $O(\frac{n}{p})$, e em cada rodada de comunicação são enviados ou recebidos, no máximo, $O(\frac{n}{p})$ dados. Neste modelo queremos sempre minimizar o número de rodadas de comunicação. Os algoritmos que descreveremos são: *Ordenação_Bitonica*, *Ordenação_CD* e *Ordenação_por_Divisão*. O algoritmo *Ordenação_Bitonica* utiliza $O(\log p)$ rodadas de comunicação e tempo de computação local $O(\frac{n \log n}{p})$. Os algoritmos *Ordenação_CD* e *Ordenação_por_Divisão* utilizam $O(1)$ rodadas de comunicação e tempo de computação local $O(\frac{n \log n}{p})$. A implementação do algoritmo *Ordenação_CD* apresenta resultados muito bons em relação ao tempo de execução, mostrando que este algoritmo é eficiente se executado para entradas grandes. Segundo os resultados experimentais, o algoritmo *Ordenação_CD* é o que apresenta os melhores resultados para todas as entradas.

Abstract

Sorting is one of the most studied problems in Computer Science. Sorting a sequence $S = (a_1, a_2, \ldots, a_n)$, consists in obtaining another sequence $S' = (a'_1, a'_2, \ldots, a'_n)$, where $a'_i \leq a'_j$, se $i < j$. To reduce execution time of some problems that require high processing we can use parallel computing. In this work we describe three parallel sorting algorithms designed in BSP/CGM model, in which each processor has $O(\frac{n}{p})$ of local memory, and at most $O(\frac{n}{p})$ data are sent or received in each communication round. In this model, the number of communication rounds has to be as minimum as possible. The algorithms that we will descibe are: *Ordenação_Bitonica*, *Ordenação_CD* and *Ordenação_por_Divisão*. The *Ordenação_Bitonica* algorithm has $O(\log p)$ communication rounds and local time $O(\frac{n \log n}{p})$. *Ordenação_CD* and *Ordenação_por_Divisão* algorithms has $O(1)$ communication rounds local time $O(\frac{n \log n}{p})$. The *Ordenação_CD* presents very good results, showing that this algorithm is efficient when executed with large entries. In agreement with experimental results, the *Ordenação_CD* algorithm presents better results for all entries.

Conteúdo

Capítulo 1

Introdução

A operação de ordenação é uma das operações mais executadas na área de computação, pois a manipulação de dados ordenados, na maioria das aplicações, é mais fácil que a manipulação de dados em uma ordem arbitrária.

O problema de ordenação pode ser definido como a reorganização de uma coleção de elementos em ordem crescente ou decrescente [16]. Knuth [15], define formalmente ordenação da seguinte forma:

Definição 1.1 *Um conjunto de elementos satisfaz uma ordem linear* $<$ *se e somente se:*

1. *quaisquer dois elementos* a *e* b *temos* $a < b$, $a = b$ *ou* $b < a$;
2. *quaisquer três elementos* a, b *e* c, *se* $a < b$ *e* $b < c$, *então* $a < c$.

Definição 1.2 *Dado um conjunto com* n *elementos* $S = \{s_1, s_2, \ldots, s_n\}$, *ordenar* S *em ordem crescente consiste em obter uma permutação* $S' = \{s'_1, s'_2, \ldots, s'_n\}$ *de* S, *tal que se* $i < j$, *então* $s'_i < s'_j$, *para todo* $1 < i, j < n$.

A ordenação é muito importante para outros problemas, especialmente na busca de informações. Em paralelo, a ordenação é utilizada como sub-rotina para resolver diversos problemas em grafos [8, 10, 21], tais como numeração-st, decomposição em orelhas, circuitos de Euler.

Diversos algoritmos foram desenvolvidos para resolver o problema da ordenação. Estes algoritmos, denominados de algoritmos de ordenação, podem ser divididos em duas categorias: algoritmos baseados em comparações e algoritmos não baseados em comparações [16]. Os algoritmos baseados em comparações utilizam operações de comparação e troca, ou seja, fazem várias comparações entre pares de elementos e os trocam de lugar, caso estejam fora de ordem. Os algoritmos não baseados em comparações utilizam algumas propriedades conhecidas dos elementos a serem ordenados.

Com o aumento da utilização de computadores nas mais diversas áreas, aplicações que demandam maiores poderes de processamento também têm aumentado [1]. Associado

a isto, temos a limitação dos componentes físicos utilizados na construção dos computadores, o que limita o poder de processamento de computadores, assim como os custos envolvidos na construção de computadores com melhor desempenho. Dessa forma, para rodarmos aplicações que precisam de alto poder de processamento, podemos utilizar diversos processadores em conjunto, cada um executando simultaneamente uma parte do problema. Conseqüentemente, aplicações que gastariam bastante tempo se executadas em um único processador, podem ter seus tempos de execução reduzidos, se executadas paralelamente.

Para a descrição de algoritmos paralelos, modelos de computação paralela, denominados de realísticos, foram desenvolvidos, tais como BSP, CGM e LogP. Estes modelos são assim denominados porque procuram definir, através de parâmetros, o comportamento de máquinas paralelas reais. O modelo CGM, que foi utilizado no desenvolvimento de nossos algoritmos, foi proposto por Dehne *et al* [9]. Neste modelo, temos p processadores, cada um com memória de tamanho $O(\frac{n}{p})$, onde n é o tamanho da entrada. Além disso, na troca de mensagens entre os processadores podem ser enviados ou recebidos $O(\frac{n}{p})$ dados.

O objetivo do nosso trabalho é estudar e implementar algoritmos BSP/CGM de ordenação a fim de construir uma biblioteca de algoritmos paralelos de ordenação para que possa ser utilizada como parte de outros problemas. Os algoritmos que foram estudados são todos baseados em comparações. É importante observar que, seqüencialmente, o limite inferior para algoritmos de ordenação baseados em comparações é $\Omega(n \log n)$.

1.1 Trabalhos Relacionados

Existem diversas propostas e implementações para o problema de ordenação em paralelo. Chan e Dehne [4] apresentaram um algoritmo de ordenação CGM , que descreveremos mais detalhadamente no Capítulo 4. Em [4], Chan e Dehne executaram testes utilizando 2, 4 e 8 processadores Pentium, com sistema operacional Linux e interligados através de um switch Ethernet. O tamanho das entradas utilizadas varia entre 100000 a 524288 inteiros, onde os tempos de execução obtidos para entradas de 524288 inteiros são, aproximadamente, 1.2s, 2s, e 3.2s utilizando 2, 4 e 8 processadores, respectivamente.

Dusseau *et al.* [11] apresentaram implementações de algoritmos paralelos no modelo LogP. A linguagem utilizada na implementação foi o *Split-C* e a máquina utilizada foi a CM-5.

1.2 Organização do Texto

No Capítulo 2, descrevemos os modelos realísticos de computação paralela BSP, CGM e LogP. Nos Capítulos 3, 4 e 5, descrevemos os algoritmos de ordenação *Ordenação_Bitônica* [1, 3], *Ordenação de Chan & Dehne* [4] e *Ordenação_por_Divisão* [12], respectivamente, bem como os resultados obtidos com a implementação dos mesmos. O ambiente computacional utilizado em nossos experimentos foi um Beowulf de 64 processadores Pentium III

de 500 MHz, cada um com 256 MB de memória RAM. Todos os nós estavam interconectados através de um switch Fast Ethernet. Cada nó executava o sistema operacional Linux RedHat 7.3 com g++ 2.96 e MPI/LAM 6.5.6 [22]. Segundo resultados experimentais, o algoritmo *Ordenação_Bitônica* não apresenta bons resultados em relação ao tempo de execução, porém apresenta ganhos significativos em relação ao algoritmo de ordenação bitônica seqüencial. Neste algoritmo a comunicação é realizada sempre entre pares de processadores, ou seja, não existem troca de mensagens coletivas. Além disso, em cada mensagem são enviados sempre $O(\frac{n}{p})$ dados. Os algoritmos *Ordenação de Chan & Dehne* e *Ordenação_por_Divisão*, apresentam bons resultados para entradas grandes, sendo que um melhor desempenho é apresentado pelo algoritmo *Ordenação de Chan & Dehne*, além de apresentar um bom tempo de execução para entradas menores também.

Por fim, no Capítulo 6, fazemos algumas comparações entre os algoritmos implementados e sugestões de trabalhos futuros.

Capítulo 2

Modelos de Computação Paralela

Ao contrário do que acontece com máquinas seqüenciais, nas quais a principal medida de complexidade é o tempo de execução do algoritmo, no desenvolvimento de algoritmos paralelos devemos levar em consideração diversos outros parâmetros. Dentre esses parâmetros, podemos citar o número de processadores utilizados na solução, bem como a forma como estes foram utilizados, a comunicação e a sincronização entre os processadores.

Dessa forma, diversos modelos para o desenvolvimento de algoritmos paralelos foram desenvolvidos. Dentre os modelos desenvolvidos, o mais utilizado no desenvolvimento de algoritmos paralelos, devido à sua simplicidade, é o modelo PRAM (*Parallel Access Random Machine*). Entretanto, o *speedup* dos resultados obtidos em algoritmos desenvolvidos para esse modelo geralmente não são observados quando implementados em máquinas paralelas reais. Nesse modelo, não é levada em consideração a comunicação entre os processadores, sendo que podem ser usados n^k processadores ($k \in \mathbb{N}$) e n é o tamanho da entrada. Por essa razão, foram desenvolvidos modelos de computação paralela denominados de realísticos. Alguns destes modelos, descritos a seguir, definem alguns parâmetros com a finalidade de mapear as principais características de uma máquina paralela real, ou seja, levando em consideração, dentre outras coisas, o tempo de comunicação entre os processadores.

2.1 Modelo BSP

O modelo BSP (*Bulk Synchronous Parallel*) proposto por Valiant [24] em 1990, foi um dos primeiros modelos de computação paralela a levar em consideração o tempo de comunicação entre os processadores.

Uma máquina BSP consiste em p processadores com memória local e dispositivos de comunicação, interligados através de um roteador, que envia mensagens ponto-a-ponto entre pares de processadores. Além disso, o modelo define algumas facilidades de sincronização entre os processadores, através de alguns parâmetros.

Um algoritmo BSP resume-se em uma seqüência de superpassos, onde cada superpasso

é composto por um passo de computação local e um passo de comunicação entre os processadores, como mostra a Figura 2.1. A soma dos tempos de todos superpassos corresponde ao tempo de execução de um algoritmo BSP.

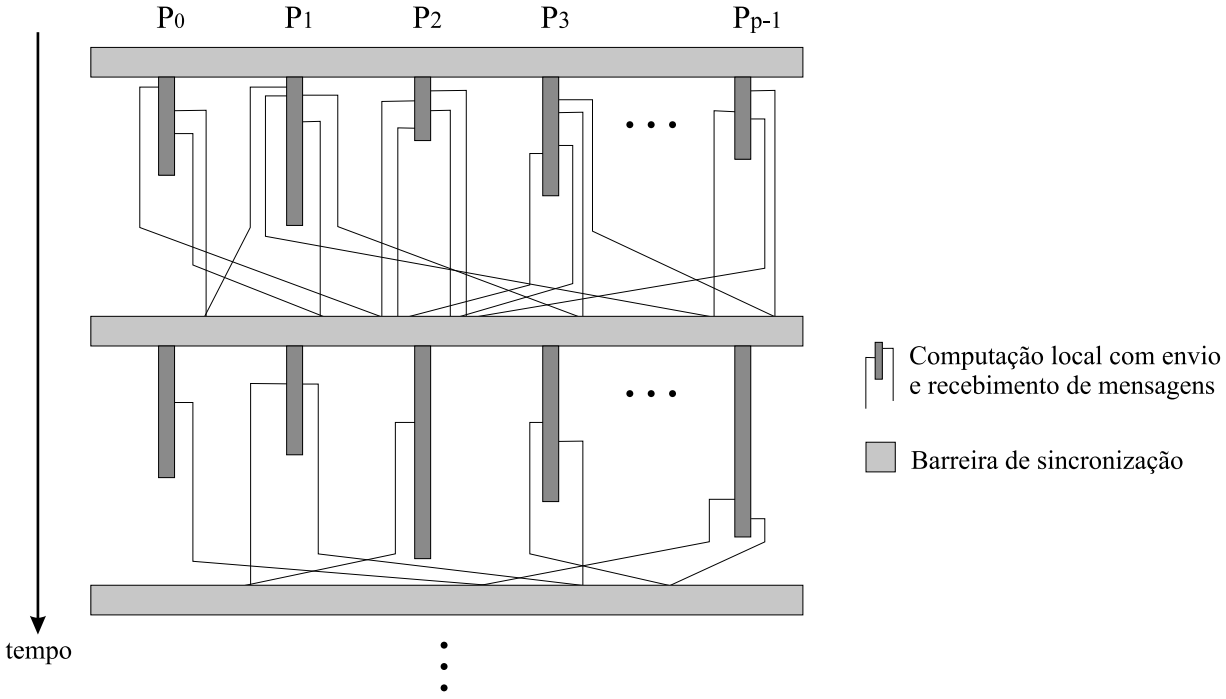


Figura 2.1: Representação da execução de um algoritmo BSP [14].

Os superpassos são separados por uma barreira de sincronização, realizada a cada L unidades de tempo, assegurando que os dados recebidos pelos processadores no i-ésimo superpasso estarão disponíveis no $(i+1)$-ésimo superpasso. Assim, L é definido como a periodicidade ou o tempo máximo de um superpasso.

Outro parâmetro existente no modelo BSP é a taxa de eficiência de computação e comunicação, denominada de g, e correspondente à razão entre a capacidade computacional e a capacidade de comunicação do sistema.

Neste modelo, uma h-relação é definida como o envio ou recebimento de no máximo h mensagens em um superpasso por processador.

2.2 Modelo CGM

O modelo CGM (*Coarsed Grained Multicomputer*) foi proposto por Dehne *et al.* [9] em 1993, e consiste em um conjunto de p processadores, cada um com memória local de tamanho $O(\frac{n}{p})$ e conectados por uma rede de interconexão, onde n é o tamanho do problema e $\frac{n}{p} \geq p$. Este modelo é mostrado na Figura 2.2.

Um algoritmo CGM possui rodadas de computação local alternadas com rodadas de

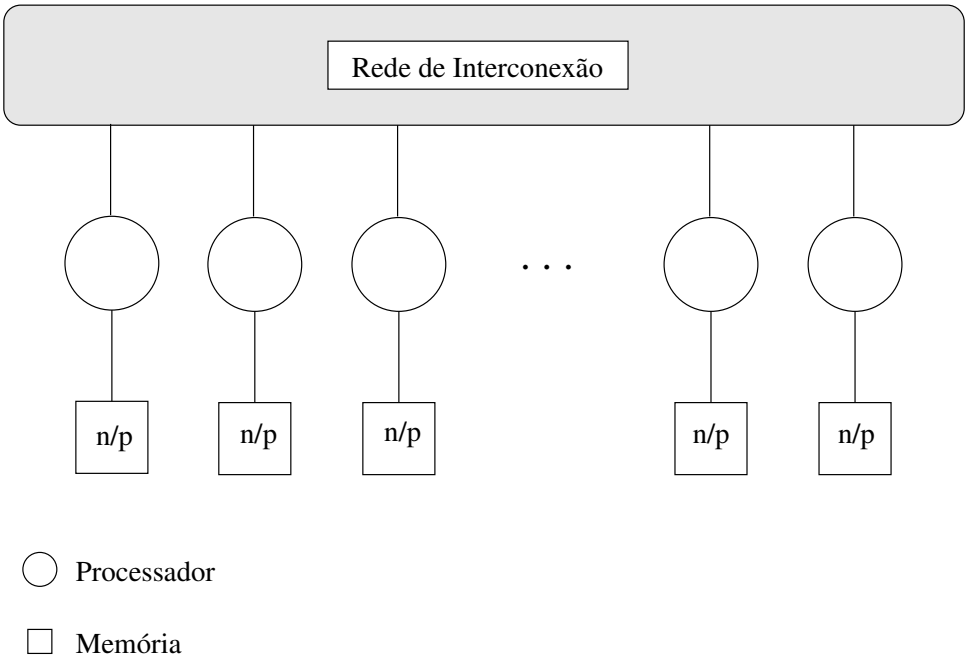


Figura 2.2: Modelo CGM com p processadores e tamanho da entrada igual a n [20].

comunicação entre os processadores, onde cada processador envia e recebe, no máximo, $O(\frac{n}{p})$ dados no máximo. As rodadas de computação local e de comunicação entre os processadores são separadas por barreiras de sincronização, como mostrado na Figura 2.3.

O tempo de execução de um algoritmo CGM é a soma dos tempos gastos tanto com computação local quanto com comunicações entre os processadores. Nas rodadas de computação local, geralmente utilizamos o melhor algoritmo seqüencial para o processamento, além de estarmos interessados em minimizar o número de rodadas de computação.

O modelo CGM, assim como o modelo BSP descrito anteriormente, é um modelo dito realístico, pois leva em consideração o tempo gasto com comunicação entre os processadores para computar o tempo total de execução dos algoritmos.

Além do número de parâmetros, o que diferencia o modelo CGM do modelo BSP, é que no CGM os passos de computação local e de comunicação são separados por barreiras de sincronização, enquanto no BSP, temos passos de computação local junto com passos de comunicação entre processadores, porém os dados comunicados só estão disponíveis para o processamento após a sincronização.

2.3 Modelo LogP

O modelo LogP é um modelo realístico desenvolvido por Culler *et al.* [7] em 1993, com o objetivo de definir, em poucos parâmetros, o funcionamento de máquinas paralelas reais.

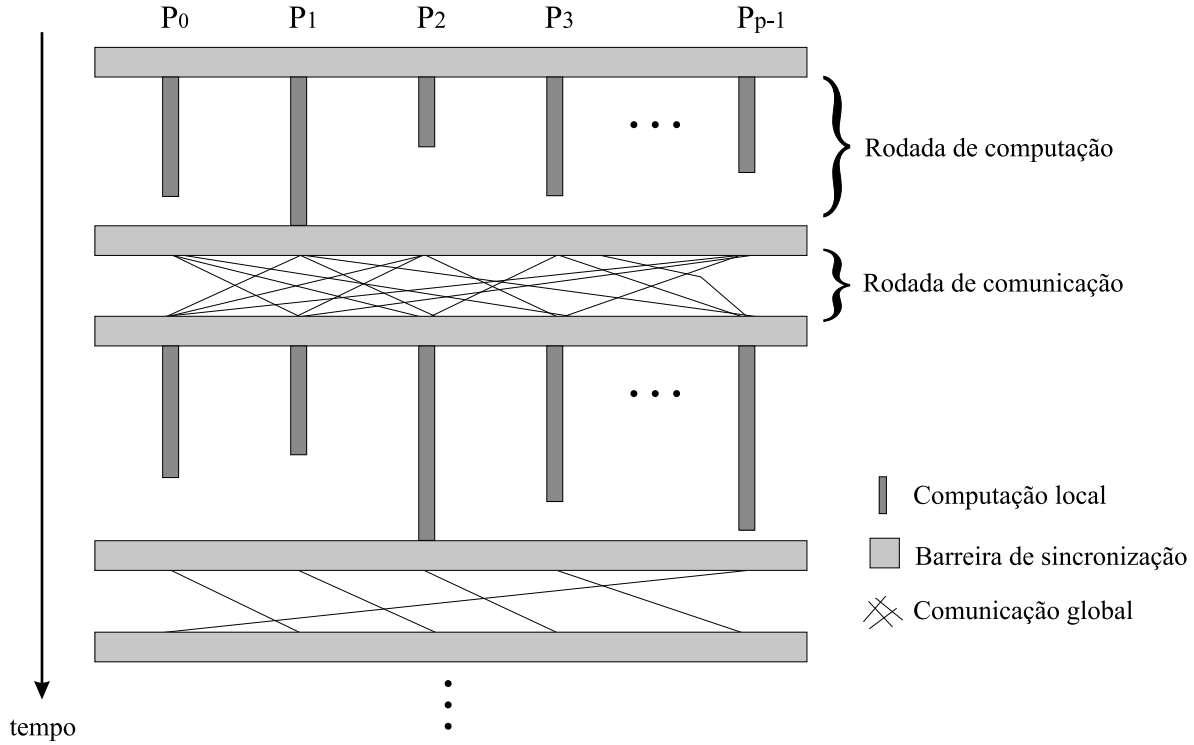


Figura 2.3: Representação da execução de um algoritmo CGM [14].

Neste modelo, assume-se que uma máquina paralela é composta por processadores com memória local, interligados por uma rede de interconexão. Os parâmetros definidos no modelo, não especificam a estrutura da rede de interconexão, mas sim, o desempenho da mesma durante a execução do algoritmo.

Os principais parâmetros existentes no modelo LogP são:

- L: limite superior para a latência,
- o: o *overhead*, definido como a quantidade de tempo em que um processador está envolvido na transmissão ou no recebimento de cada mensagem. Durante este período, o processador não pode executar nenhuma outra operação.
- g: o *gap*, definido como o intervalo mínimo entre o recebimento ou envio de mensagens consecutivas em um processador.
- P: número de processadores utilizados na solução. Assume-se que uma operação local leva uma unidade de tempo para ser executada, denominada de ciclo.

Além desses parâmetros, o modelo LogP impõe restrições sobre a rede de interconexão, assumindo que em cada instante de tempo, um processador pode estar enviando ou recebendo no máximo $\lceil \frac{L}{g} \rceil$ mensagens. Caso uma operação exceda esse limite, o processador responsável pela mesma deve esperar até que a transmissão possa ocorrer sem que essa restrição seja violada.

Este modelo é assíncrono, pois os processadores trabalham de forma assíncrona e o tempo gasto para o envio de uma mensagem não pode ser previsto, apesar de ser limitado por L. Neste modelo, assume-se que as mensagens sejam de tamanho pequeno e que os parâmetros L, o e g são medidos como múltiplos do ciclo do processador.

Em um algoritmo LogP, as principais métricas são o tempo máximo e o espaço utilizado por cada processador.

2.4 Notas

Neste capítulo apresentamos três modelos realísticos de computação paralela. Estes modelos, além de levarem em consideração o tempo de troca de mensagens entre os processadores, definem parâmetros para diminuir o gargalo gerado pela comunicação entre os processadores.

Assim, estes modelos buscam representar o comportamento de máquinas paralelas reais, através de parâmetros que possam expressar o comportamento da rede de interconexão.

Capítulo 3

Ordenação Bitônica

Neste capítulo descreveremos um algoritmo paralelo de ordenação que utiliza a idéia de unir pares de subseqüências, alternadamente com ordenações locais. O algoritmo é baseado na idéia do algoritmo apresentado em Cáceres *et al.* [3]. Na Seção 3.1, definimos o conceito de ordenação bitônica e descrevemos um algoritmo seqüencial de ordenação bitônica. Na Seção 3.2, descreveremos um algoritmo paralelo para ordenação bitônica que tem como entrada uma seqüência de n números e utiliza p processadores, onde $\frac{n}{p} \geq p$. Na Seção 3.3, mostramos os resultados obtidos com a implementação. Por fim, apresentamos na Seção 3.4 alguns comentários sobre os resultados obtidos.

3.1 Algoritmo Seqüencial

Definição 3.1 *Uma seqüência de números (a_1, a_2, ..., a_n) é dita bitônica, se existe um inteiro $1 \leq j \leq n$, tal que $a_1 \leq a_2 \leq \ldots \leq a_j \geq a_{j+1} \geq \ldots \geq a_n$.*

Uma seqüência de números também é denominada bitônica se pudermos deslocá-la ciclicamente, tal que a seqüência resultante S seja bitônica, ou seja, existe um número inteiro $1 \leq k \leq n$, tal que $S_1 \leq S_2 \leq \ldots \leq S_k \geq S_{k+1} \geq \ldots \geq S_n$, onde S_i é um elemento da seqüência após o deslocamento.

Exemplo 3.1 *A seqüência (1, 3, 5, 6, 7, 4, 2) é bitônica, com $k = 5$. Analogamente, a seqüência (9, 6, 3, 2, 5, 7, 10) também é bitônica, pois podemos deslocá-la ciclicamente, obtendo a seqüência bitônica (2, 5, 7, 10, 9, 6, 3), com $k = 4$.*

Teorema 3.1 *Seja $S = (a_1, a_2, \ldots, a_{2n})$ uma seqüência bitônica. Podemos obter duas seqüências bitônicas aplicando a operação de **divisão bitônica** da seguinte forma [1]:*

$S_{min} = (min\{a_1, a_{n+1}\}, \ldots, min\{a_n, a_{2n}\})$

e

$S_{max} = (max\{a_1, a_{n+1}\}, \ldots, max\{a_n, a_{2n}\})$.

Além disso, temos que $max(S_{min}) \leq min(S_{max})$, ou seja, todos os elementos de S_{min} são menores ou iguais aos elementos de S_{max}.

Prova. Sejam $d_i = \min(a_i, a_{n+i})$ e $e_i = \max(a_i, a_{n+i})$, onde $1 \leq i \leq n$, então temos que $S_{min} = d_1, d_2, \ldots, d_n$ e $S_{max} = e_1, e_2, \ldots, e_n$. Temos que provar as seguintes propriedades:

1. S_{min} e S_{max} são bitônicas.
2. $\max(S_{min}) \leq \min(S_{max})$.

Como vimos anteriormente, se deslocarmos uma seqüência bitônica ciclicamente, continuamos tendo uma seqüência bitônica. Entretanto, esse deslocamento, apesar de alterar os elementos de S_{min} e S_{max}, não altera as propriedades 1 e 2, apresentadas anteriormente. Dessa forma, precisamos provar o teorema onde

$$a_1 \leq a_2 \leq \ldots \leq a_j \geq a_{j+1} \geq \ldots \geq a_{2n}$$

seja verdadeira para $1 \leq j \leq 2n$. Mais ainda, como a seqüência inversa $a_{2n}, a_{2n-1}, \ldots, a_1$ também é bitônica e as propriedades 1 e 2 continuam valendo, podemos assumir, sem perda de generalidade, que $n \leq j \leq 2n$ e provarmos o teorema para esse intervalo.

Caso 1: $a_n \leq a_{2n}$, então $a_i \leq a_{n+i}$. Conseqüentemente, $d_i = a_i$ e $e_i = a_{n+i}$, $1 \leq i \leq n$. Neste caso, ambas as propriedades são satisfeitas.

Caso 2: $a_n > a_{2n}$, então como $a_{j-n} < a_j$, podemos encontrar um índice k, $j \leq k \leq 2n$, tal que $a_{k-n} \leq a_k$ e $a_{k-n+1} > a_k$.

Segue que

$$d_i = a_i \text{ e } e_i = a_{n+i}, 1 \leq i \leq k - n$$

e

$$d_i = a_{n+i} \text{ e } e_i = a_i, k - n < i \leq n.$$

Portanto, temos que

$$d_i \leq d_{i+1}, 1 \leq i \leq k - n$$

e

$$d_i \geq d_{i-i}, k - n < i \leq n,$$

ou seja, $S_{min} = d_1, d_2, \ldots, d_n$ é bitônica. Analogamente, temos que

$$e_i \leq e_{i+1}, k-n \leq i \leq n$$

e

$$e_i \geq e_{i+1}, 1 \leq i \leq k-n,$$

o que significa que $S_{max} = e_1, e_2, \ldots, e_n$ é bitônica. Portanto, provamos a propriedade 1. Para provar 2, observe que

$$\max(S_{min}) = \max(d_{k-n}, d_{k-n+1}) = \max(a_k, a_{k-n+1})$$

e

$$\min(S_{max}) = \min(e_{k-n}, e_{k-n+1}) = \min(a_k, a_{k-n+1}).$$

Como $a_k \geq a_{k+1}$, $a_k \geq a_{k-n}$, $a_{k-n+1} \geq a_k$ e $a_{k-n+1} \geq a_{k+1}$, temos que $\max(a_{k-n}, a_{k+1})$ $\leq \min(a_k, a_{k-n+1})$. Logo,

$$\max(S_{min}) \leq \min(S_{max})$$

□

Exemplo 3.2 *Considere a seqüência bitônica S = (2, 3, 6, 7, 8, 5, 4, 1). Logo, podemos obter as seguintes seqüências bitônicas: S_{min} = (2, 3, 4, 1) e S_{max} = (8, 5, 6, 7).*

Como todos os elementos de S_{min} são menores ou iguais aos elementos de S_{max} e ambas as seqüências são bitônicas, podemos dividir recursivamente S_{min} e S_{max} até que sejam obtidas seqüências de tamanho 1. Quando isto ocorrer, toda a seqüência estará ordenada.

Exemplo 3.3 *Dada a seqüência bitônica S = (1, 4, 5, 6, 9, 8, 7, 2). Aplicando a divisão bitônica sucessivamente até que as seqüências obtidas tenham tamanho 1, obtemos a seqüência S′ = (1, 2, 4, 5, 6, 7, 8, 9). A Figura 3.1 mostra os passos desse procedimento.*

Assim, dada uma seqüência bitônica, obtemos uma seqüência ordenada, gastando tempo $O(n \log n)$, através da aplicação de operações de divisões bitônicas sucessivas. Entretanto, nem toda seqüência numérica S é bitônica. Neste caso, devemos transformar S em uma seqüência bitônica S'. Em seguida, podemos ordenar os elementos de S' através de divisões bitônicas sucessivas e, como os elementos de S e S' são os mesmos, temos a seqüência S ordenada.

Inicialmente, é importante observarmos que qualquer seqüência contendo dois elementos é bitônica. Mais ainda, uma seqüência com dois elementos já está ordenada de forma

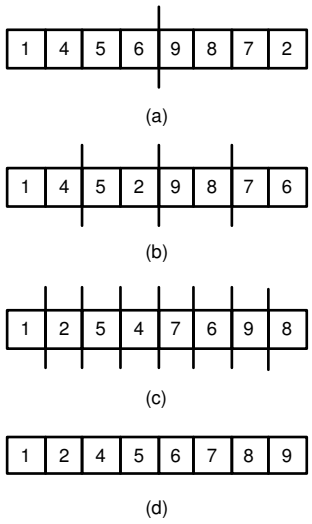


Figura 3.1: Aplicações suscessivas da operação de divisão bitônica.

crescente ou decrescente. Dadas duas seqüências $S_1 = (a_0, a_1)$ e $S_2 = (a_2, a_3)$, transformamos S_1 em crescente e S_2 em decrescente, comparando os elementos de cada seqüência e posicionando-os de forma apropriada. Dessa forma, obtemos uma seqüência bitônica $S = S_1 \cup S_2$ com quatro elementos.

Para um seqüência com n elementos, podemos dividir a seqüência em subseqüências bitônicas de $\frac{n}{2}$ elementos. Para ilustrar, consideremos uma seqüência $S = (a_0, a_1, \ldots, a_7)$, com oito elementos. Podemos dividir S em duas subseqüências $S_1 = (a_0, a_1, a_2, a_3)$ e $S_2 = (a_4, a_5, a_6, a_7)$. Como mostrado anteriormente, S_1 e S_2 podem ser transformadas em seqüências bitônicas. Para que S seja bitônica, devemos ordenar uma das seqüências de forma crescente e outra de forma decrescente. A subseqüência S_1 pode ser ordenada de forma crescente através de divisões bitônicas sucessivas, como descrito anteriormente. Para ordenar S_2 de forma decrescente, podemos aplicar, recursivamente, divisões bitônicas modificadas, colocando os máximos na primeira metade e os mínimos na segunda metade da seqüência. Após estas operações, temos que $S' = S_1 \cup S_2$ é uma seqüência bitônica, onde os elementos de a'_0 a a'_3 estão ordenados em ordem crescente e os elementos de a'_4 a a'_7 estão ordenados em ordem decrescente. Assim, como visto anteriormente, podemos ordenar esta seqüência, gastando mais $O(n \log n)$.

3.2 Algoritmo Paralelo

No algoritmo paralelo (*Ordenação_Bitônica*), assumimos que o número de elementos n da seqüência a ser ordenada seja potência de dois. Caso n não seja potência de dois, podemos completar com valores inválidos, que serão descartados ao final da execução do algoritmo. A cada rodada do algoritmo, cada processador armazena $O(\frac{n}{p})$ elementos da seqüência global.

A idéia do algoritmo que descreveremos é baseada na operação de divisões bitônicas sucessivas e ordenações locais, até que toda a seqüência esteja ordenada. É importante observar que, em paralelo, a operação de divisão bitônica é sempre executada entre pares de processadores, de forma que, após cada operação, os elementos da seqüência bitônica S_{min} ficarão armazenadas em um dos processadores (o de menor índice entre os dois), enquanto os elementos da seqüência bitônica S_{max} ficarão armazenados em outro processador (o de maior índice entre os dois).

Algoritmo: Ordenação_Bitônica
Entrada: uma seqüência de n elementos, distribuída entre os p (com rótulos gloabais de 0 a $p-1$) processadores, com $\frac{n}{p}$ dados em cada processador. Durante o algoritmo os processadores são re-rotulados localmente da forma $P_{g,i}$, onde g é o rótulo do grupo a que pertencem e i é o rótulo do processador dentro do grupo.
Saída: A seqüência ordenada de elementos distribuída entre os p processadores.

1. Cada processador ordena seus dados localmente, utilizando o quicksort. Os processadores de identificador par ordenam em ordem crescente e os processadores de identificador ímpar ordenam os dados em ordem decrescente.
2. **para** i **de** 1 **ate** $\log p - 1$ **faça**
3. $k = 2^{i-1}$
4. Agrupe os processadores em $g = \frac{p}{2k}$ grupos, contendo $t = 2k$ processadores adjacentes cada. Os grupos são rotulados de 0 a $g-1$ e os processadores são identificados dentro do seu grupo através de índices de 0 a $t-1$.
5. Execute em paralelo, em cada um dos g grupos, a operação de divisão bitônica entre $P_{g,j}$ e $P_{g,j+k}$, onde $0 \leq j < k$. Nos grupos de rótulo par, S_{min} ficará armazenada no processador de menor índice global e S_{max} ficará armazenada no processador de maior índice global. Nos grupos de rótulo ímpar, S_{min} ficará armazenada no processador de maior índice global e S_{max} ficará armazenada no processador de menor índice global.
6. Cada processador ordena seus dados localmente, onde os processadores pertencentes aos grupos de rótulo par, ordenam os dados em ordem crescente e os processadores que pertencem aos grupos de rótulo ímpar, ordenam os dados em ordem decrescente.
7. **para** i **de** 1 **até** $\log p$ **faça**
8. $k = \frac{p}{2^i}$
9. Agrupe os processadores em $g = \frac{p}{2k}$ grupos, contendo $t = 2k$ processadores adjacentes cada. Os grupos são rotulados de 0 a $g-1$ e os processadores são identificados dentro do seu grupo através de índices de 0 a $t-1$.
10. Execute em paralelo, em cada um dos g grupos, a operação de divisão bitônica entre $P_{g,j}$ e $P_{g,j+k}$, onde $0 \leq j < k$. Nos grupos de rótulo par, S_{min} ficará armazenada no processador de menor índice global e S_{max} ficará armazenada no processador de maior índice global. Nos grupos de rótulo ímpar, S_{min} ficará armazenada no processador de maior índice global e S_{max} ficará armazenada no processador de menor índice global.
11. Cada processador ordena localmente seus elementos em ordem crescente.

fim algoritmo

Teorema 3.2 *O algoritmo* Ordenação_Bitônica *ordena corretamente n inteiros utilizando p processadores, com* $\frac{n}{p}$ *inteiros por processador.*

Prova. Para ordenar n inteiros utilizando o algoritmo *Ordenação_Bitônica* são necessárias duas etapas: inicialmente, é necessário transformar a seqüência de entrada S em uma seqüência bitônica S'. Em seguida, podemos ordenar S' utilizando operações de divisões bitônicas. Como os elementos de S e S' são os mesmos, ao final destas duas etapas, temos a seqüência S ordenada.

Vamos mostrar que ao término do laço da linha 2, teremos uma seqüência bitônica distribuída entre os processadores envolvendo todos os dados. Para isto, consideremos o seguinte invariante: no fim de cada iteração do laço da linha 2, temos seqüências bitônicas de tamanho $4\frac{n}{p}k$.

Antes da primeira iteração cada processador está ordenado crescente ou decrescente, de acordo com a paridade do seu identificador. Os processadores, cujo identificador é par, estão ordenados em ordem crescente e os processadores com identificador ímpar estão ordenados em ordem decrescente.

Cada par de processadores representa uma seqüência bitônica de tamanho $2\frac{n}{p}$. Se considerarmos $i = 0$ e $k = \frac{1}{2}$, o invariante é válido antes do início do laço.

Na primeira iteração temos $k = 1$. No Passo 4, os processadores são agrupados e re-rotulados com rótulos $P_{g,j}$, onde $0 \leq g \leq \frac{p}{2k} - 1$ e $0 \leq j \leq 2k - 1$. A operação de divisão bitônica do Passo 5 é realizada entre pares de processadores vizinhos, de modo que nos processadores de índices globais pares teremos os valores menores e nos processadores de índices globais ímpares os maiores, alternadamente.

No Passo 6, os dados locais são ordenados crescente ou decrescente, conforme os rótulos dos seus grupos. Assim, ao final da primeira iteração teremos $4\frac{n}{p}$ elementos formando seqüências bitônicas em 4 processadores adjacentes. Dessa forma, o invariante vale para $i = 1$.

Vamos supor que o invariante seja válido no início da r-ésima iteração. Mostraremos que ele permanece válido no final desta iteração. Na iteração anterior, $k = 2^{r-2}$ e existiam seqüências bitônicas de tamanho $4\frac{n}{p}2^{r-2} = \frac{n}{p}2^r$, entre $4*2^{r-2} = 2^r$ processadores disjuntos.

Na r-ésima iteração, k é atualizado para $k = 2^{r-1}$. No Passo 4, são construídos $g = \frac{p}{2^r}$ grupos. Os processadores recebem novos rótulos $P_{g,j}$, onde $0 \leq g \leq \frac{p}{2^r} - 1$ e $0 \leq j \leq 2^r - 1$.

Como supusemos que o invariante vale no inicio da iteração, os dados dos processadores $P_{g,j}$ com $0 \leq j < k$ estão em ordem crescente e os dados dos processadores $P_{g,j}$, $k \leq j < 2k$ estão em ordem decrescente, e formam uma seqüência bitônica de tamanho $2\frac{n}{p}k$, com $k = 2^{r-1}$.

No Passo 5, é realizada, em cada grupo, uma divisão bitônica entre os processadores de rótulo $P_{g,j}$ e $P_{g,j+k}$. Se g é par, na divisão bitônica S_{min} estará nos processadores $P_{g,j}$ com $0 \leq j < k$, e S_{max} nos processadores $P_{g,j}$ com $k \leq j < 2k$. Caso contrário, S_{max} estará nos processadores $P_{g,j}$ com $0 \leq j < k$ e S_{min} nos processadores $P_{g,j}$ com $k \leq j < 2k$. Assim, ao final da operação de divisão bitônica, teremos $x \leq y$ se $x \in P_{g,j}$ e $y \in P_{g,j+1}$,

$0 \leq j < 2k - 1$.

No Passo 6, é realizada uma ordenação dentro de cada processador conforme o rótulo do grupo.

Dessa forma, ao final da r-ésima iteração teremos uma seqüência bitônica de tamanho $4\frac{n}{p}k$, com $k = 2^{r-1}$, e o invariante é válido.

Portanto, após a execução da última iteração do laço da linha 2, teremos uma seqüência bitônica de tamanho $4\frac{n}{p}2^{\log p-2} = n$, pois $k = 2^{\log p-2}$. Ou seja, transformamos a seqüência inicial em uma seqüência bitônica formada pelos n elementos distribuída entre os p processadores.

Mostraremos agora, que ao término da linha 11, teremos uma seqüência ordenada distribuída entre os p processadores, envolvendo todos os elementos.

Para isto, consideremos inicialmente o seguinte invariante: após a cada iteração do laço da linha 7, temos seqüências bitônicas de tamanho $k\frac{n}{p}$.

Antes da primeira iteração, se considerarmos $i = 0$ e $k = p$, teremos uma seqüência bitônica de tamanho n, como mostrado anteriormente, e o invariante é válido.

Na primeira iteração do laço da linha 7 do algoritmo *Ordenação_Bitônica*, temos $k = \frac{p}{2}$. No Passo 9, os processadores são agrupados e re-rotulados com rótulos $P_{g,j}$, onde $0 \leq g \leq \frac{p}{2k} - 1$ e $0 \leq j \leq 2k - 1$. A operação de divisão bitônica do Passo 10 é executada entre pares de processadores P_j e P_{j+k}, de modo que os processadores de índices menores armazenam os menores elementos e os processadores de índices globais maiores armazenam os valores maiores. Assim, ao final da primeira iteração teremos S_{min} armazenada em $\frac{p}{2}$ processadores adjacentes P_j, $0 \leq j \leq k - 1$ e S_{max} armazenada nos $\frac{p}{2}$ processadores adjacentes P_j, $k \leq j \leq 2k - 1$, ou seja, temos seqüências bitônicas de tamanho $\frac{n}{2}$. Dessa forma, o invariante vale para $i = 1$.

Supusemos que o invariante seja válido no início da r-ésima iteração, vamos mostrar que ele permanece válido ao final desta iteração. Na iteração anterior, $k = \frac{p}{2^{r-1}}$ e existiam seqüências bitônicas de tamanho $\frac{p}{2^{r-1}}$.

Na r-ésima iteração, $k = \frac{p}{2^r}$. No Passo 9, são construídos $g = \frac{p}{2}$ grupos e os processadores recebem novos rótulos $P_{g,j}$, $0 \leq g \leq \frac{p}{2^r} - 1$ e $0 \leq j \leq 2k - 1$.

Como supomos que o invariante vale no início da iteração, temos seqüências bitônicas de tamanho $\frac{n}{2^{r-1}}$. No Passo 10, é realizada, em cada grupo, uma divisão bitônica entre os processadores de rótulo $P_{g,j}$ e $P_{g,j+k}$. Como cada uma das seqüências bitônicas de tamanho $\frac{n}{2^{r-1}}$ foi dividida em duas seqüências bitônicas, temos agora seqüências bitônicas de tamanho $\frac{n}{2^r}$.

Dessa forma, ao final da r-ésima iteração, teremos uma seqüência bitônica de tamanho $\frac{n}{2^r} = k\frac{n}{p}$, e o invariante é válido.

Portanto, após a execução da última iteração do laço da linha 7, temos uma seqüência bitônica de tamanho $\frac{p}{2^{\log p}}\frac{n}{p} = \frac{n}{p}$. Ou seja, cada processador armazena uma seqüência bitônica de tamanho $\frac{n}{p}$.

Mais ainda, como em todas as iterações os elementos da seqüência S_{min} estarão armazenados nos processadores $P_{g,j}$ com $0 \leq j \leq k$, enquanto os elementos de S_{max} estarão armazenados nos processadores $P_{g,j}$ com $k \leq j \leq 2k-1$, teremos $x \leq y$ se $x \in P_j$ e $y \in P_{j+1}$, $0 \leq j \leq p-1$.

No Passo 11, é realizada uma ordenação local em cada um dos p processadores, garantindo que tenhamos uma seqüência ordenada com n elementos, distribuída pelos p processadores.

□

Teorema 3.3 *O algoritmo* Ordenação_Bitônica *utiliza* $O(\log p)$ *rodadas de comunicação (*$\log p$ $\frac{n}{p}$*-relações) e tempo de computação local* $O(\frac{n \log n}{p})$ *para ordenar* n *números distribuídos entre* p *processadores.*

Prova. Para provarmos o Teorema 3.3, temos de mostrar o tempo gasto com computação local e o tempo gasto com comunicação entre os processadores.

Mostraremos inicialmente o tempo gasto com comunicação entre os processadores. No laço da linha 2, em cada iteração é executada uma operação de divisão bitônica em paralelo. Como são necessárias $\log p - 1$ iterações deste laço e em cada operação de divisão bitônica é necessária uma rodada de comunicação, temos $O(\log p)$ rodadas de comunicação no laço da linha 2.

Analogamente, no laço da linha 7, novamente é executada uma comunicação em cada iteração. Como existem, $\log p$ iterações, temos $O(\log p)$ rodadas de comunicação para executar o laço da linha 7.

Portanto, o algoritmo *Ordenação_Bitônica* precisa de $O(\log p)$ rodadas de comunicação para ordenar n elementos distribuídos em p processadores.

Mostraremos agora, o tempo gasto com computação local. Nas ordenações dos Passo 1, 6, 11, cada processador gasta tempo $O(\frac{n \log n}{p})$. Logo, o tempo de computação local gasto pelo algoritmo *Ordenação_Bitônica* é $O(\frac{n \log n}{p})$.

Portanto, o algoritmo *Ordenação_Bitônica* gasta $O(\frac{n \log n}{p})$ de tempo de computação local e $O(\log p)$ rodadas de comunicação para ordenar n elementos distribuídos entre p processadores, $\frac{n}{p}$ elementos em cada.

□

É importante observar, que apesar de utilizar a operação de divisão bitônica, a entrada deste algoritmo não precisa ser necessariamente uma seqüência bitônica, pois o mesmo transforma a seqüência de entrada em uma seqüência bitônica, utilizando $\log p - 1$ passos, onde em cada passo i são executadas i rodadas de comunicação.

Para ilustrar o funcionamento do algoritmo *Ordenação_Bitônica*, considere a seqüência de elementos (7, 3, 9, 14, 16, 8, 1, 10, 12, 4, 5, 13, 15, 2, 6, 11), distribuída por 4 processadores. A Figura 3.2 mostra a execução do algoritmo para ordenar esta seqüência de inteiros.

No exemplo da Figura 3.2, são necessários 2 passos para ordenação dos elementos. O primeiro passo transforma a seqüência de entrada em uma seqüência bitônica, utilizando uma rodada de comunicação. No segundo passo, são utilizadas duas rodadas de comunicação para ordenar a seqüência obtida no Passo 1.

Inicialmente, os elementos são distribuídos pelos processadores (Figura 3.2(a)). Após a distribuição, os elementos são ordenados localmente: os processadores de identificador par ordenam os dados em ordem crescente e os processadores de identificador ímpar ordenam os dados em ordem decrescente. Assim, obtemos duas seqüências bitônicas, uma formada pelos elementos dos processadores P_0 e P_1 e outra formada pelos elementos dos processadores P_2 e P_3, como mostra a Figura 3.2*(b)*.

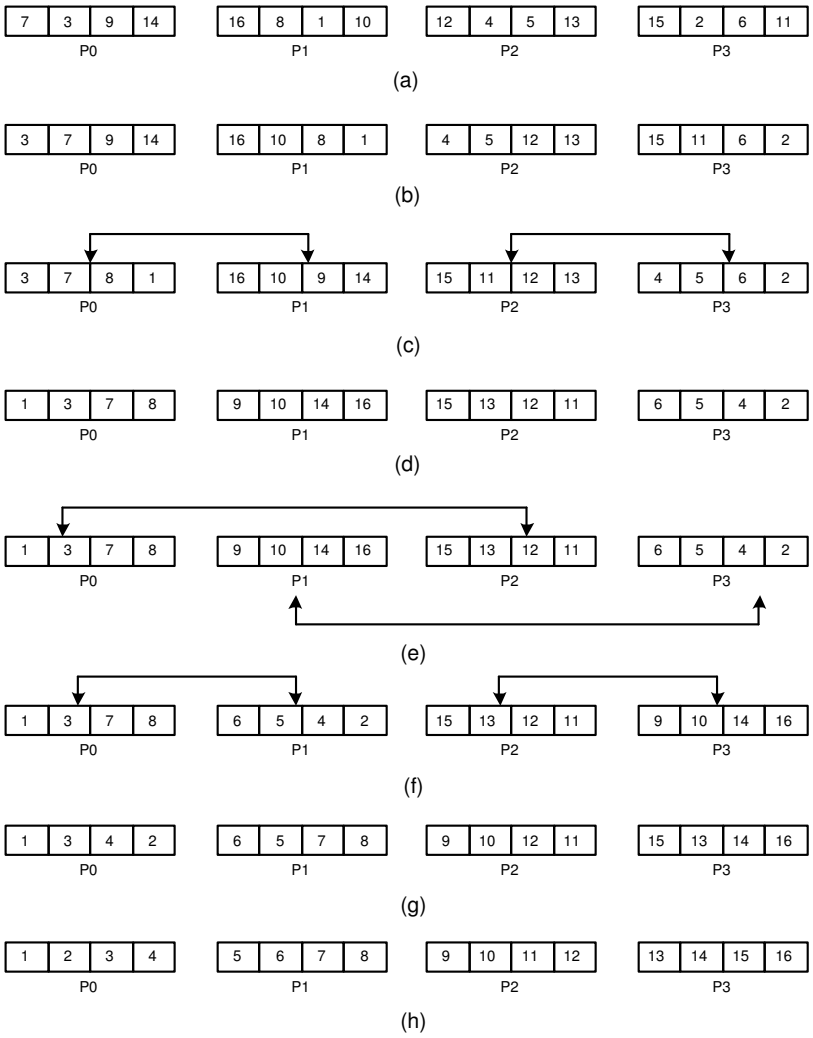


Figura 3.2: Exemplo dos passos do algoritmo *Ordenação_Bitônica*, com $p = 4$ e $n = 16$.

Em seguida, é executada uma rodada de comunicação, na qual duas operações de divisão bitônica são executadas em paralelo, uma delas entre os processadores P_0 e P_1 (P_{0+1}) e outra entre os processadores P_2 e P_3 (P_{2+1}), pois $k = 1$. Na Figura 3.2c, é mostrado o resultado após as divisões bitônicas. Na operação de divisão bitônica entre

os processadores P_0 e P_1, os elementos de S_{min} são armazenados em P_0 e os de S_{max} em P_1. Na divisão bitônica entre P_2 e P_3, os maiores elementos ficam em P_2 e os menores elementos ficam em P_3. Isto é realizado, para que tenhamos novamente uma seqüência bitônica, porém, envolvendo os elementos dos quatro processadores. Finalizada a rodada de comunicação, cada processador executa uma ordenação local. A configuração após esse passo é mostrada na Figura 3.2*(d)*, onde temos uma seqüência bitônica formada pelos elementos dos processadores P_0, P_1, P_2 e P_3.

Neste instante, a seqüência de entrada foi transformada em uma seqüência bitônica e podemos aplicar novamente operações de divisão bitônica. Como $k = 2$, serão executadas operações de divisão bitônica entre P_0 e P_2, e entre P_1 e P_3 (Figura 3.2*(e)*). Após estas operações, os elementos ficam distribuídos como mostra a Figura 3.2*(f)* e k passa a ser igual a 1. A seguir, são executadas operações de divisão bitônica entre os processadores P_0 e P_1 e entre P_2 e P_3, como mostra a Figura 3.2*(f)*, obtendo-se os dados mostrados na Figura 3.2*(g)*.

Por fim, cada processador executa uma ordenação local. Após a ordenação, temos todos os elementos ordenados, distribuídos pelos processadores, como ilustrado na Figura 3.2*(h)*.

3.3 Implementação e Resultados

O algoritmo *Ordenação_Bitônica* apresentado na Seção 3.2 foi implementado na linguagem C/C++ em conjunto com a biblioteca de troca de mensagens MPI.

Os tempos obtidos para ordenação foram medidos em segundos, não incluindo o tempo de distribuição dos dados. A entrada do programa é um arquivo texto, contendo um número inteiro que indica a quantidade de elementos e os elementos a serem ordenados. Os arquivos de entrada foram gerados aleatoriamente e para cada entrada foram utilizados 1, 2, 4, 8, 16 e 32 processadores.

Para melhor visualização dos resultados obtidos dividimos os gráficos contendo os tempos de execução e *speedup* em duas partes. Foram realizados 30 experimentos e o maior tempo obtido foi utilizado para construir os gráficos das Figuras 3.3 a 3.6. OS tempos médios, menor tempo, maior tempo e tempos médios de computação local são apresentados no Apêndice A. Além disso, são mostrados alguns gráficos com tempo de computação local e comunicação no Apêndice D.

Na Figura 3.3 mostramos o tempo de execução do algoritmo *Ordenação_Bitônica* para $32768 \leq n \leq 524288$, onde n é uma potência de 2.

Podemos observar que o algoritmo não apresenta um bom desempenho para poucos processadores, mas o algoritmo apresenta um ganho significativo em relação ao tempo de execução do algoritmo seqüencial. Além disso, com o aumento do número de processadores, o tempo de execução diminui. Podemos verificar experimentalmente que isto ocorre porque quando aumentamos o número de processadores, o tempo gasto com comunicação também aumenta, fazendo com que a comunicação se torne o gargalo do algoritmo,

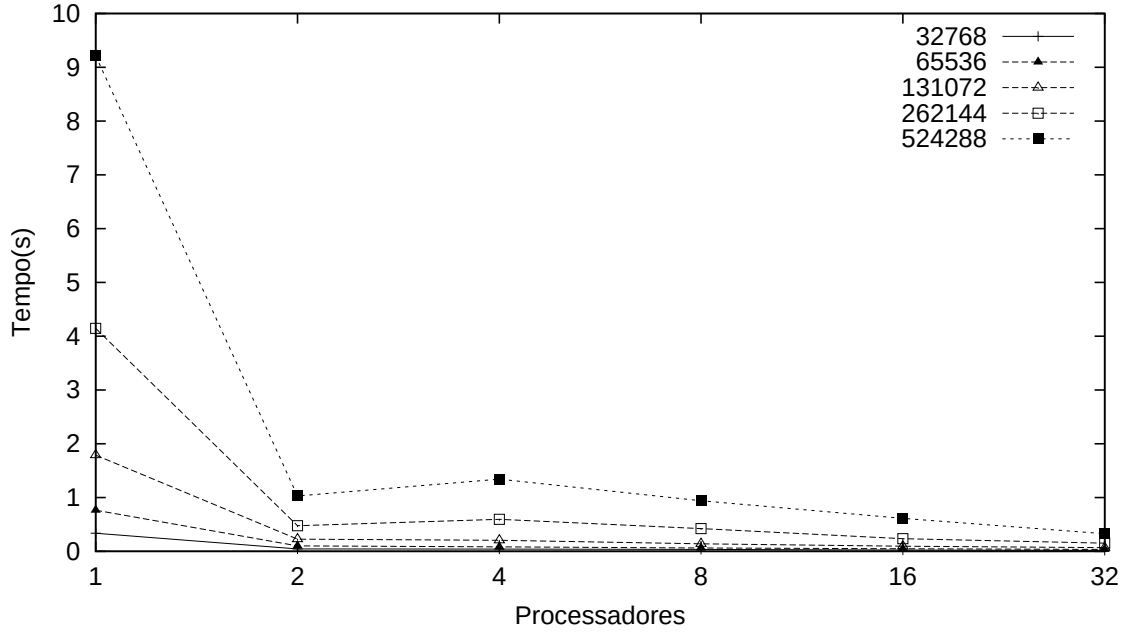


Figura 3.3: Tempo de execução do algoritmo *Ordenação_Bitônica,* $32768 \leq n \leq 524288$

já que a quantidade de rodadas de comunicação é função do número de processadores, como mostrado no Apêndice D. Com poucos processadores, apesar do tempo gasto com comunicação ser relativamente pequeno, o gargalo passa a se tornar o tempo gasto com computação local.

Na Figura 3.4 apresentamos os tempos de execução do algoritmo *Ordenação_Bitônica* para $1048576 \leq n \leq 8388608$, onde n é uma potência de 2. Observamos que mesmo aumentando o tamanho da entrada, continuamos tendo um desempenho ruim com poucos processadores e, à medida que aumentamos o número de processadores, o desempenho do algoritmo melhora. Além disso, o algoritmo paralelo apresenta um ganho mais significativo em relação ao algoritmo seqüencial para entradas maiores.

Nas Figuras 3.5 e 3.6 podemos observar o *speedup* do algoritmo *Ordenação_Bitônica*.

Notamos nestes gráficos que o *speedup* do algoritmo tende a aumentar a medida em que aumentamos o número de processadores, mostrando novamente um desempenho bom em relação ao algoritmo seqüencial.

3.4 Notas

Neste capítulo apresentamos o algoritmo *Ordenação_Bitônica* que apresentou ganhos significativos de desempenho em relação ao algoritmo seqüencial. Além disso, o desempenho do algoritmo melhora à medida em que o número de processadores aumenta.

Outra característica interessante deste algoritmo é que as trocas de mensagens são

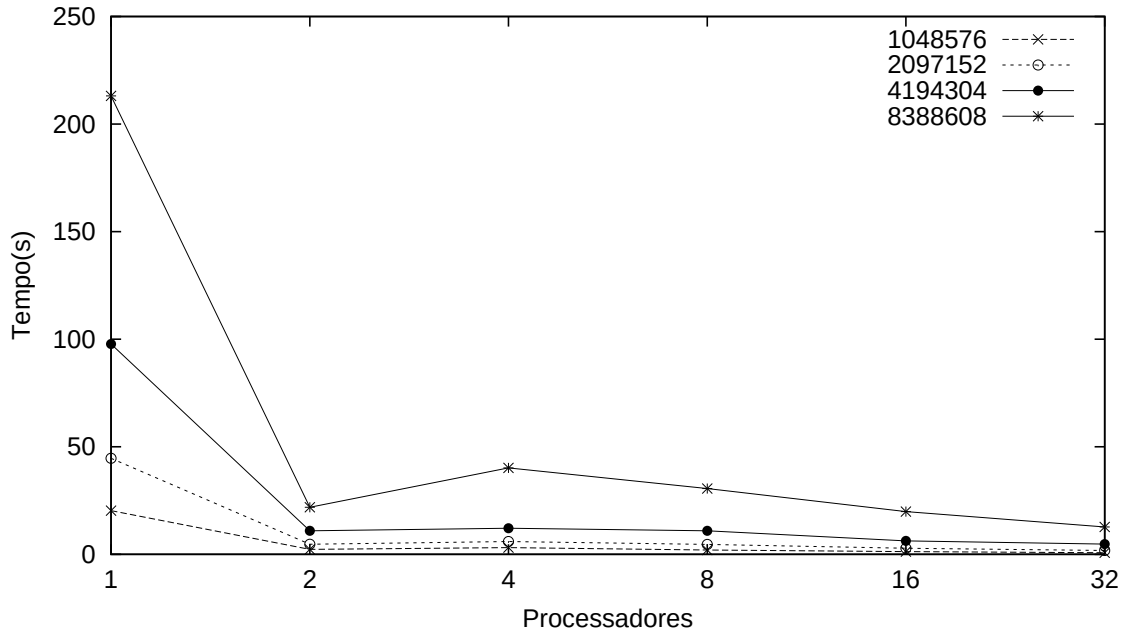


Figura 3.4: Tempo de execução do algoritmo *Ordenação_Bitônica,* $1048576 \leq n \leq 8388608$.

sempre executadas entre pares de processadores e, em cada rodada de comunicação são enviados exatamente $\frac{n}{p}$ dados.

Além disso, este algoritmo é de fácil entendimento e implemenmtação e pode ser adaptado para ser aplicado em problemas de ordenação externa, pois cada processador sempre armazena $\frac{n}{p}$ elementos.

Este algoritmo foi descrito e implementado em diversos outros modelos, como, por exemplo, no modelo LogP [11], porém não encontramos na literatura referências à implementação deste algoritmo no modelo BSP/CGM.

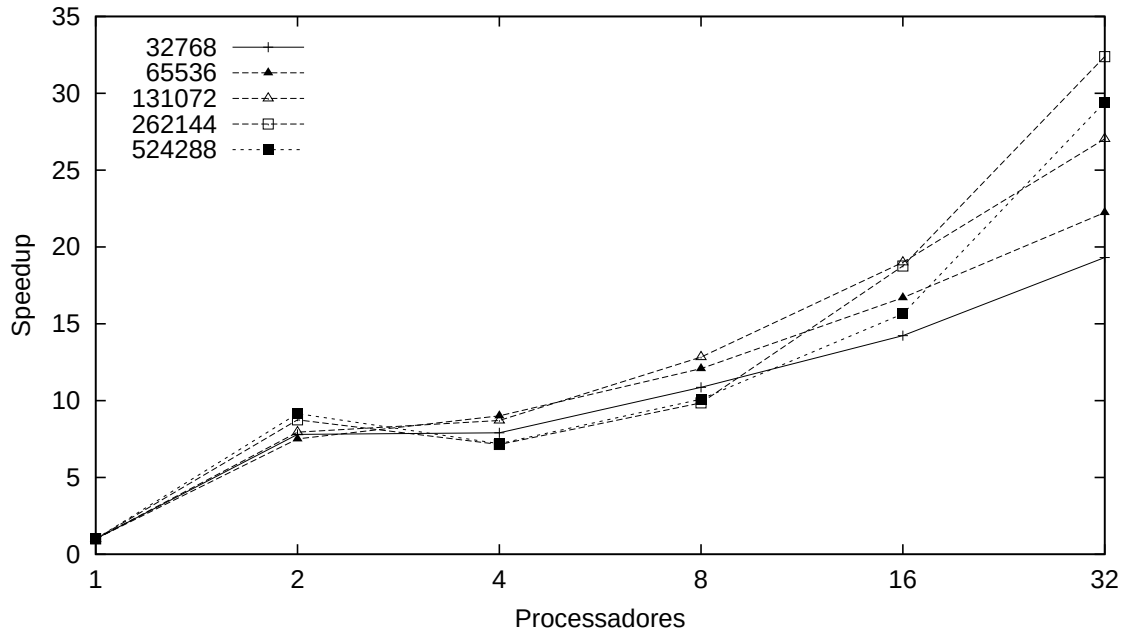


Figura 3.5: *Speedup* do algoritmo *Ordenação_Bitônica,* $32768 \leq n \leq 524288$.

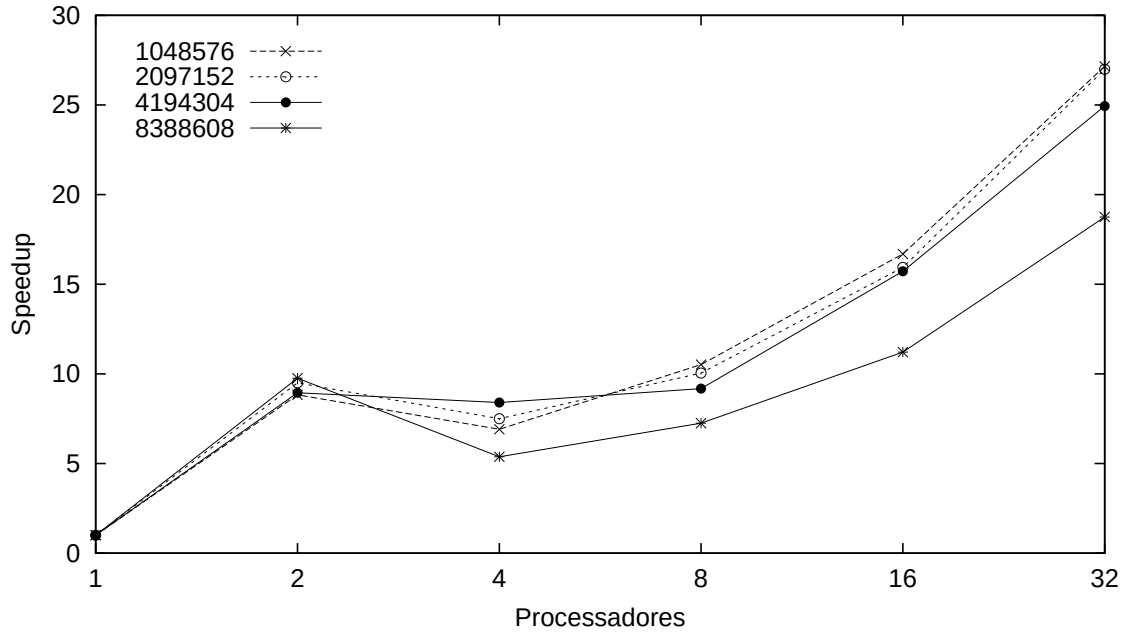


Figura 3.6: *Speedup* do algoritmo *Ordenação_Bitônica,* $1048576 \leq n \leq 8388608$.

Capítulo 4

Ordenação de Chan & Dehne

O algoritmo que vamos apresentar neste capítulo foi proposto por Chan e Dehne [4], e baseia-se no método de ordenação por amostragem, combinado com um algoritmo de ordenação básico para computação local.

Apresentamos inicialmente, na Seção 4.1, um algoritmo para ordenação onde $\frac{n}{p} \geq p^2$, onde n é o tamanho da entrada e p é o número de processadores utilizados.

Na Seção 4.2, apresentamos o algoritmo proposto por Chan e Dehne [4], onde $\frac{n}{p} \geq p$. Nas Seções 4.3 e 4.4, descrevemos a implementação e os testes realizados, bem como uma descrição sobre os resultados obtidos.

4.1 Algoritmo Ordenação_CD

O algoritmo *Ordenação_CD*, também conhecido como *regular sampling*, pode ser dividido em três fases principais:

- Baseado em divisores locais, um conjunto de $p-1$ divisores globais é calculado. Os divisores globais são elementos que dividem os $\frac{n}{p}$ elementos de cada processador em p intervalos.
- Cada processador divide seus $\frac{n}{p}$ elementos em p cestos, de acordo com os divisores globais, e envia seu $i-$ésimo cesto ao processador P_i. Um cesto é um conjunto de elementos delimitados por um mesmo intervalo.
- Cada processador ordena localmente seus cestos.

No algoritmo que apresentaremos mais detalhadamente a seguir, os divisores são obtidos de forma determinística como veremos na descrição. Entretanto, pode-se obter divisores de forma aleatória.

Algoritmo: Ordenação_CD
Entrada: n elementos divididos em p processadores, com $\frac{n}{p}$ elementos em cada proces-

sador.
Saída: os elementos ordenados, distribuídos entre os processadores.

1. Cada processador P_i ordena localmente seus $\frac{n}{p}$ dados, com $1 \leq i \leq p$, usando um algoritmo de tempo $O(n \log n)$.
2. Cada processador P_i seleciona uma amostra local S_i de p elementos com índice $j\frac{n}{p^2}$, como $1 \leq i \leq p$ e $0 \leq j \leq p-1$.
3. Cada processador P_i envia o S_i para processador P_1, onde $1 \leq i \leq p$.
4. Processador P_1 ordena os elementos recebidos no passo anterior, utilizando um algoritmo de tempo $\Omega(n \log n)$, e armazena no vetor S.
5. Processador P_1 seleciona uma amostra G de p elementos contidos em S, com identificador jp, onde $1 \leq j \leq p$.
6. Processador P_1 envia G para todos os processadores.
7. Cada processador P_i separa seu conjunto de $\frac{n}{p}$ dados em cestos $B_{i,j}$, contendo os elementos entre o $(j-1)$-ésimo e o j-ésimo elementos da amostra G recebida de P_1, onde $1 < j \leq p$.
8. Cada processador P_i envia o cesto $B_{i,j}$ ao processador P_j, onde $1 \leq i, j \leq p$.
9. Cada processador P_i ordena o conjunto de elementos recebido no passo anterior, denominado de R_i, utilizando um algoritmo seqüencial de tempo $O(n \log n)$, e calcula $r_i =\mid R_i \mid$, onde $1 \leq i \leq p$.
10. Cada processador P_i envia r_i para P_1, $1 \leq i \leq p$.
11. P_1 calcula, para cada processador P_j, um vetor A_j, contendo a quantidade e para quais processadores devem ser enviados alguns elementos, sem que a ordem dos elementos seja alterada. Este passo é necessário para distribuir de forma balanceada os elementos pelos processadores.
12. P_1 envia A_j para processador P_j, $1 \leq j \leq p$.
13. Baseado em A_j, cada processador P_j envia alguns de seus elementos para os demais processadores.

fim algoritmo

Teorema 4.1 *O algoritmo* Ordenação_CD *ordena corretamente n elementos armazenados em um BSP/CGM de p processadores,* $\frac{n}{p}$ *elementos por processador,* $\frac{n}{p} \geq p^2$.

Prova: O algoritmo *Ordenação_CD* utiliza a idéia do algoritmo de ordenação por amostragem, combinado com um algoritmo seqüencial de ordenação para os passos de computação local. A idéia do algoritmo é inicialmente fazer com que os n elementos sejam separados em p intervalos, denominados de cestos. No final do algoritmo, todos os elementos do

i-ésimo cesto devem estar armazenados no processador P_i e que todos os elementos do processador P_j devem ser menores que os elementos do processador P_k, se $j < k$, onde $1 \leq i, j, k \leq p$.

Para isso, inicialmente o algoritmo determina os limites de cada intervalo, delimitados pelos divisores globais G, que são calculados no Passo 5 e distribuídos a todos os processadores no Passo 6.

Após o Passo 6, cada processador P_i divide seu conjunto de inteiros em cestos $B_{i,j}$, de acordo com os valores da amostra global G, de tal forma que no cesto $B_{i,j}$, todos os elementos sejam maiores ou iguais ao $(j-1)$-ésimo elemento de G e menores que o j-ésimo elemento de G, onde $1 \leq i, j \leq p$. Neste instante, cada processador possui seus $\frac{n}{p}$ inteiros divididos em p cestos (Passo 7). Mais ainda, em cada processador P_i. $1 \leq i \leq p$, temos que os elementos do cesto $B_{i,j}$ são menores que os elementos do cesto $B_{i,k}$, se $j < k$, $1 \leq j < k \leq p$.

No próximo passo do algoritmo os elementos do cesto i de todos os processadores são enviados ao processador P_i, $1 \leq i \leq p$, ou seja, todos os processadores possuem seus elementos dentro dos limites utilizados na determinação dos cestos. Mais ainda, temos que os elementos do processador P_j são menores que os elementos do processador P_k, se $j < k$. Uma preocupação neste passo é o balanceamento de carga, ou seja, garantir que nenhum processador receba muitos elementos. Segundo Helman *et al.* [6], o número máximo de elementos que um processador recebe é $2(\frac{n}{p})$.

No Passo 9, cada processador ordena localmente os dados recebidos anteriormente. Após este passo, todos os n elementos distribuídos pelos p processadores estão ordenados. Como a quantidade de dados recebidos no passo anterior pode ser diferente em cada processador, nos Passos 10, 11, 12 e 13 é executado um balanceamento, para que que cada processador possua exatamente $\frac{n}{p}$ elementos, sem que a ordem dos elementos seja alterada, mantendo os n elementos ordenados.

Assim, o algoritmo *Ordenação_CD* ordena n inteiros divididos entre os p processadores.

□

Teorema 4.2 *O algoritmo* Ordenação_CD *utiliza* $O(\frac{n}{p})$ *de memória local por processador, 6 rodadas de comunicação (2-*$\frac{n}{p}$ *relações e 4-*p^2 *relações), e tempo de computação local* $O(\frac{n \log n}{p})$ *para ordenar* n *elementos distribuídos entre* p *processadores.*

Prova. Para provar o Teorema 4.2, mostraremos inicialmente o tempo gasto com computação local e depois o tempo gasto com comunicação entre os processadores.

No Passo 1 do algoritmo, é realizada um ordenação local, gastando tempo de computação local $O(\frac{n \log n}{p})$. Após a ordenação local, para calcular os divisores locais no Passo 2, é necessário $O(p)$ de computação local. A ordenação do Passo 4 leva tempo $O(p^2 \log p)$. A determinação dos divisores globais (Passo 5) gasta tempo $O(p)$.

No Passo 7, cada processador divide seus elementos em cestos, gastando $O(\frac{n}{p})$. No Passo 9, são ordenados os elementos recebidos no Passo 8. Como cada processador recebe $O(\frac{n}{p})$ elementos [6] , esta ordenação gasta $O(\frac{n \log n}{p})$.

Como $\frac{n}{p} \geq p^2$, temos que $\frac{n}{p} \log \frac{n}{p} \geq p^2 \log p^2$. Assim, o algoritmo *Ordenação_CD* ordena n inteiros distribuídos entre p processadores utilizando tempo de computação local igual a $O(\frac{n \log n}{p})$.

Mostraremos agora o tempo que o algoritmo *Ordenação_CD* gasta com comunicação entre os processadores.

O algoritmo *Ordenação_CD* utiliza uma rodada de comunicação em cada um dos Passos 3, 6, 8, 10, 12, 13. Logo, o algoritmo gasta 6 rodadas de comunicação para ordenar os n elementos.

Portanto, o algoritmo *Ordenação_CD* faz a ordenação de n elementos divididos entre p processadores usando 6 rodadas de comunicação e tempo de computação local $O(\frac{n \log n}{p})$.

□

É importante observar que caso o conjunto de dados seja composto somente de inteiros, podemos utilizar localmente um algoritmo linear para ordenação de inteiros, usando 6 rodadas de comunicação e tempo de computação local $O(\frac{n}{p})$ para ordenar n inteiros, divididos entre p processadores. Dessa forma, o algoritmo ordena n inteiros distribuídos entre p processadores em tempo $O(\frac{n}{p})$ usando 6 rodadas de comunicação.

Para ilustrar o funcionamento do algoritmo *Ordenação_CD*, observemos a Figura 4.1.

Na Figura 4.1(a), temos os dados distribuídos entre os processadores com as amostras locais de cada um dos processadores em destaque. Em seguida, cada um dos processadores envia sua amostra para P_1. O vetor com as amostras recebidas e já ordenadas encontra-se na Figura 4.1(b). O processador P_1 seleciona uma amostra G de p inteiros, como mostram as células hachuradas da Figura 4.1(b). O processador P_1 envia a amostra G para todos os processadores P_i, que dividem seus $\frac{n}{p}$ dados em p cestos $B_{i,j}$. Cada processador i envia seu cesto $B_{i,j}$ para o processador P_j, $1 \leq i, j \leq 4$. A Figura 4.1(c) mostra os cestos recebidos por cada um dos processadores.. Cada um dos processadores ordena localmente os inteiros recebidos e envia para P_1, a quantidade de inteiros recebida, denominada de r_i.

No exemplo da Figura 4.1, $r_1 = 7$, $r_2 = 11$, $r_3 = 8$, $r_4 = 6$. Baseados nestes valores, o processador P_0 calcula, para cada processador P_j, um vetor A_j, contendo a quantidade de elementos a serem enviados para cada um dos processadores, como mostra a Figura 4.1(d). Nesta figura, por exemplo, A_2 =(1, 0, 2, 0), ou seja, o processador P_2 deve enviar um elemento para P_1 e dois para P_3. Cada processador j recebe então seu vetor A_j e envia os elementos determinados para os processadores adequados. Após esta operação, todos os elementos estão ordenados e cada processador possui exatamente $\frac{n}{p}$ elementos.

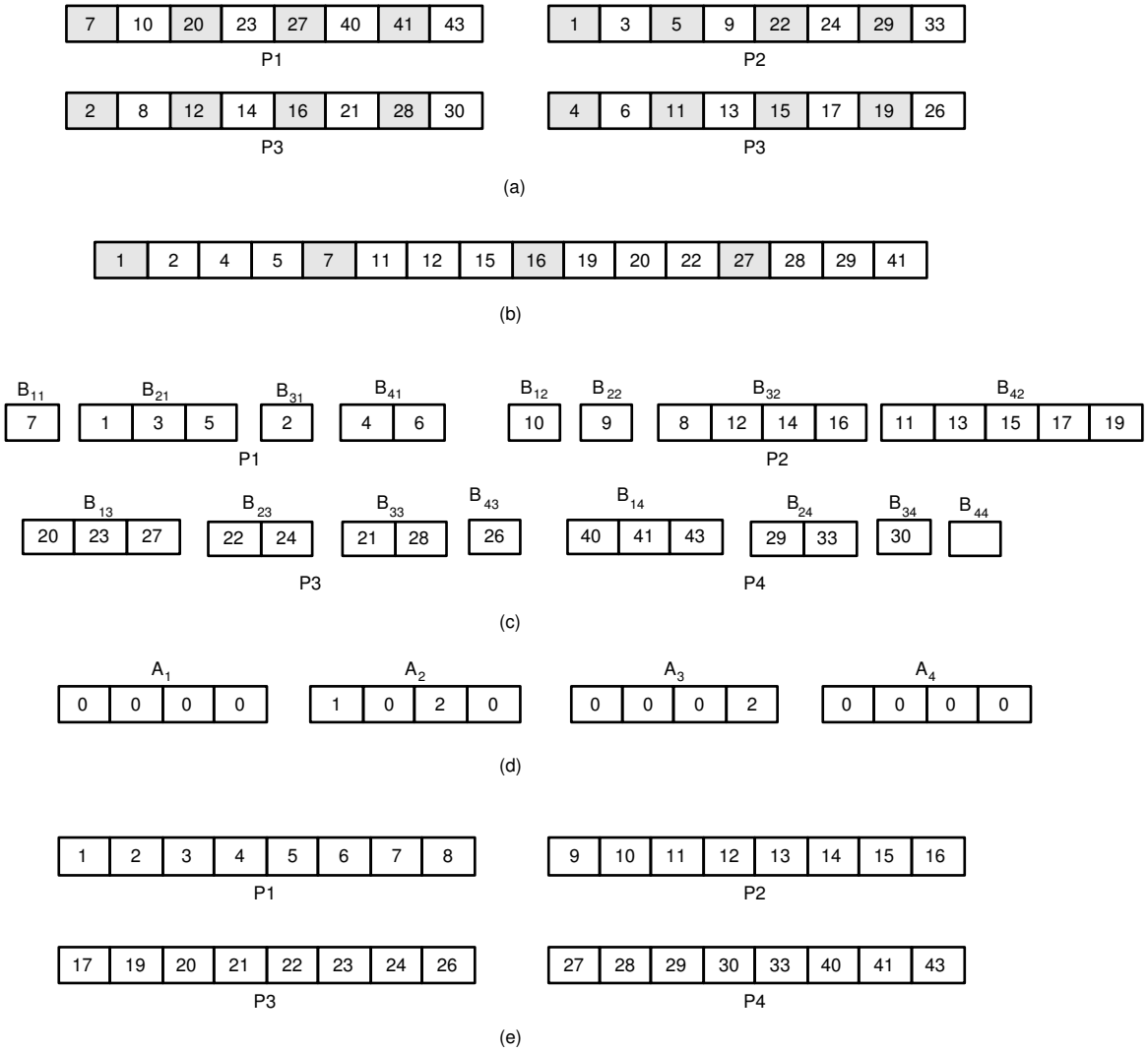


Figura 4.1: Exemplo de execução do algoritmo Ordenação_CD.

4.2 Algoritmo Ordenação_CGM

O algoritmo que descreveremos a seguir utiliza a idéia de particionar os p processadores em $\sqrt{p}$ grupos de mesmo tamanho e realizar permutações entre os elementos a serem ordenados de tal forma que teremos todos os elementos do grupo G_i menores que os elementos no grupo G_j, se $i < j$. Para realizar a ordenação dos grupos, aplicamos o algoritmo *Ordenação_Amostragem* em cada um dos $\sqrt{p}$ grupos. Este algoritmo utiliza a idéia de dividir os processadores em grupos para garantir que $\frac{n}{p} \geq p$.

Algoritmo: Ordenação_CGM
Entrada: n elementos divididos em p processadores, com $\frac{n}{p}$ elementos em cada processador, com $\frac{n}{p} \geq p$.
Saída: os elementos ordenados, divididos em grupos de processadores.

1. Agrupe os p processadores em $\sqrt{p}$ grupos G_1, G_2, ..., $G_{\sqrt{p}}$.

2. Em cada um dos grupos obtidos, aplique o algoritmo *Ordenação_CD*.

3. Cada processador obtém seu menor elemento, denominado de *min_local* e o envia para P_1.

4. P_1 ordena os elementos recebidos no Passo 3 e seleciona $\sqrt{p}$ inteiros com identificador $i\sqrt{p}$, denominados de divisores globais. Em seguida, envia os divisores globais para todos os processadores.

5. Cada processador P_i divide seu conjunto de $\frac{n}{p}$ inteiros em $\sqrt{p}$ cestos $B_{i,j}$, contendo os elementos entre $(j-1)$-ésimo e o j-ésimo elementos dos divisores globais, onde $1 \leq j \leq \sqrt{p}$.

6. P_i envia $B_{i,j}$ para um processador no grupo G_j, onde $1 \leq i \leq p$ e $1 \leq j \leq \sqrt{p}$. Denominaremos de R'_j, o conjunto de inteiros enviados para o grupo G_j.

7. Em cada um dos grupos G_i, calcula-se o tamanho do conjunto de inteiros enviados ao grupo G_j, $1 \leq j \leq \sqrt{p}$, denominado de $t_{i,j}$.

8. Todos $t_{i,j}$ e os tamanhos de $B_{i,j}$ são enviados para um processador em cada grupo, denominado de processador principal.

9. Em cada grupo, o processador principal computa uma escala de roteamento para realizar balanceamento na distribuição dos dados entre os processadores do grupo e a envia aos demais processadores do grupo.

10. Usando o algoritmo *Ordenação_CD*, cada grupo G_j, $1 \leq j \leq \sqrt{j}$, ordena R'_j

11. Uma operação de balanceamento é executada de forma análoga aos Passos 11, 12 e 13 do Algoritmo *Ordenação_CD*, porém, utilizando duas fases como nos Passos 7, 8 e 9.

fim algoritmo

Teorema 4.3 *O algoritmo* Ordenação_CGM *ordena n inteiros armazenados em um BSP/CGM de p processadores,* $\frac{n}{p}$ *inteiros por processador, onde* $\frac{n}{p} \geq p$.

Prova. No Passo 1, os processadores são divididos em $\sqrt{p}$ grupos com $\sqrt{p}$ processadores em cada. Temos que $n' = \frac{n}{\sqrt{p}}$ e $p' = \sqrt{p}$. Logo, $\frac{n'}{p'} = \frac{n/\sqrt{p}}{\sqrt{p}} = \frac{n}{p} \geq p = p'^2$. Assim, o algoritmo *Ordenação_CD* pode ser aplicado em cada um dos grupos. Após ter todos os elementos ordenados em cada um dos grupos, o algoritmo calcula os divisores globais G (Passos 2 e 3), e baseado nos divisores globais, cada processador divide seu conjunto de dados em $\sqrt{p}$ cestos. Então, após o Passo 5, cada processador P_i possui seu conjunto de elementos divididos em $\sqrt{p}$ cestos, $B_{i,j}$, contendo elementos entre o $(j-1)$-ésimo e o j-ésimo valor de G, $1 \leq j \leq \sqrt{p}$.

Em seguida, cada processador P_i envia seu cesto $B_{i,j}$ para um processador no grupo G_j. Assim, neste instante, temos todos os elementos entre o $(j-1)$-ésimo e o j-ésimo

divisor global armazenados nos processadores do grupo G_j, e temos também que todos os elementos do grupo G_k são menores que os elementos do grupo G_l, se $k < l$.

Para ordenar os elementos dentro dos grupos, novamente é utilizado o algoritmo *Ordenação_Amostragem*. Assim, temos todos os grupos com seus elementos ordenados. Como os elementos do grupo k são menores que os elementos do grupo l, se $k < l$, existe também uma ordenação entre os grupos de processadores. Logo, todos os n elementos estão ordenados e divididos entre os processadores. Por fim, nos Passos 11, 12 e 13 é realizado um balanceamento, sem alteração da ordem dos dados para que todos os processadores possuam $\frac{n}{p}$ inteiros.

Portanto, o algoritmo *Ordenação_CGM* ordena um conjunto de n elementos distribuídos por p processadores, $\frac{n}{p}$ elementos por processador.

□

Teorema 4.4 *O algoritmo* Ordenação_CGM *utiliza 23 rodadas de comunicação (6-$\frac{n}{p}$ relações e 18 p-relações)* $O(\frac{n}{p})$ *de memória local por processador e tempo de computação local* $O(\frac{n \log n}{p})$.

Prova. Mostraremos inicialmente o tempo gasto com computação local e depois o tempo gasto com comunicação entre os processadores.

Os Passos 2 e 10 do algoritmo *Ordenação_CGM*, gastam $O(\frac{n \log n}{p})$ de computação local, como mostrado no Teorema 4.1. O Passo 3 do algoritmo gasta $O(\frac{n}{p})$ de computação local, enquanto o Passo 4 gasta $O(p \log p)$. Como $\frac{n}{p} \geq p$, temos $p \log p \leq \frac{n}{p} \log \frac{n}{p} = O(\frac{n \log n}{p})$. Para a divisão dos inteiros em cestos, realizada no Passo 5, o algoritmo leva tempo de computação local igual a $O(\frac{n}{p})$. Logo, o tempo de computação local do algoritmo *Ordenação_CGM* é $O(\frac{n \log n}{p})$.

Para a comunicação, temos que os Passos 2 e 10 utilizam 6 rodadas de comunicação cada um, como mostrado no Teorema 4.1. O Passo 3 utiliza 1 rodada de comunicação, o Passo 4 utiliza 2 rodadas de comunicação, o Passo 6 utiliza 1 rodada de comunicação, Passo 8 gasta 2 rodadas de comunicação. No Passo 9, o algoritmo gasta 1 rodada de comunicação e no Passo 11, são necessárias 4 rodadas de comunicação. Assim, o algoritmo *Ordenação_CGM* utiliza 23 rodadas de comunicação.

Portanto, o algoritmo *Ordenação_CGM* ordena n elementos divididos em p processadores, $\frac{n}{p}$ inteiros por processador, utilizando $O(\frac{n \log n}{p})$ de computação local e 23 rodadas de comunicação.

□

A Figura 4.2 ilustra o funcionamento do algoritmo *Ordenação_CGM*.

Inicialmente, temos os 4 processadores divididos em dois grupos: G_1, composto por P_1 e P_2; e G_2, composto por P_3 e P_4. Em seguida, o algoritmo *Ordenação_CD* é aplicado em cada um dos grupos separadamente. Assim, dentro dos grupos temos os elementos ordenados. Após a ordenação dos elementos dentro dos grupos, cada processador seleciona

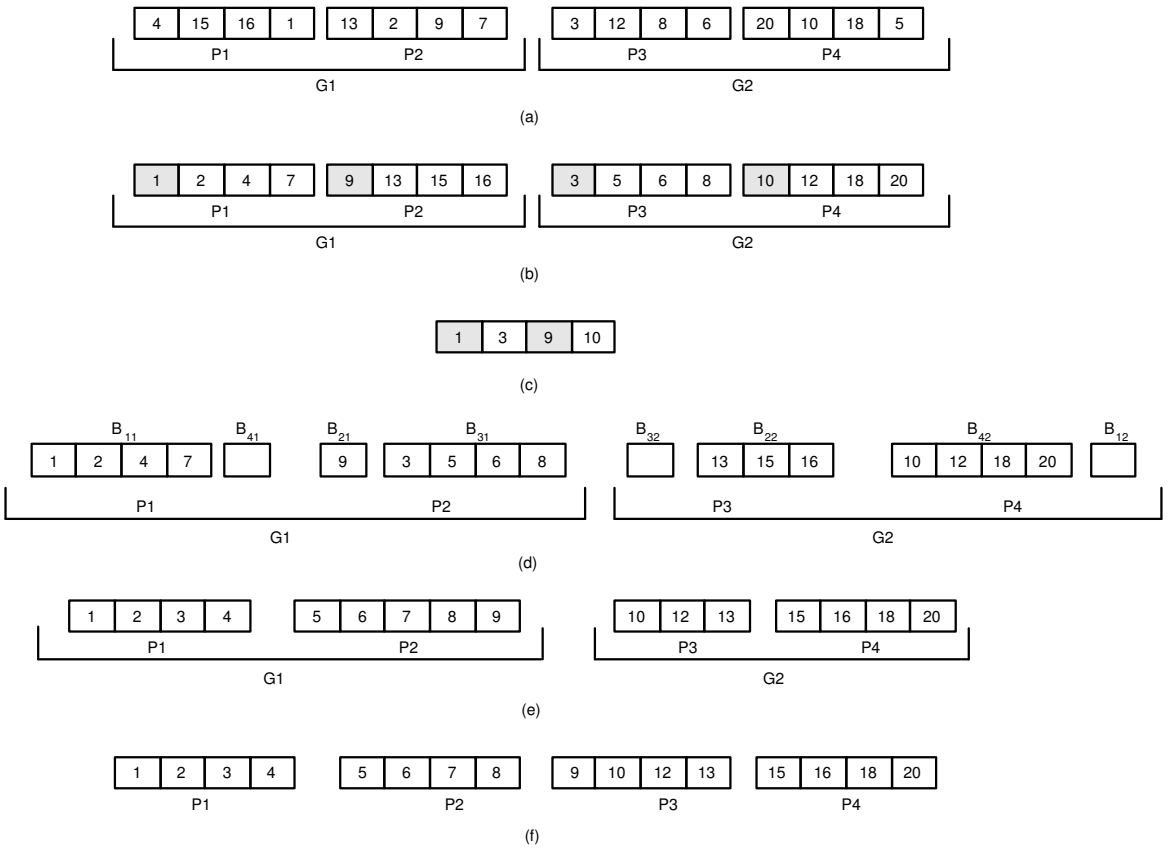


Figura 4.2: Exemplo de execução do algoritmo Ordenação_CGM.

seu menor elemento (Figura 4.2(b)) e envia para P_1. O processador P_1 ordena os elementos recebidos e escolhe uma amostra de $\sqrt{p}$ inteiros, e envia para todos os processadores, como mostrado na Figura 4.2(c).

Após receber a amostra global de P_1, cada processador divide seus $\frac{n}{p}$ dados em cestos $B_{i,j}$, e envia para um processador no grupo G_j. Por exemplo, na Figura 4.2(d), o processador P_1 permaneceu com o cesto $B_{1,1}$ e enviou o cesto $B_{1,2}$ para o processador P_4, que pertence ao grupo G_2. Analogamente, o processador P_3 enviou seu cesto $B_{3,1}$ para o processador P_2, que pertence ao grupo G_1.

Cada um dos grupos G_1 e G_2 executa novamente o algoritmo *Ordenação_CD*, obtendo como resultado a Figura 4.2(e). Por fim, é realizado um balanceamento, onde o processador P_2 envia um elemento ao processador P_3, fazendo com que todos os processadores tenham $\frac{n}{p}$ dados (Figura 4.2(f)).

4.3 Implementação e Resultados

O algoritmo *Ordenação_CD* apresentado na Seção 4.1 foi implementado na linguagem C/C++ em conjunto com a biblioteca de troca de mesagens MPI. O algoritmo *Or-*

denação_CGM não foi implementado por utilizar mais rodadas de comunciação que o algoritmo *Ordenação_CD*.

O algoritmo de ordenação usado localmente em cada processador foi o *quicksort*. O *quicksort* foi utilizado por ser um algoritmo eficiente, além de permitir que qualquer conjunto de dados, e não somente inteiros, possa ser ordenado.

Os tempos obtidos para ordenação foram medidos em segundos, após a distribuição dos dados. A entrada do programa é um arquivo texto, contendo um número inteiro que indica a quantidade de elementos e os elementos a serem ordenados. Os arquivos de entrada foram gerados aleatoriamente e para cada entrada foram utilizados 1, 2, 4, 8, 16 e 32 processadores.

Para melhor visualização dos resultados obtidos dividimos os gráficos contendo os tempos de execução e *speedup* em duas partes. Foram realizados 30 experimentos dos quais obtivemos o maior tempo utilizado para construir os gráficos das Figuras 4.3 a 4.6.

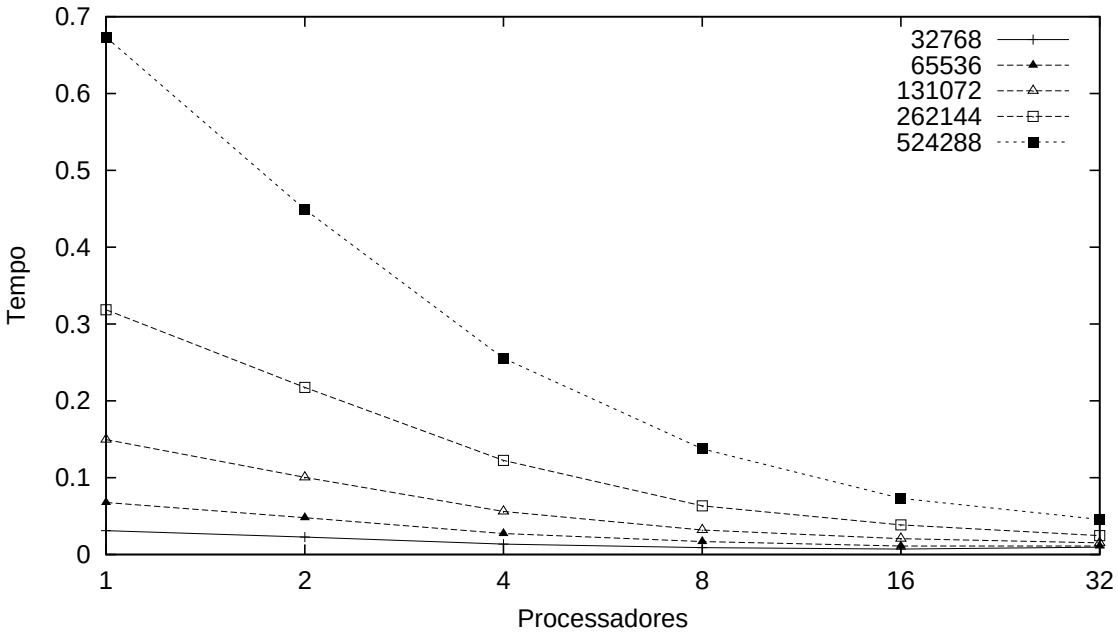


Figura 4.3: Tempo de execução do algoritmo *Ordenação_CD*, $32768 \leq n \leq 524288$.

Nas Figuras 4.3 e 4.4, apresentemos os tempos de execução obtidos através de experimentos do algoritmo *Ordenação_CD* para $32768 \leq n \leq 524288$ e $1048576 \leq n \leq 8388608$, respectivamente. Podemos observar que em todos os casos, não importando o valor de n, à medida que aumentamos o número de processadores, o tempo de execução diminui. Isto ocorre, porque o número de rodadas de comunicação é constante, independente dos valores de n e p e o tempo gasto com computação local em cada processador também é pequeno.

Nas Figuras 4.5 e 4.6, mostramos o *speedup* do algoritmo *Ordenação_CD* para $32768 \leq n \leq 524288$ e $1048576 \leq n \leq 8388608$, respectivamente. Observamos que quanto maior o tamanho da entrada, maior o valor do *speedup*, ou seja, melhor o desempenho do algoritmo.

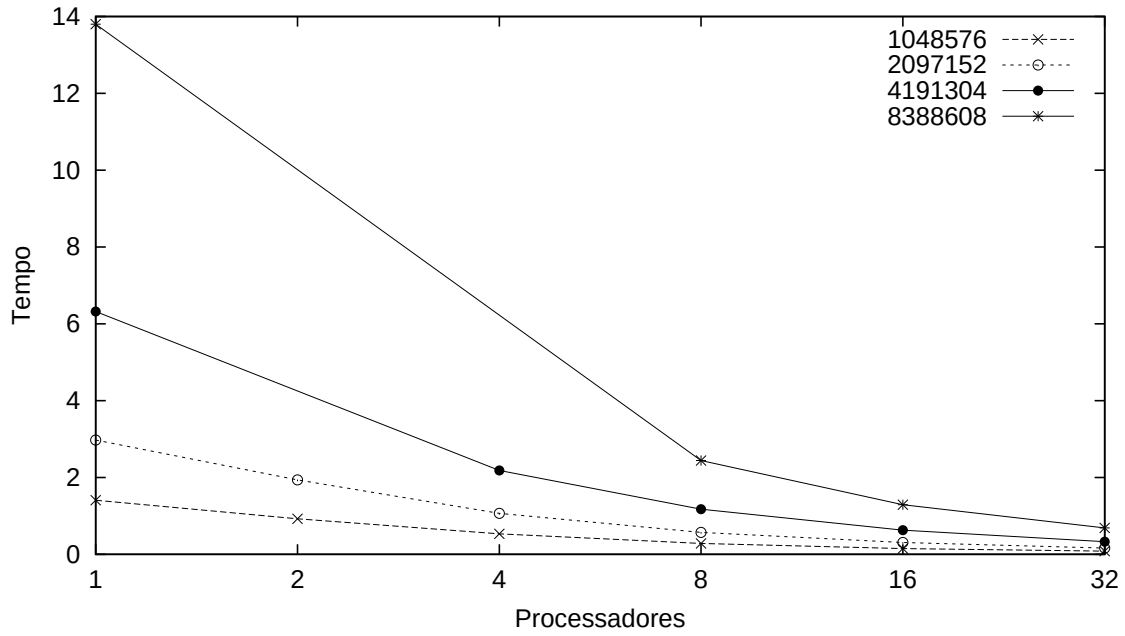


Figura 4.4: Tempo de execução do algoritmo *Ordenação_CD*, $1048576 \leq n \leq 8388608$.

Podemos notar ainda que para tamanhos de entrada pequenos, o ganho de desempenho é menor, pois a porcentagem de tempo gasto com comunicação em relação ao tempo total é muito maior nestes casos. Isto ocorre porque o número de rodadas de comunicação é sempre fixo, independente do valor de n ou de p.

4.4 Notas

Neste capítulo apresentamos dois algoritmos de ordenação paralelos. Inicialmente, mostramos um algoritmo paralelo, denominado de *Ordenação_CD*. Este algoritmo utiliza 6 rodadas de comunicação e tempo de computação local $O(\frac{n \log n}{p})$ para ordenar n elementos distribuídos por p processadores, onde $\frac{n}{p} \geq p^2$, e utiliza a idéia de dividir os elementos em cestos. Em seguida apresentamos um algoritmo, denominado de *Ordenação_CGM*, que utiliza a idéia de agrupar os processadores para realizar a ordenação, para garantir que $\frac{n}{p} \geq p$. Este algoritmo utiliza 23 rodadas de comunicação e tempo de computação local $O(\frac{n \log n}{p})$. Por fim, apresentamos a implementação e os resultados obtidos com o algoritmo *Ordenação_CD*. Pelos testes realizados e pelos resultados, este algoritmo apresenta um bom desempenho, principalmente quando aumentamos o tamanho da entrada n.

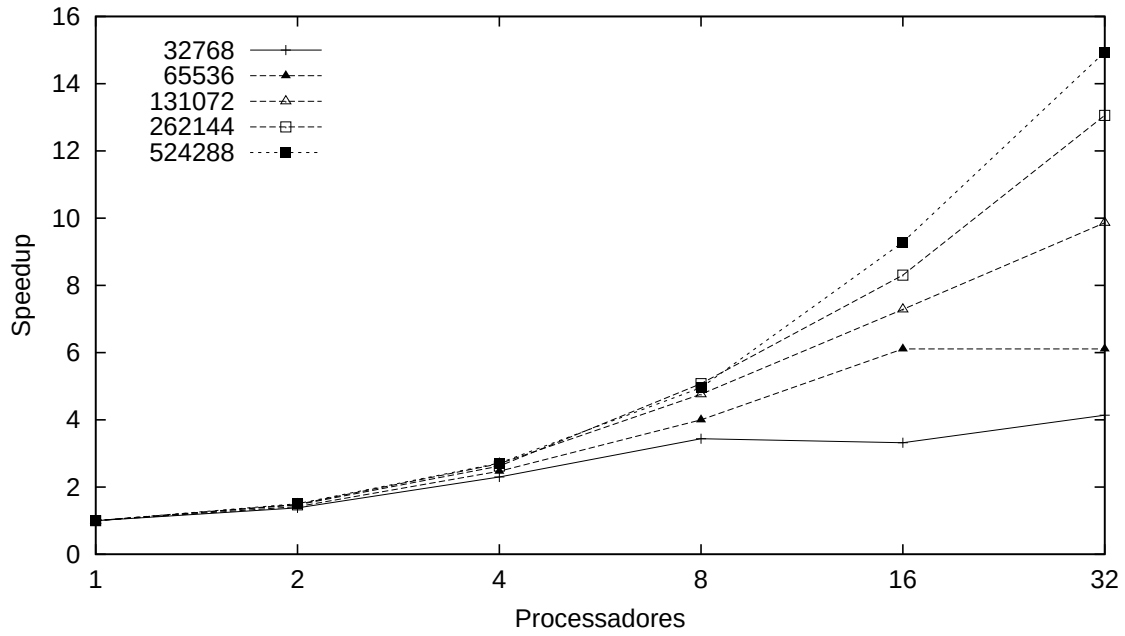


Figura 4.5: *Speedup* do algoritmo *Ordenação_CD,* $32768 \leq n \leq 524288$.

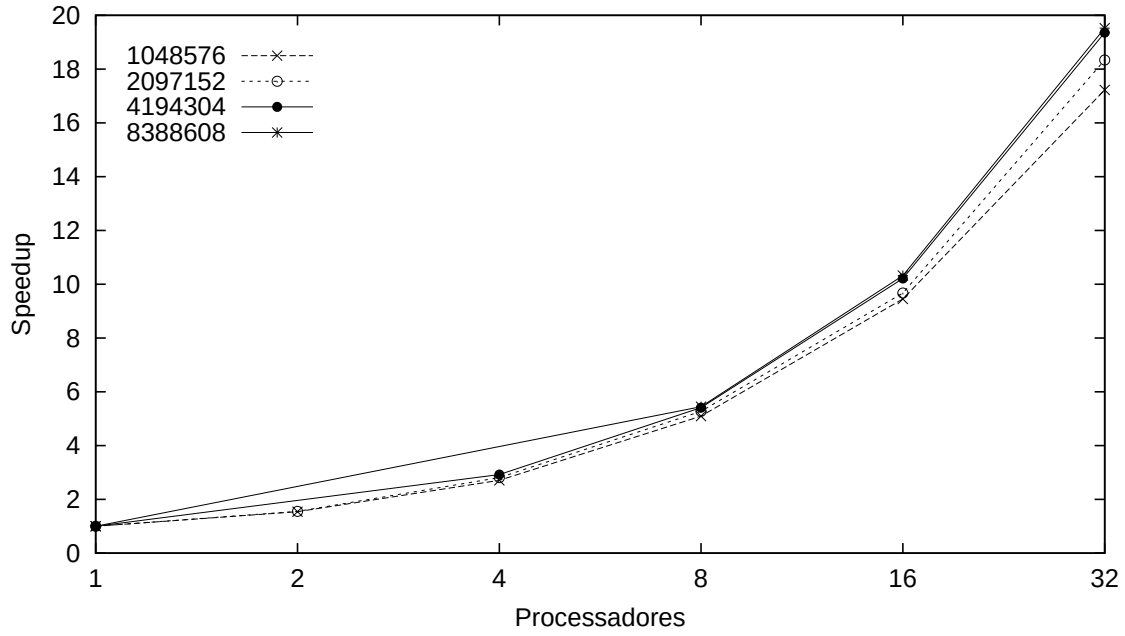


Figura 4.6: *Speedup* do algoritmo *Ordenação_CD,* $1048576 \leq n \leq 8388608$.

Capítulo 5

Ordenação por Divisão

Neste capítulo apresentaremos um algoritmo que consiste em dividir um conjunto de números em cestos, e distribuir os cestos de forma adequada, para que se possa ordenar n números divididos por p processadores, utilizando $O(1)$ rodadas de comunicação para $\frac{n}{p} \geq p^2$.

Na divisão dos cestos, utilizamos a idéia de calcular um conjunto de divisores, denominados de p-quartis, baseado no cálculo das medianas de um conjunto de elementos. Os algoritmos *Ordenação_por_Amostragem* apresentado no Capítulo 5 e o algoritmo a ser apresentado neste capítulo diferem na forma como o conjunto de divisores são calculados.

Além disso, no algoritmo *Ordenação_por_Divisão* [12], a ser apresentado detalhadamente na Seção 5.1, não há um balanceamento após a ordenação dos dados. Entretanto, este passo poderia ser adicionado, sem que a corretude do algoritmo fosse alterada. Porém, o número de rodadas de comunicação seria alterado.

Apresentaremos ainda, nas Seções 5.2 e 5.3, os resultados obtidos com a implementação e as notas finais, respectivamente.

5.1 Descrição do Algoritmo

No algoritmo *Ordenação_por_Divisão* inicialmente cada processador divide localmente seus $\frac{n}{p}$ números em p subconjuntos, baseados em um conjunto de divisores globais. Este conjunto define p intervalos, que serão utilizados por cada um dos cestos de cada processador. Dessa forma, os elementos do cesto i de um processador estão no mesmo intervalo dos demais elementos do cesto i dos demais processadores.

Por fim, cada processador envia seu i-ésimo cesto para o processador i, fazendo com que os elementos do processador P_j sejam menores que os elementos do processador P_k, se $j < k$.

É importante observarmos que, no algoritmo paralelo que descreveremos a seguir, queremos que, após o envio dos cestos, a quantidade de elementos em cada processador

seja a mais próxima possível. Para isso, fazemos o cálculo dos divisores globais baseado em um conjunto de divisores locais, calculados separadamente em cada um dos processadores. Em cada processador, os divisores locais separam os $\frac{n}{p}$ elementos em p cestos de mesmo tamanho.

Definição 5.1 *A mediana de um conjunto ordenado de n números é o* $((n+1)/2)$*-ésimo elemento de n para n ímpar ou a média aritmética entre o n-ésimo e o* $(n+1)$*-ésimo elemento para n par.*

Definição 5.2 *Dado um conjunto ordenado A de tamanho n, definimos os p-quartis de A como os* $p-1$ *elementos de índice* $\frac{n}{p}$, $\frac{2n}{p}$, ..., $\frac{(p-1)n}{p}$, *que dividem A em p partes de mesmo tamanho.*

Para computar de forma seqüencial os p-quartis de um conjunto de n elementos, podemos usar o algoritmo *p_quartis* de tempo $O(n \log p)$, descrito a seguir.

Algoritmo: *p*_**quartis**
Entrada: um vetor A com n elementos e p.
Saída: os p-quartis de A.

1. Calcule a mediana de A.
2. Usando a mediana de A, divida A em dois subconjuntos A_1 e A_2.
3. Aplique o algoritmo recursivamente até que $p-1$ divisores sejam encontrados.

fim algoritmo

A seguir descreveremos um algoritmo CGM para calcular os divisores de um vetor, particionando o conjunto de números a ser ordenado em p subconjuntos.

O algoritmo *p_quartis* é utilizado pelo algoritmo CGM na computação dos divisores globais. Para computar os divisores globais, utilizamos o algoritmo *p-quartis_CGM* descrito a seguir.

Algoritmo: *p*-**quartis_CGM**
Entrada: um vetor A com n elementos, divididos pelos p processadores, $\frac{n}{p}$ elementos em cada.
Saída: os divisores globais de A.

1. Cada processador P_i calcula seqüencialmente seus p-quartis, utilizando o algoritmo *p_quartis*. Denominaremos os p-quartis calculados por P_i de Q_i, $1 \leq i \leq p$.
2. Todos os processadores P_i enviam Q_i para processador P_1, $1 \leq i \leq p$. Denominaremos de Q, os elementos recebidos por P_1.
3. P_1 calcula os p-quartis de Q, obtendo os divisores globais.

fim algoritmo

Teorema 5.1 *O algoritmo* p-quartis_CGM *computa os divisores globais de um conjunto de* n *elementos divididos em* p *processadores,* $\frac{n}{p} \geq p^2$*, em* $O(1)$ *rodadas de comunicação (1* p*-relação) e tempo de computação local* $O(\frac{n \log p}{p})$*.*

Prova. O algoritmo *p-quartis_ CGM* utiliza $\frac{n}{p} \log p$ para que cada processador calcule os p-quartis locais no Passo 1. No Passo 4 são necessários $p^2 \log p$. Logo, o tempo de computação local necessário é $O(\frac{n \log p}{p})$.

Para calcular os divisores globais, o algoritmo utiliza 1 rodada de comunicação, no Passo 2.

Portanto, o algoritmo *Divisor_ CGM* utiliza 1 rodada de comunicação e tempo de computação local $O(\frac{n \log p}{p})$ para calcular os divisores globais de um vetor A em paralelo.

□

Para ilustrar o funcionamento do algoritmo *p-quartis_ CGM*, considere a Figura 5.1.

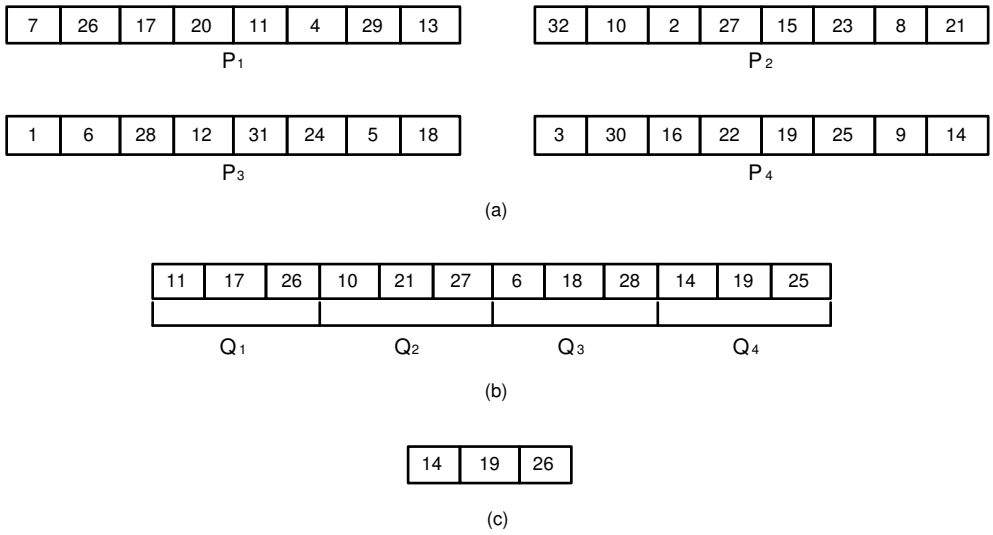


Figura 5.1: Algoritmo *p-quartis_ CGM.*

Na Figura 5.1(a) mostramos os elementos armazenados inicialmente em cada processador. Em seguida, cada um dos processadores calcula seqüencialmente seus p-quartis e envia para P_1. A Figura 5.1(b) mostra os elementos recebidos por P_1. Em seguida, P_1 calcula seqüencialmente os p-quartis destes elementos, como mostrado na Figura 5.1(c).

Após calcular o conjunto de divisores, apresentaremos um algoritmo CGM que ordena um conjunto de números dividindo-os em cestos.

Algoritmo: Ordenação_por_Divisão
Entrada: um vetor A com n elementos, divididos pelos p processadores, $\frac{n}{p}$ elementos em cada.
Saída: todos os elementos de A ordenados, divididos pelos processadores.

1. Compute o conjunto divisor S, utilizando o algoritmo *p-quartis_CGM*.
2. Processador P_1 envia S para todos os processadores.
3. Cada processador P_i, particiona seus elementos em cestos $B_{i,j}$ de acordo com os elementos de S, $1 \leq i, j \leq p$.
4. Cada processador P_i envia seu cesto $B_{i,j}$ para o processador P_j, $1 \leq i, j \leq p$.
5. Cada processador P_i faz a ordenação dos dados recebidos no passo anterior.

fim algoritmo

Teorema 5.2 *O algoritmo* Ordenação_por_Divisão *ordena corretamente n inteiros distribuídos por p processadores.*

Prova. O algoritmo *Ordenação_por_Divisão* utiliza a idéia de dividir seus elementos em p intervalos. Para isso, inicialmente o algoritmo calcula um conjunto de $p-1$ divisores, denominado de S. Este conjunto de elementos é enviado a todos os processadores pelo processador P_1.

Em seguida, cada processador divide seus $\frac{n}{p}$ números, em p cestos, de forma que todos os elementos do cesto j, são maiores ou iguais ao $(j-1)$-ésimo e menores que o j-ésimo elemento de S, $1 \leq j \leq p$. Assim, os j-ésimos cestos de todos os processadores estão delimitados pelo mesmo intervalo.

No Passo 4, cada processador P_i envia seu cesto $B_{i,j}$ para o processador P_j, fazendo com que os elementos do processador P_k sejam menores que os elementos do processador P_l, se $k < l$.

Por fim, cada processador executa uma ordenação local dos elementos recebidos. Como os dados estão ordenados localmente e os elementos do processador P_k são menores que os elementos do processador P_l, se $k < l$, então os n elementos estão ordenados.

Portanto, o algoritmo *Ordenação_por_Divisão* ordena corretamente n elementos distribuídos por p processadores.

□

Teorema 5.3 *O algoritmo* Ordenação_por_Divisão *ordena n inteiros distribuídos por p processadores, utilizando 3 rodadas de comunicação (2 p-relações e 1 $\frac{n}{p}$-relação) e tempo de computação local $O(\frac{n \log n}{p})$.*

Prova. Para provarmos o Teorema 5.3 temos que provar o tempo gasto com comunicação e o tempo gasto com computação local.

Provaremos inicialmente o tempo gasto com computação local. No Passo 1, o algoritmo gasta $O(\frac{n \log p}{p})$. No Passo 3, o algoritmo gasta $\frac{n}{p}$. Por fim, no Passo 5, cada um dos processadores gasta $O(\frac{n}{p} \log \frac{n}{p}) = O(\frac{n \log n}{p})$.

Mostraremos o tempo gasto com comunicação entre os processadores. No Passo 1, o algoritmo utiliza 1 rodada de comunicação, como mostrado no Teorema 5.2. Os Passos 2 e 4 utilizam uma rodada de comunicação cada.

Portanto, o algoritmo *Ordenação_por_Divisão* ordena n números divididos entre p processadores, utilizando 3 rodadas de comunicação e tempo de computação local $O(\frac{n \log n}{p})$.

□

Para ilustrar o funcionamento do algoritmo *Ordenação_por_Divisão*, considere a Figura 5.2.

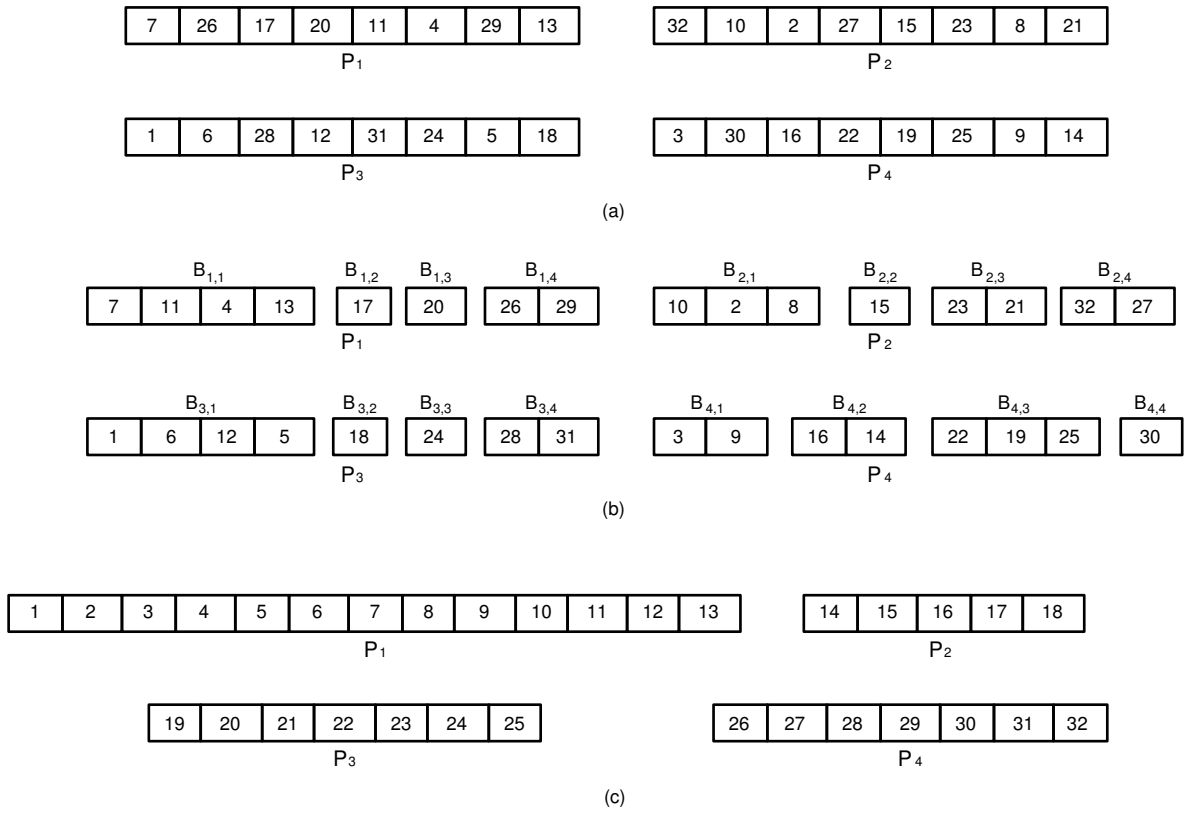


Figura 5.2: Algoritmo *Ordenação_por_Divisão.*

Na Figura 5.2, vamos considerar que os divisores já foram calculados e enviados a todos os processadores, ou seja, os Passos 1 e 2 do algoritmo já foram executados. O conjunto de divisores a ser utilizado é mostrado na Figura 5.2*(c)*.

Baseados nos divisores recebidos de P_1, cada um dos processadores P_i divide seus elementos em p cestos (Figura 5.2(b)). Em seguida, cada um dos processadores P_i envia seu cesto $B_{i,j}$ para o processador P_j, $1 \leq i, j \leq p$. Por exemplo, $B_{1,2}$, foi enviado por P_1 ao processador P_2.

Após todos os cestos serem enviados, cada processador ordena localmente os dados recebidos no Passo 4, como mostra a Figura 5.2*(c)*.

5.2 Implementação e Resultados

O algoritmo *Ordenação_por_Divisão* apresentado na Seção 5.1 foi implementado na linguagem C/C++ em conjunto com a biblioteca de troca de mensagens MPI.

Os tempos obtidos para ordenação foram medidos em segundos, após a distribuição dos dados. A entrada do programa é um arquivo texto, contendo um número inteiro que indica a quantidade de elementos e os elementos a serem ordenados. Os arquivos de entrada foram gerados aleatoriamente e para cada entrada foram utilizados 1, 2, 4, 8, 16 e 32 processadores.

Para melhor visualização dos resultados obtidos dividimos os gráficos contendo os tempos de execução e *speedup* em duas partes. Foram realizados 30 experimentos. Os gráficos das Figuras 5.3 a 5.6 foram construídos utilizando o maior tempo obtido nos experimentos. O menor tempo, tempo médio e tempos médios gastos com computação local e comunicação são apresentados no Apêndice C.

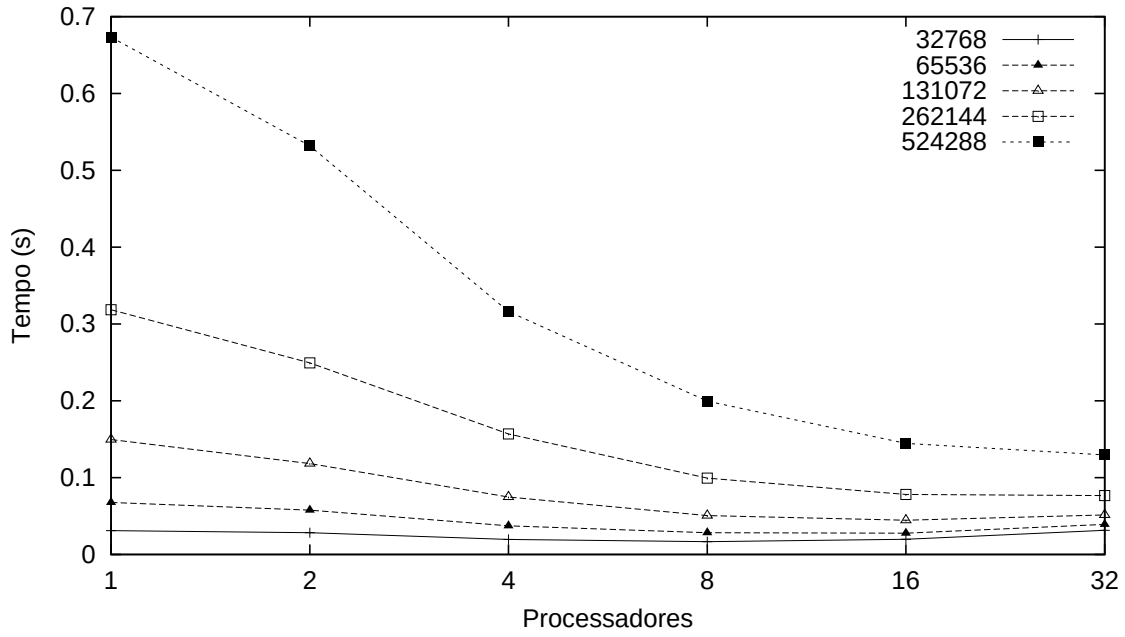


Figura 5.3: Tempo de execução do algoritmo *Ordenação_por_Divisão,* $32768 \leq n \leq 524288$.

Nas Figuras 5.3 e 5.4, apresentemos os tempos de execução obtidos dos experimentos realizados para o algoritmo *Ordenação_por_Divisão* para $32768 \leq n \leq 524288$ e $1048576 \leq n \leq 8388608$, respectivamente. Podemos observar que os tempos de execução para valores pequenos de n ($32768 \leq n \leq 262144$), mesmo aumentando o número de processadores, o tempo de execução não diminui significativamente. Além disso, em alguns casos, como por exemplo para $n = 32768$ e 32 processadores, ao aumentarmos o número de processadores, o tempo de execução também aumenta, ou seja, o algoritmo apresenta um desempenho pior com mais processadores.Isto ocorre porque nestes casos

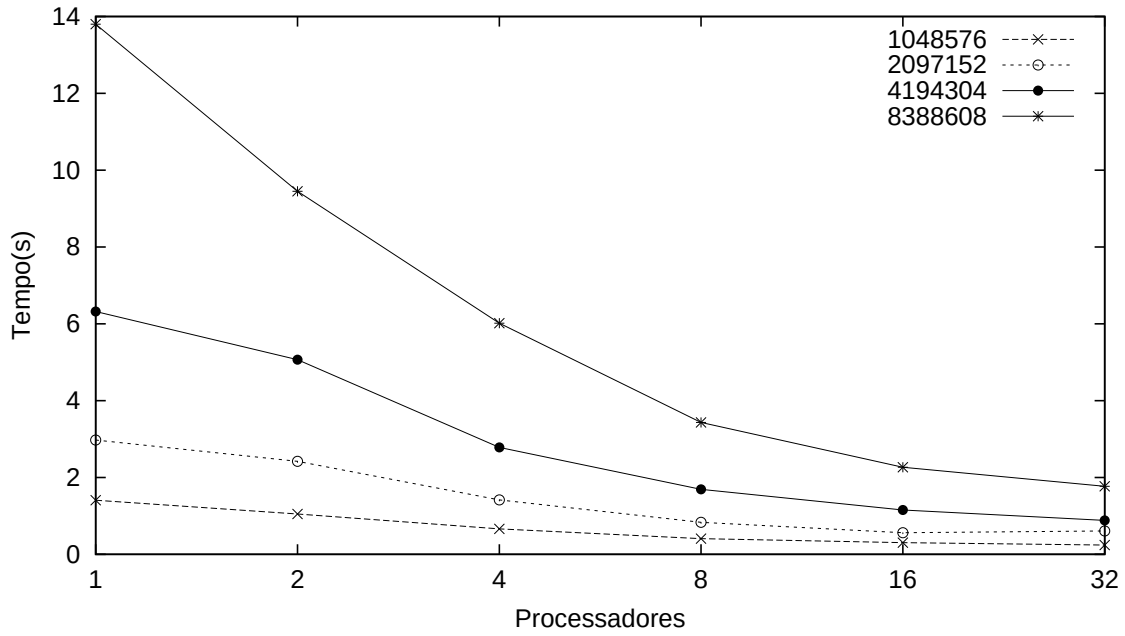


Figura 5.4: Tempo de execução do algoritmo *Ordenação_por_Divisão,* $1048576 \leq n \leq 8388608$.

a porcentagem de tempo gasto com comunicação entre os processadores é maior que o tempo de processamento.

Entretanto, quando o valor de n aumenta ($525288 \leq n \leq 8388608$), ao aumentarmos o número de processadores, o desempenho do algoritmo melhora em todos os casos, ou seja, o tempo de execução do algoritmo diminui. Isto ocorre pelo fato de termos sempre um número constante de rodadas de comunicação. Assim, ao aumentarmos o número de processadores, temos o mesmo número de rodads de comunicação, porém o tempo gasto com computação local diminui, pois temos mais processadores realizando a operação de ordenação.

Nas Figuras 5.5 e 5.6 apresentamos o *speedup* do algoritmo *Ordenação_por_Divisão* para $32768 \leq n \leq 524288$ e $1048576 \leq n \leq 8388608$, respectivamente.

Podemos notar que quanto maior o valor da entrada, maior o valor do *speedup*, ou seja, ao aumentarmos o tamanho da entrada, o algoritmo tem um melhor desempenho. Mais ainda, podemos observar que, para valores pequenos de n ($32768 \leq n \leq 262144$), com 32 processadores, o tempo de execução do algoritmo aumenta em relação a 16 processadores. Como o tamanho da entrada é pequeno, o algoritmo passa mais tempo realizando comunicação entre processadores do que processando informações.

Porém, notamos que para valores maiores de n, sempre que aumentamos o número de processadores, o valor do *speedup* aumenta. Ou seja, o desempenho do algoritmo é melhor quando temos valores maiores de n. Isto acontece devido ao fato do número de rodadas de comunicação ser constante e termos mais processadores envolvidos na execução do problema, diminuindo o tempo de computação local. Conseqüentemente, o tempo total

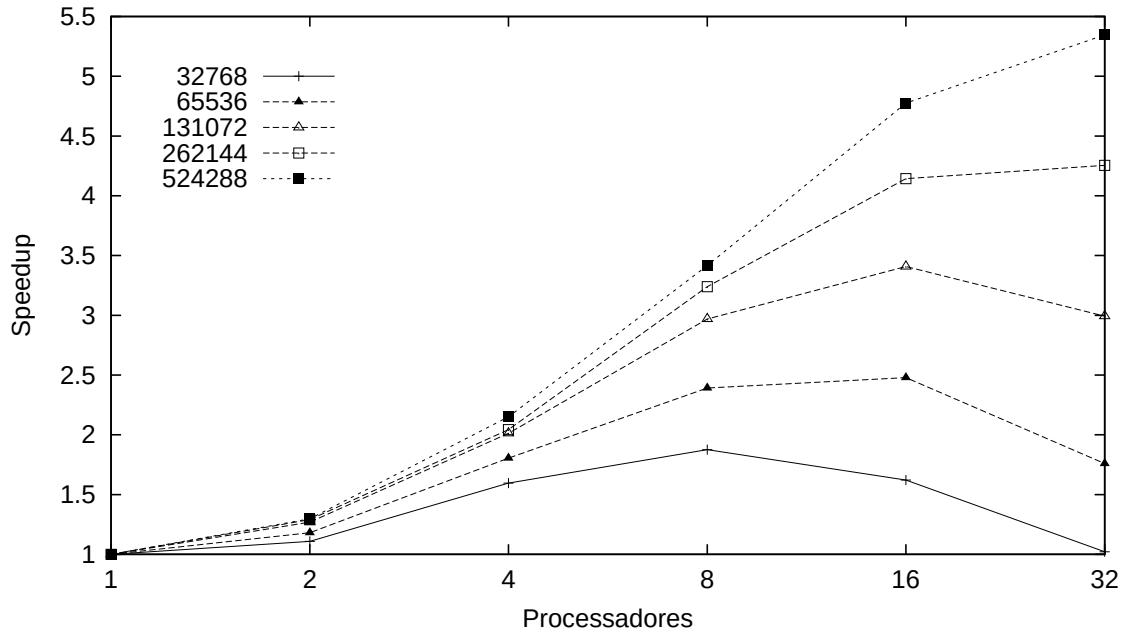


Figura 5.5: *Speedup* do algoritmo *Ordenação_por_Divisão,* $32768 \leq n \leq 524288$.

de execução do algoritmo diminui.

5.3 Notas

Neste capítulo apresentamos o algoritmo de ordenação por divisão, onde os elementos são separados em p intervalos e cada intervalo é enviado a um processador. Este algoritmo utiliza 3 rodadas de comunicação e tempo de computação local $O(\frac{n \log n}{p})$. Experimentalmente, o algoritmo apresenta um bom desempenho quando o tamanho da entrada aumenta.

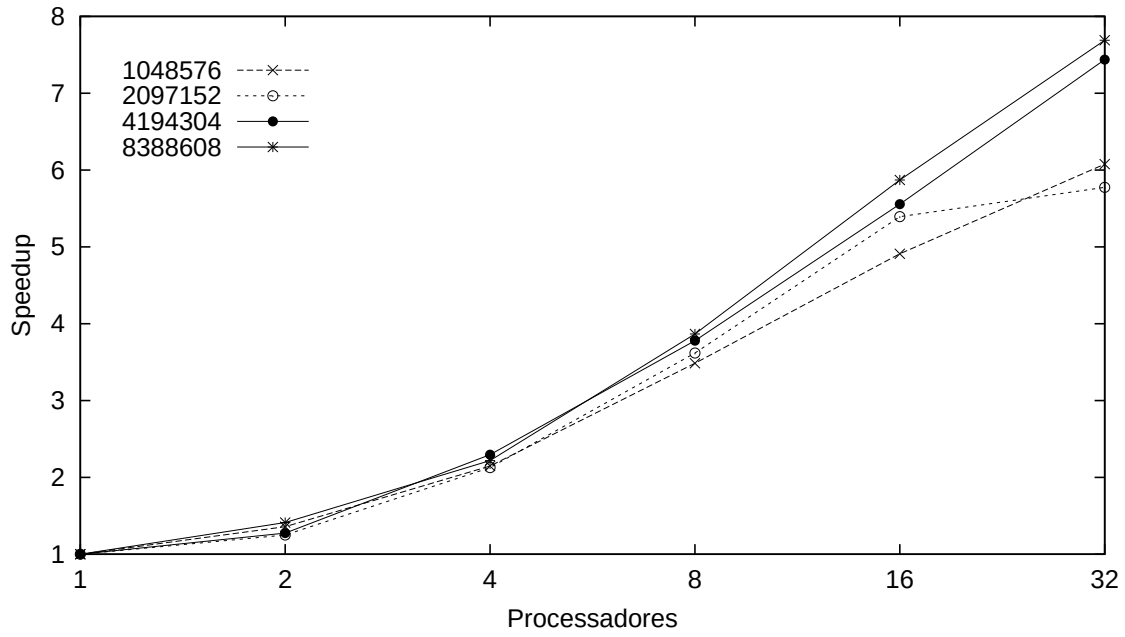


Figura 5.6: *Speedup* do algoritmo *Ordenação_por_Divisão*, $1048567 \leq n \leq 8388608$.

Capítulo 6

Conclusão

O problema da ordenação é um problema muito estudado na área de computação. O nosso objetivo neste trabalho é implementar alguns algoritmos paralelos de ordenação para utilizar na resolução de problemas que têm a ordenação como sub-rotina. Desta forma, além do desempenho do algoritmo, levamos também em consideração a facilidade de implementação, o espaço utilizado, dentre outros fatores.

O algoritmo *Ordenação_Bitônica* apesar de não possuir um bom desempenho em relação ao tempo de execução, é um algoritmo de fácil entendimento e implementação, além de apresentar um ganho significativo em relação ao algoritmo de ordenação bitônica seqüencial. Além disso, neste algoritmo, temos sempre $O(\frac{n}{p})$ elementos em cada processador, o que permite que o mesmo seja facilmente adaptado para ser utilizado em problemas que utilizam ordenação externa. Outra característica interessante deste algoritmo é que a troca de mensagens é realizada sempre entre pares de processadores e cada processador envia e recebe sempre $\frac{n}{p}$ elementos em cada rodada de comunicação.

De acordo com os resultados experimentais apresentados no Capítulo 5, a implementação do algoritmo *Ordenação_CD* apresenta um bom desempenho. Podemos observar que independente do tamanho da entrada, ao aumentarmos o número de processadores, o tempo de execução do algoritmo diminui. Por exemplo, com 32 processadores, o algoritmo *Ordenação_CD* executa 17 vezes mais rápido do que o algoritmo seqüencial para entradas grandes ($1048576 \leq n \leq 8388608$). Assim, este algoritmo pode ser utilizado como sub-rotina de problemas cujo desempenho seja um parâmetro importante.

O algoritmo *Ordenação_por_Divisão*, de acordo com os resultados experimentais, apresenta um bom desempenho quando o tamanho da entrada é grande. Porém, quando temos entradas pequenas ($32768 \leq n \leq 262144$), o algoritmo não tem um bom desempenho ao utilizarmos 32 processadores. Para o tamanho de entrada $n = 8388608$, o *speedup* alcançado foi de 7.68.

Dentre os algoritmos implementados, o que apresentou melhor desempenho de acordo com os resultados experimentais foi o algoritmo *Ordenação_CD* apresentado no Capítulo 5. Mesmo apresentando a mesma complexidade teórica, os algoritmos *Ordenação_CD* e *Ordenação_por_Divisão* não apresentam o mesmo desempenho na prática. Isto ocorre em

conseqüência da forma como o conjunto de divisores é calculado nos dois algoritmos. No algoritmo *Ordenação_por_Amostragem*, em cada um dos p processadores, os $\frac{n}{p}$ elementos são ordenados para o cálculo dos divisores locais. Conseqüentemente, a divisão dos elementos em cestos, bem como a junção dos cestos recebidos, pode ser implementada de forma mais simples e eficiente que no algoritmo *Ordenação_por_Divisão*. Por exemplo, como os elementos estão ordenados, basta realizar uma intercalação dos cestos, ao invés de uma ordenação, como ocorre no algoritmo *Ordenação_por_Divisão*.

O algoritmo *Ordenação_Bitônica* não apresenta um bom desempenho, mas pode ser aplicado em casos onde o desempenho não é fundamental, mas sim o espaço de armazenamento disponível, como por exemplo, na ordenação externa. Isto é possível porque neste algoritmo, sempre temos a mesma quantidade de dados em cada processador.

Trabalhos futuros incluem a implementação destes algoritmos utilizando o paradigma da orientação a objetos para a construção de uma classe que envolva algoritmos BSP/CGM para ordenação.

Além disso, poderíamos tentar melhorar o algoritmo *Ordenação_Bitônica* diminuindo a quantidade de informações trocadas entre os processadores, enviando apenas as informações necessárias para a realização da operação de divisão bitônica local. Em conseqüência disso, também conseguimos diminuir o tempo de computação local, já que temos menos dados para a realização da operação local.

Em casos onde os elementos sejam inteiros, podemos utilizar algoritmos de tempo linear para as ordenações locais, reduzindo os tempos gastos com computação local.

A implementação de outros algoritmos que apresentaram bom desempenho quando descritos em outros modelos, como o *radix sort*, apresentado por Cueller *et al.* [2], também poderia ser incluída como trabalho futuro.

Apêndice A

Ordenação Bitônica: Tempos de Execução

As tabelas listadas a seguir apresentam o menor tempo, o maior tempo, os tempos médios, tempos de computação local e comunicação obtidos em nossos experimentos, executado em um Beowulf de 64 processadores Pentium III de 500 MHz, cada um com 256 MB de memória RAM. Todos os nós estavam interconectados através de um switch Fast Ethernet. Cada nó executava o sistema operacional Linux RedHat 7.3 com g++ 2.96 e MPI/LAM 6.5.6.

Processadores	**Menor Tempo**	**Tempo Médio**	**Maior Tempo**
1	0.335572	0.33568	0.336224
2	0.042978	0.0430501	0.043313
4	0.0423541	0.0424521	0.0425708
8	0.03073	0.0308869	0.034677
16	0.0224969	0.0235939	0.0250459
32	0.0173788	0.0177067	0.0181248

Tabela A.1: Tempo de execução em segundos do algoritmo *Ordenação_Bitônica* para n = 32768.

Processadores	**Menor Tempo**	**Tempo Médio**	**Maior Tempo**
1	0.758402	0.759123	0.76433
2	0.101028	0.101133	0.101227
4	0.0840461	0.0842008	0.084619
8	0.0625141	0.0628176	0.0636492
16	0.044426	0.0454712	0.0467191
32	0.0326321	0.0341297	0.036468

Tabela A.2: Tempo de execução em segundos do algoritmo *Ordenação_Bitônica* para $n = 65536$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	1.7824	1.78552	1.79002
2	0.224449	0.224715	0.225461
4	0.204493	0.204693	0.205093
8	0.132763	0.138455	0.139174
16	0.090831	0.0922247	0.09395
32	0.0622721	0.0660557	0.068692

Tabela A.3: Tempo de execução em segundos do algoritmo *Ordenação_Bitônica* para n = 131072.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	4.14511	4.14586	4.14667
2	0.472377	0.473531	0.475107
4	0.567995	0.58047	0.595842
8	0.420118	0.420635	0.421302
16	0.221	0.226899	0.234798
32	0.128007	0.141614	0.151576

Tabela A.4: Tempo de execução em segundos do algoritmo *Ordenação_Bitônica* para n = 262144.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	9.20911	9.21156	9.21375
2	0.998025	1.00605	1.02869
4	1.22867	1.27988	1.342
8	0.891891	0.912474	0.93929
16	0.579864	0.588572	0.614278
32	0.277624	0.313142	0.328195

Tabela A.5: Tempo de execução em segundos do algoritmo *Ordenação_Bitônica* para n = 524288.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	20.316	20.3253	20.3387
2	2.15176	2.30095	2.31111
4	2.90732	2.93997	3.1127
8	1.88966	1.93281	1.98947
16	1.19543	1.21803	1.24327
32	0.739168	0.748479	0.772222

Tabela A.6: Tempo de execução em segundos do algoritmo *Ordenação_Bitônica* para n = 1048576.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	44.6138	44.6206	44.6264
2	4.68034	4.6952	4.70728
4	5.93708	5.95007	5.96567
8	4.37892	4.4408	4.6327
16	2.72517	2.79939	2.84903
32	1.62253	1.65297	1.76022

Tabela A.7: Tempo de execução em segundos do algoritmo *Ordenação_Bitônica* para n = 2097152.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	97.5997	97.6327	97.77
2	10.8868	10.9113	10.9222
4	11.337	11.6221	12.1227
8	10.3021	10.6412	10.9514
16	6.14982	6.2098	6.23863
32	3.61524	3.91549	4.767682

Tabela A.8: Tempo de execução em segundos do algoritmo *Ordenação_Bitônica* para n = 4194304.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	212.679	212.707	213.098
2	21.7779	21.7984	21.8531
4	38.8659	39.6048	40.1978
8	28.4144	29.3437	30.6131
16	18.0515	18.9676	19.8506
32	10.7314	11.3462	12.7122

Tabela A.9: Tempo de execução em segundos do algoritmo *Ordenação_Bitônica* para n = 8388608.

Processadores	n = **32768**	n = **65536**	n = **131072**
1	1	1	1
2	7.7974267	7.506184	7.9457090
4	7.9072648	9.014856	8.7059041
8	10.8680379	12.084559	12.8294077
16	14.2274062	16.694589	19.0050026
32	19.3154878	22.242299	27.0305212

Tabela A.10: *Speedup* do algoritmo *Ordenação_Bitônica*, $32768 \leq n \leq 131072$.

Processadores	n=**262144**	n=**524288**	n = **1048576**
1	1	1	1
2	8.7552029	9.1561652	8.8334383
4	7.1422468	7.1972059	6.9134378
8	9.8561936	10.0951479	10.5159327
16	18.7595475	15.6506935	16.6870274
32	32.3877600	29.4165586	27.1554712

Tabela A.11: *Speedup* do algoritmo *Ordenação_Bitônica*, $262144 \leq n \leq 1048576$.

Processadores	n = **2097152**	n = **4194304**	n = **8388608**
1	1	1	1
2	9.5034503	8.9478522	9.7579180
4	7.4991722	8.4006074	5.3707378
8	10.0478742	9.1749708	7.2488132
16	15.9394010	15.7223582	11.2142284
32	26.9941983	24.9349889	18.7469813

Tabela A.12: *Speedup* do algoritmo *Ordenação_Bitônica*, $2097152 \leq n \leq 8388608$.

Processadores	**Computação Local**	**Comunicação**
1	0.33568	0.0
2	0.0451705	0.0133022
4	0.0377657	0.0189157
8	0.0251301	0.0172364
16	0.0112355	0.022775
32	0.00670443	0.0219461

Tabela A.13: Tempo em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_Bitônica* para $n = 32768$.

Processadores	Computação Local	Comunicação
1	0.759123	0.0
2	0.0852617	0.031448
4	0.0722563	0.0348004
8	0.048468	0.0317061
16	0.0248305	0.0411221
32	0.0147134	0.0427737

Tabela A.14: Tempo em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_Bitônica* para $n = 65536$.

Processadores	Computação Local	Comunicação
1	1.78552	0.0
2	0.218868	0.0569869
4	0.167672	0.0667196
8	0.130777	0.0639052
16	0.103205	0.0572067
32	0.0932296	0.0572842

Tabela A.15: Tempo em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_Bitônica* para $n = 131072$.

Processadores	Computação Local	Comunicação
1	4.14586	0.0
2	0.741553	0.0917763
4	0.578092	0.130436
8	0.374679	0.112221
16	0.278103	0.087132
32	0.2041	0.0857901

Tabela A.16: Tempo em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_Bitônica* para $n = 262144$.

Processadores	Computação Local	Comunicação
1	9.21156	0.0
2	1.73308	0.156386
4	1.34338	0.293431
8	0.799823	0.261167
16	0.249944	0.47429
32	0.363242	0.19056

Tabela A.17: Tempo em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_Bitônica* para $n = 524288$.

Processadores	Computação Local	Comunicação
1	20.3253	0.0
2	4.6266	0.309793
4	3.42472	0.612577
8	1.8895	0.585536
16	0.513702	0.984679
32	0.300239	0.755325

Tabela A.18: Tempo em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_Bitônica* para $n = 1048576$.

Processadores	Computação Local	Comunicação
1	44.6206	0.0
2	9.72991	0.552948
4	7.44638	1.2422
8	4.54211	1.22391
16	1.22486	2.1813
32	0.70064	1.59414

Tabela A.19: Tempo gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_Bitônica* para $n = 2097152$.

Processadores	Computação Local	Comunicação
1	97.6327	0.0
2	17.7458	1.89385
4	13.218	2.49989
8	8.31315	3.94103
16	2.70617	4.91003
32	1.51292	3.4913

Tabela A.20: Tempo gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_Bitônica* para $n = 4191304$.

Processadores	Computação Local	Comunicação
1	212.707	0.0
2	31.01	2.19893
4	39.582	5.027528
8	26.2399	5.0009
16	6.46223	14.3506
32	3.69952	10.0667

Tabela A.21: Tempo gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_Bitônica* para $n = 8388608$.

Apêndice B

Ordenação de Chan & Dehne: Tempos de Execução

As tabelas listadas a seguir apresentam o menor tempo, o maior tempo, os tempos médios, tempos de computação local e comunicação obtidos em nossos experimentos, executado em um Beowulf de 64 processadores Pentium III de 500 MHz, cada um com 256 MB de memória RAM. Todos os nós estavam interconectados através de um switch Fast Ethernet. Cada nó executava o sistema operacional Linux RedHat 7.3 com g++ 2.96 e MPI/LAM 6.5.6.

Processadores	**Menor Tempo**	**Tempo Médio**	**Maior Tempo**
1	0.0307579	0.0308479	0.030951
2	0.0221851	0.0223028	0.022778
4	0.0133631	0.0134215	0.0135629
8	0.00888491	0.00897192	0.00904918
16	0.00694299	0.00699738	0.00705218
32	0.0091989	0.00929437	0.0096271

Tabela B.1: Tempo de execução em segundos do algoritmo *Ordenação de Chan & Dehne* para $n = 32768$.

Processadores	**Menor Tempo**	**Tempo Médio**	**Maior Tempo**
1	0.066618	0.0668539	0.067631
2	0.04685	0.0470813	0.0478389
4	0.02687	0.027067	0.027251
8	0.0162871	0.0167216	0.016783
16	0.0108778	0.0109415	0.010994
32	0.0108659	0.0109492	0.0110059

Tabela B.2: Tempo de execução em segundos do algoritmo *Ordenação de Chan & Dehne* para $n = 65536$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	0.148372	0.14885	0.149606
2	0.099977	0.100221	0.100465
4	0.054599	0.0552371	0.056078
8	0.0311019	0.031257	0.031786
16	0.020318	0.0204259	0.0205369
32	0.0150011	0.0150958	0.0152361

Tabela B.3: Tempo de execução em segundos do algoritmo *Ordenação de Chan & Dehne* para $n = 131072$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	0.317221	0.317764	0.318605
2	0.215007	0.215818	0.21741
4	0.120058	0.121174	0.1225
8	0.0621381	0.0625737	0.0632548
16	0.038096	0.0382721	0.0385962
32	0.024235	0.0243341	0.0245509

Tabela B.4: Tempo de execução em segundos do algoritmo *Ordenação de Chan & Dehne* para $n = 262144$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	0.672115	0.672709	0.673416
2	0.444739	0.446364	0.449131
4	0.240233	0.247925	0.255359
8	0.131842	0.135734	0.13745
16	0.0713291	0.0724344	0.0731111
32	0.0448868	0.045033	0.0457091

Tabela B.5: Tempo de execução em segundos do algoritmo *Ordenação de Chan & Dehne* para $n = 524288$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	1.40445	1.40533	1.40654
2	0.901052	0.91173	0.926725
4	0.501172	0.518537	0.534067
8	0.270946	0.275923	0.281554
16	0.144791	0.148654	0.150582
32	0.08111	0.0815705	0.0824029

Tabela B.6: Tempo de execução em segundos do algoritmo *Ordenação de Chan & Dehne* para $n = 1048576$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	2.9731	2.9742	2.9753
2	1.8883	1.91484	1.93651
4	1.05235	1.05781	1.06571
8	0.555749	0.563695	0.569232
16	0.304441	0.307416	0.309677
32	0.161247	0.16222	0.163268

Tabela B.7: Tempo de execução em segundos do algoritmo *Ordenação de Chan & Dehne* para $n = 2097152$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	6.32377	6.32804	6.32283
2	-	-	-
4	2.15	2.16538	2.18354
8	1.1579	1.16981	1.17517
16	0.600166	0.619487	0.629664
32	0.323804	0.326904	0.329013

Tabela B.8: Tempo de execução em segundos do algoritmo *Ordenação de Chan & Dehne* para $n = 4191304$.

Processadores	**Menor Tempo**	**Tempo Médio**	**Maior Tempo**
1	13.1637	13.1917	13.7997
2	-	-	-
4	-	-	-
8	2.40287	2.42084	2.4397
16	1.264	1.27838	1.29171
32	0.667109	0.675671	0.686665

Tabela B.9: Tempo de execução em segundos do algoritmo *Ordenação de Chan & Dehne* para $n = 8388608$.

Processadores	$n =$ **32768**	$n =$ **65536**	$n =$ **131072**
1	1	1	1
2	1.3831402335	1.4199671632	1.4852675587
4	2.2983943672	2.4699412569	2.6948373466
8	3.4382718526	3.9980564061	4.7622932463
16	3.3189877313	6.1101220125	7.2875613804
32	4.1400331599	6.1086146087	9.8606897283

Tabela B.10: *Speedup* do algoritmo *Ordenação de Chan & Dehne*.

Processadores	$n =$ **262144**	$n =$ **524288**	$n =$ **1048576**
1	1	1	1
2	1.4723702379	1.5070861449	1.5413883496
4	2.6223777378	2.7133568620	2.7101826871
8	5.0782357444	4.9560832216	5.0931962903
16	8.3027584063	9.2871480954	9.4536978487
32	13.0583830920	14.9381342571	17.2284097805

Tabela B.11: *Speedup* do algoritmo *Ordenação de Chan & Dehne*.

Processadores	$n =$ **2097152**	$n =$ **4191304**	$n =$ **8388608**
1	1	1	1
2	1.5532368240	-	-
4	2.8116580482	2.9223692839	-
8	5.2762575506	5.4094596558	5.4492242362
16	9.6748380045	10.2149681914	10.3190757052
32	18.3366214550	19.3574872134	19.5238511050

Tabela B.12: *Speedup* do algoritmo *Ordenação de Chan & Dehne*.

Processadores	Computação Local	Comunicação
1	0.0308479	0.0
2	0.0177303	0.00456621
4	0.004316	0.00914166
8	0.00493205	0.00400178
16	0.00270752	0.00423814
32	0.00161862	0.00771575

Tabela B.13: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação de Chan & Dehne* para $n = 32768$.

Processadores	Computação Local	Comunicação
1	0.0668539	0.0
2	0.0381845	0.00886091
4	0.0195049	0.00752968
8	0.0103255	0.00644742
16	0.00542659	0.00546602
32	0.00298776	0.00800172

Tabela B.14: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação de Chan & Dehne* para $n = 65536$.

Processadores	Computação Local	Comunicação
1	0.14885	0.0
2	0.0812065	0.0189782
4	0.0409999	0.0142448
8	0.021495	0.0097503
16	0.0117079	0.00870253
32	0.00638743	0.00872643

Tabela B.15: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação de Chan & Dehne* para $n = 131072$.

Processadores	Computação Local	Comunicação
1	0.317764	0.0
2	0.168569	0.0474384
4	0.0867528	0.0340581
8	0.045234	0.017138
16	0.0237794	0.0144343
32	0.0127711	0.0115369

Tabela B.16: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação de Chan & Dehne* para $n = 262144$.

Processadores	Computação Local	Comunicação
1	0.672709	0.0
2	0.356578	0.089944
4	0.190674	0.0522445
8	0.0958386	0.0406718
16	0.049314	0.0229376
32	0.0268458	0.0181969

Tabela B.17: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação de Chan & Dehne* para $n = 524288$.

Processadores	Computação Local	Comunicação
1	1.40533	0.0
2	0.749801	0.157393
4	0.400325	0.113463
8	0.206509	0.0702411
16	0.105204	0.0592093
32	0.0541196	0.0279467

Tabela B.18: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação de Chan & Dehne* para $n = 1048576$.

Processadores	Computação Local	Comunicação
1	2.9742	0.0
2	1.59069	0.321015
4	0.856645	0.203488
8	0.443953	0.121054
16	0.229764	0.0862117
32	0.113906	0.0535296

Tabela B.19: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação de Chan & Dehne* para $n = 2097152$.

Processadores	Computação Local	Comunicação
1	6.32804	0.0
2	-	-
4	1.77175	0.391405
8	0.938168	0.235661
16	0.478448	0.141283
32	0.245177	0.0807583

Tabela B.20: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação de Chan & Dehne* para $n = 4194304$.

Processadores	**Computação Local**	**Comunicação**
1	13.1917	0.0
2	-	-
4	-	-
8	1.96483	0.478758
16	1.01083	0.273049
32	0.520121	0.15755

Tabela B.21: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação de Chan & Dehne* para $n = 8388608$.

Apêndice C

Ordenação por Divisão: Tempos de Execução

As tabelas listadas a seguir apresentam o menor tempo, o maior tempo, os tempos médios, tempos de computação local e comunicação obtidos em nossos experimentos, executado em um Beowulf de 64 processadores Pentium III de 500 MHz, cada um com 256 MB de memória RAM. Todos os nós estavam interconectados através de um switch Fast Ethernet. Cada nó executava o sistema operacional Linux RedHat 7.3 com g++ 2.96 e MPI/LAM 6.5.6.

Processadores	**Menor Tempo**	**Tempo Médio**	**Maior Tempo**
1	0.0307579	0.0308479	0.030951
2	0.026876	0.0278436	0.0282509
4	0.0189619	0.019332	0.0195889
8	0.016083	0.0164445	0.0166268
16	0.0181	0.0190205	0.0198929
32	0.0282171	0.0302203	0.031548

Tabela C.1: Tempo de execução em segundos do algoritmo *Ordenação_por_Divisão* para $n = 32768$.

Processadores	**Menor Tempo**	**Tempo Médio**	**Maior Tempo**
1	0.066618	0.0668539	0.067631
2	0.0553842	0.0566134	0.0578029
4	0.0362291	0.037053	0.0373869
8	0.027143	0.0279533	0.0283549
16	0.0259628	0.0269783	0.0277131
32	0.034384	0.0379908	0.039258

Tabela C.2: Tempo de execução em segundos do algoritmo *Ordenação_por_Divisão* para $n = 65536$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	0.148372	0.14885	0.149606
2	0.114546	0.11731	0.118338
4	0.0728528	0.0739498	0.0747252
8	0.049181	0.0501368	0.0506601
16	0.0422981	0.0436701	0.044553
32	0.0476849	0.0497292	0.0515618

Tabela C.3: Tempo de execução em segundos do algoritmo *Ordenação_por_Divisão* para $n = 131072$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	0.317221	0.317764	0.318605
2	0.241602	0.246163	0.249284
4	0.153172	0.155541	0.156956
8	0.095464	0.098097	0.0992792
16	0.0741749	0.0766829	0.0781889
32	0.0725369	0.0746892	0.076576

Tabela C.4: Tempo de execução em segundos do algoritmo *Ordenação_por_Divisão* para $n = 262144$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	0.672115	0.672709	0.673416
2	0.500249	0.518632	0.531913
4	0.307142	0.312757	0.316201
8	0.192868	0.196923	0.199547
16	0.137046	0.140887	0.144535
32	0.120165	0.125848	0.129668

Tabela C.5: Tempo de execução em segundos do algoritmo *Ordenação_por_Divisão* para $n = 524288$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	1.40445	1.40533	1.40654
2	1.00656	1.03466	1.04797
4	0.643196	0.65434	0.662682
8	0.392077	0.403171	0.40982
16	0.268262	0.286142	0.305787
32	0.220592	0.231241	0.241716

Tabela C.6: Tempo de execução em segundos do algoritmo *Ordenação_por_Divisão* para $n = 1048576$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	2.9731	2.9742	2.9753
2	2.29547	2.37804	2.41896
4	1.36699	1.39807	1.41817
8	0.808469	0.821789	0.833144
16	0.538741	0.551493	0.561731
32	0.452259	0.515092	0.606596

Tabela C.7: Tempo de execução em segundos do algoritmo *Ordenação_por_Divisão* para $n = 2097152$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	6.32377	6.32804	6.32283
2	4.69465	4.96405	5.06729
4	2.70324	2.75589	2.78353
8	1.63814	1.67402	1.69192
16	1.12067	1.13872	1.15619
32	0.820716	0.850736	0.884822

Tabela C.8: Tempo de execução em segundos do algoritmo *Ordenação_por_Divisão* para $n = 4194304$.

Processadores	Menor Tempo	Tempo Médio	Maior Tempo
1	13.1637	13.1917	13.7997
2	9.06123	9.34409	9.44935
4	5.80575	5.94526	6.01488
8	3.37021	3.41138	3.43455
16	2.21737	2.24711	2.2658
32	1.6165	1.71554	1.77201

Tabela C.9: Tempo de execução em segundos do algoritmo *Ordenação_por_Divisão* para $n = 8388608$.

Processadores	$n = \mathbf{32768}$	$n = \mathbf{65536}$	$n = \mathbf{131072}$
1	1	1	1
2	1.1078991222	1.1808847375	1.2688602847
4	1.5956910821	1.8042776563	2.0128519617
8	1.8758794733	2.3916281798	2.9688771521
16	1.6218238216	2.4780619979	3.4085106285
32	1.0207674973	1.7597392000	2.9932112320

Tabela C.10: *Speedup* do algoritmo *Ordenação_por_Divisão*, $32768 \leq n \leq 131072$.

Processadores	$n = \mathbf{262144}$	$n = \mathbf{524288}$	$n = \mathbf{1048576}$
1	1	1	1
2	1.2908682458	1.2970834811	1.3582529526
4	2.0429597340	2.1508998999	2.1477060855
8	3.2392835662	3.4161017250	3.4856921752
16	4.1438704065	4.7748124383	4.9113027797
32	4.2544839146	5.3454087470	6.0773392261

Tabela C.11: *Speedup* do algoritmo *Ordenação_por_Divisão*, $262144 \leq n \leq 1048576$.

Processadores	$n = \mathbf{2097152}$	$n = \mathbf{4194304}$	$n = \mathbf{8388608}$
1	1	1	1
2	1.2506938487	1.2747736223	1.4117693643
4	2.1273612909	2.2961874385	2.2188600666
8	3.6191771853	3.7801459958	3.8669687926
16	5.3929968286	5.5571518898	5.8705181321
32	5.7741141388	7.4383122378	7.6895321589

Tabela C.12: *Speedup* do algoritmo *Ordenação_por_Divisão*, $2097152 \leq n \leq 8388608$.

Processadores	Computação Local	Comunicação
1	0.0308479	0.0
2	0.0224935	0.00608037
4	0.0144594	0.00487255
8	0.0112193	0.00522516
16	0.0113208	0.00769978
32	0.012791	0.0174293

Tabela C.13: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_por_Divisão* para $n = 32768$.

Processadores	Computação Local	Comunicação
1	0.0668539	0.0
2	0.0457064	0.010907
4	0.0279181	0.00913491
8	0.0200059	0.00794746
16	0.0176794	0.00929892
32	0.0194272	0.0185636

Tabela C.14: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_por_Divisão* para $n = 65536$.

Processadores	Computação Local	Comunicação
1	0.14885	0.0
2	0.0902579	0.0270517
4	0.0568576	0.0170922
8	0.0388336	0.0113032
16	0.0309395	0.0127306
32	0.0301524	0.0195768

Tabela C.15: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_por_Divisão* para $n = 131072$.

Processadores	Computação Local	Comunicação
1	0.317764	0.0
2	0.195463	0.05070077
4	0.119831	0.03571
8	0.0769269	0.0211701
16	0.0593194	0.0173635
32	0.0516286	0.0230606

Tabela C.16: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_por_Divisão* para $n = 262144$.

Processadores	Computação Local	Comunicação
1	0.672709	0.0
2	0.434108	0.0845234
4	0.247876	0.0648801
8	0.15383	0.0430922
16	0.111339	0.0295476
32	0.0923865	0.0334619

Tabela C.17: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_por_Divisão* para $n = 524288$.

Processadores	Computação Local	Comunicação
1	1.40533	0.0
2	0.864539	0.170122
4	0.53408	0.120261
8	0.32491	0.0782609
16	0.223485	0.0626573
32	0.178732	0.0525098

Tabela C.18: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_por_Divisão* para $n = 1048576$.

Processadores	Computação Local	Comunicação
1	2.9742	0.0
2	2.07073	0.307316
4	1.1743	0.22377
8	0.681878	0.139911
16	0.456027	0.0954659
32	0.343204	0.171887

Tabela C.19: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_por_Divisão* para $n = 2097152$.

Processadores	Computação Local	Comunicação
1	6.32804	0.0
2	4.38575	0.578298
4	2.33336	0.42253
8	1.42899	0.245029
16	0.949899	0.188824
32	0.705907	0.144829

Tabela C.20: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_por_Divisão* para $n = 4194304$.

Processadores	**Computação Local**	**Comunicação**
1	13.1917	0.0
2	8.11962	1.22448
4	5.09348	0.851782
8	2.91833	0.493054
16	1.94012	0.306994
32	1.46396	0.251577

Tabela C.21: Tempo médio em segundos gasto com computação local e comunicação entre processadores pelo algoritmo *Ordenação_por_Divisão* para $n = 8388608$.

Apêndice D

Gráficos com Tempo de Computação Local e Comunicação

Os gráficos apresentados a seguir representam os tempos médios obtidos com computação local e comunicação entre processadores em nossos experimentos, executado em um Beowulf de 64 processadores Pentium III de 500 MHz, cada um com 256 MB de memória RAM. Todos os nós estavam interconectados através de um switch Fast Ethernet. Cada nó executava o sistema operacional Linux RedHat 7.3 com g++ 2.96 e MPI/LAM 6.5.6.

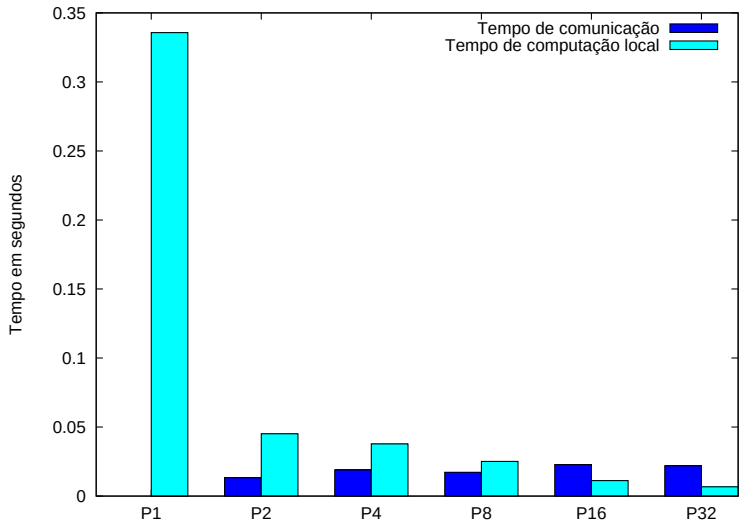


Figura D.1: Tempos de execução do algoritmo *Ordenação_Bitônica*, para $n = 32768$.

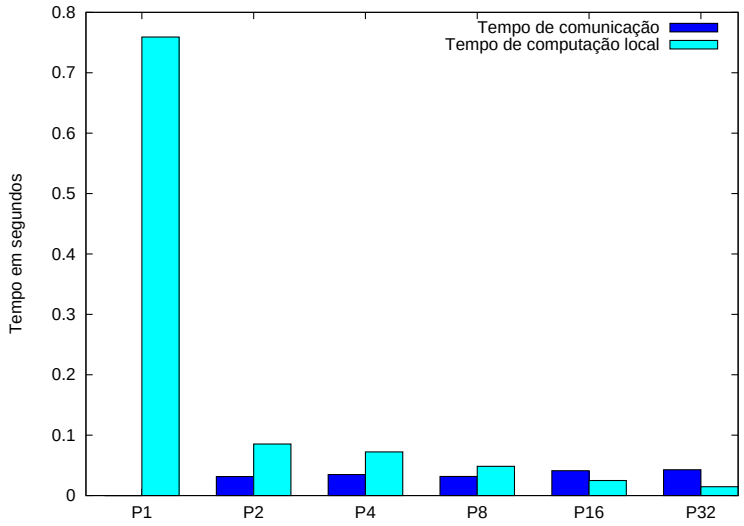


Figura D.2: Tempos de execução do algoritmo *Ordenação_Bitônica*, para $n = 65536$.

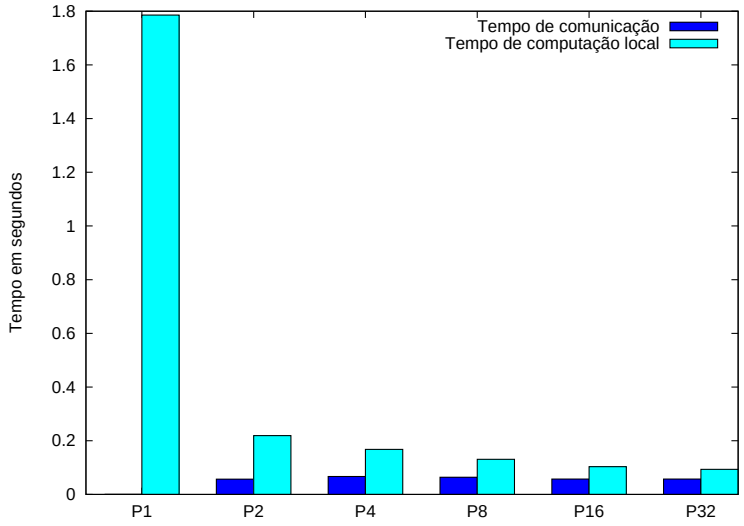


Figura D.3: Tempos de execução do algoritmo *Ordenação_Bitônica*, para $n = 131072$.

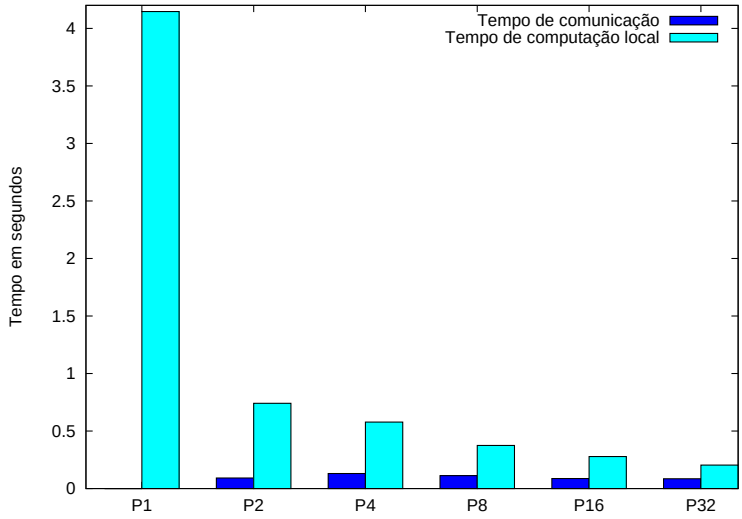


Figura D.4: Tempos de execução do algoritmo *Ordenação_Bitônica*, para $n = 262144$.

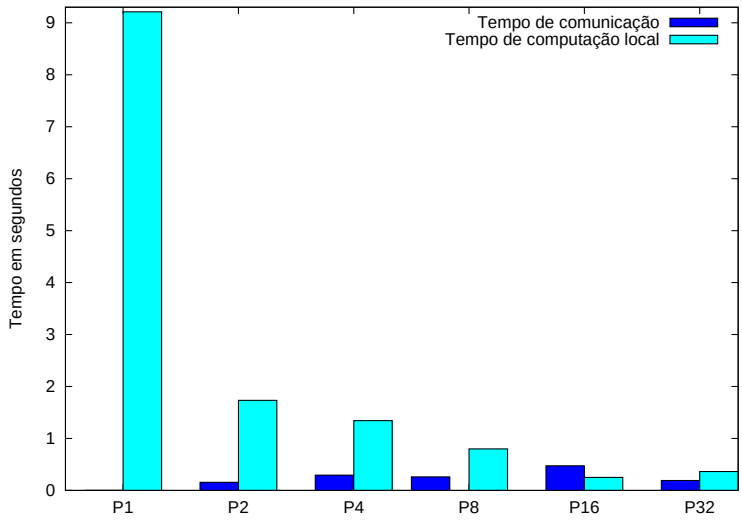


Figura D.5: Tempos de execução do algoritmo *Ordenação_Bitônica*, para $n = 524288$.

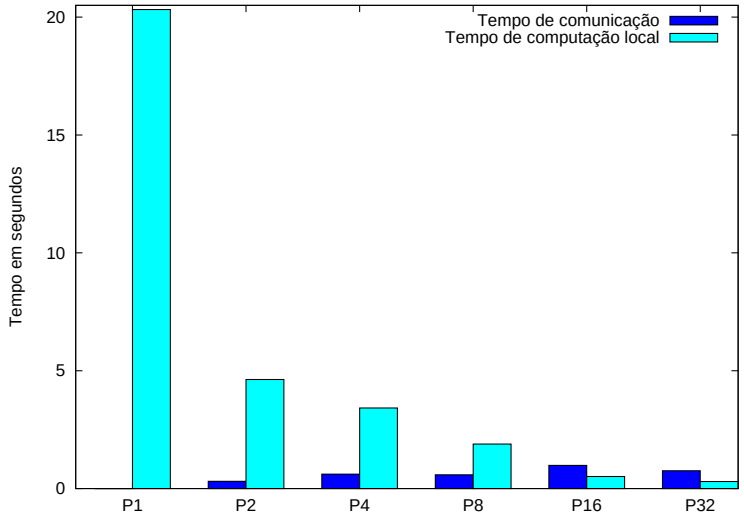


Figura D.6: Tempos de execução do algoritmo *Ordenação_Bitônica*, para $n = 1048576$.

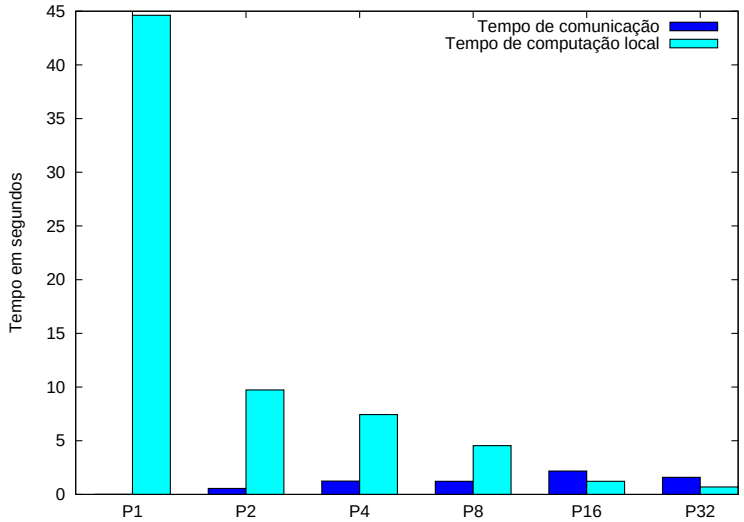


Figura D.7: Tempos de execução do algoritmo *Ordenação_Bitônica*, para $n = 2097152$.

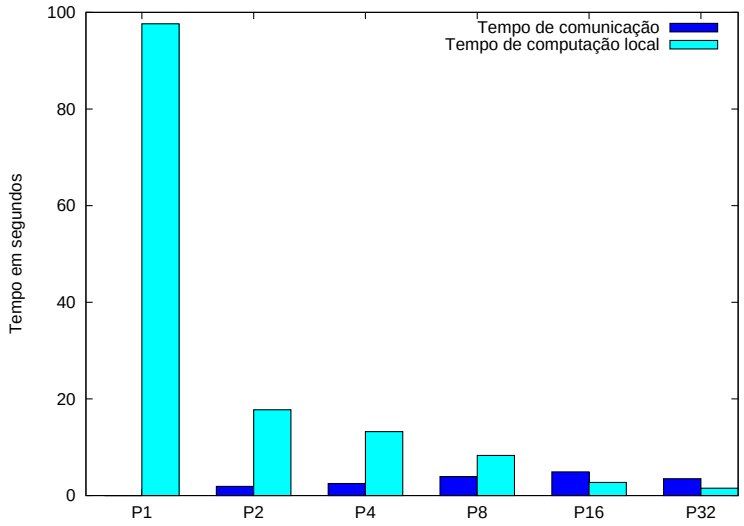


Figura D.8: Tempos de execução do algoritmo *Ordenação_Bitônica*, para $n = 4194304$.

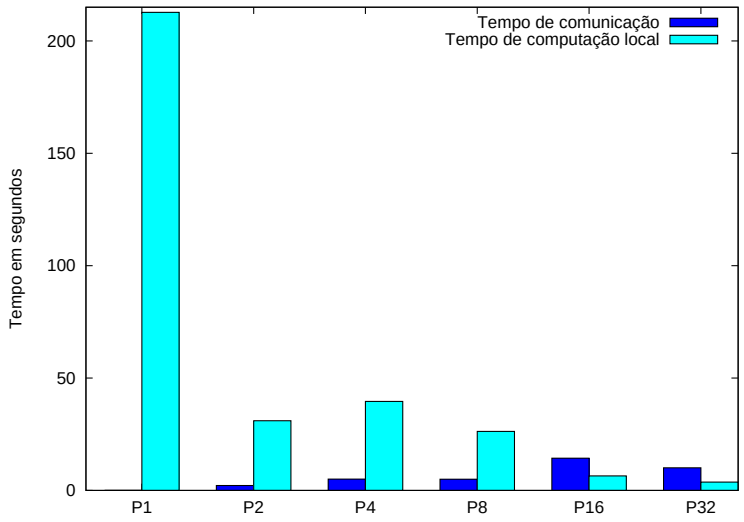


Figura D.9: Tempos de execução do algoritmo *Ordenação_Bitônica*, para $n = 8388608$.

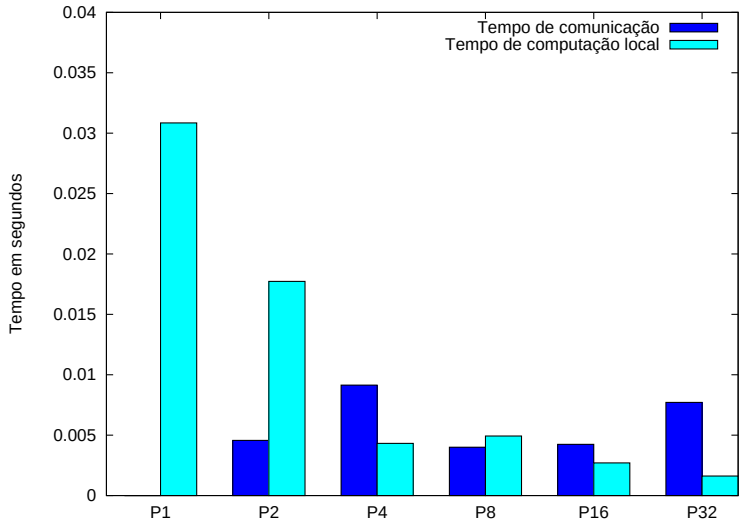


Figura D.10: Tempos de execução do algoritmo *Ordenação_CD*, para $n = 32768$.

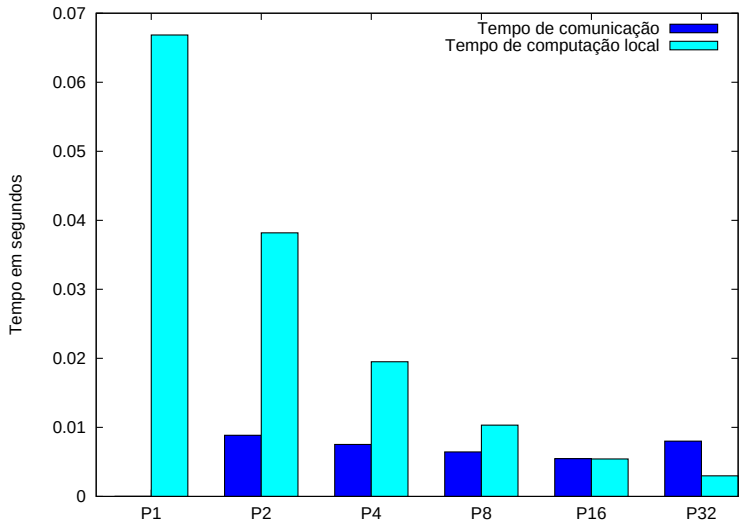


Figura D.11: Tempos de execução do algoritmo *Ordenação_CD*, para $n = 65536$.

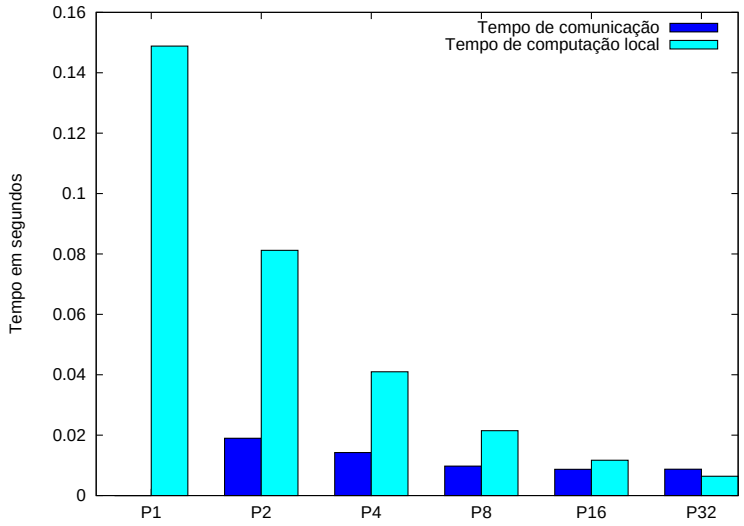


Figura D.12: Tempos de execução do algoritmo *Ordenação_CD*, para $n = 131072$.

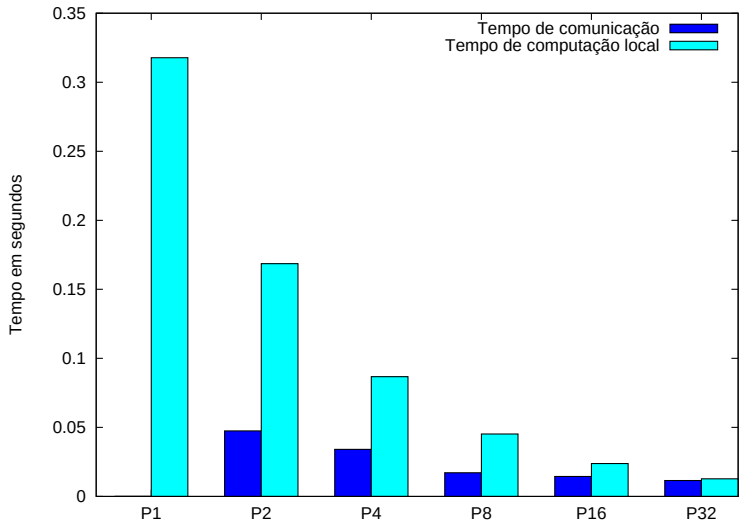


Figura D.13: Tempos de execução do algoritmo *Ordenação_CD*, para $n = 262144$.

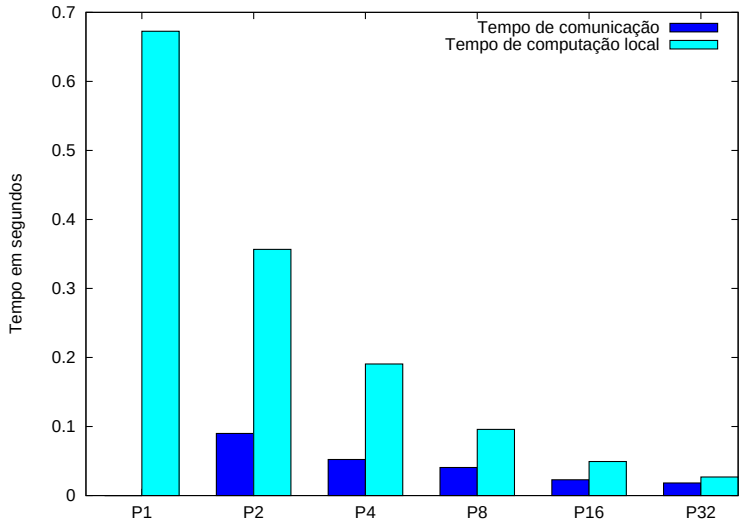


Figura D.14: Tempos de execução do algoritmo *Ordenação_CD*, para $n = 524288$.

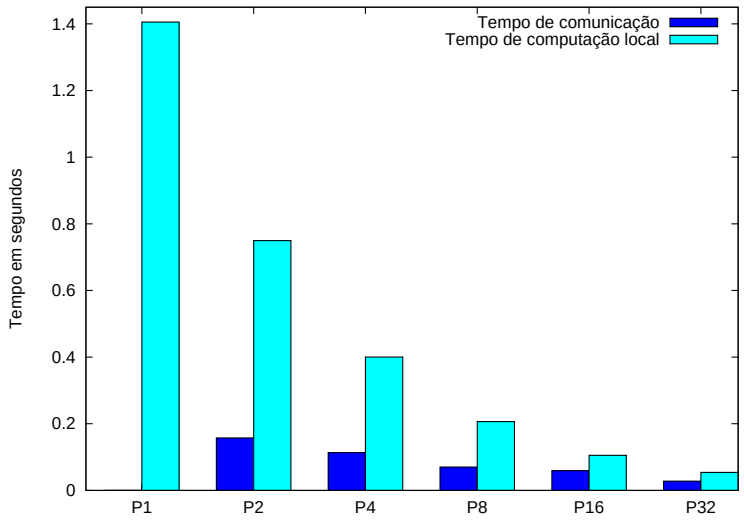


Figura D.15: Tempos de execução do algoritmo *Ordenação_CD*, para $n = 1048576$.

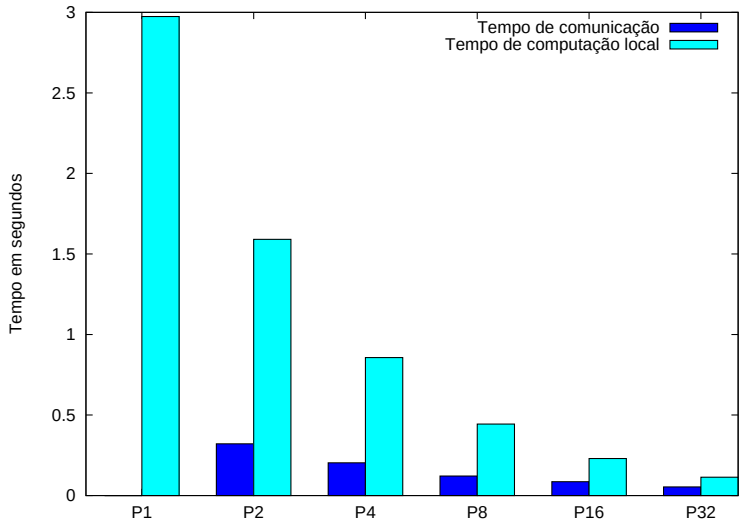


Figura D.16: Tempos de execução do algoritmo *Ordenação_CD*, para $n = 2097152$.

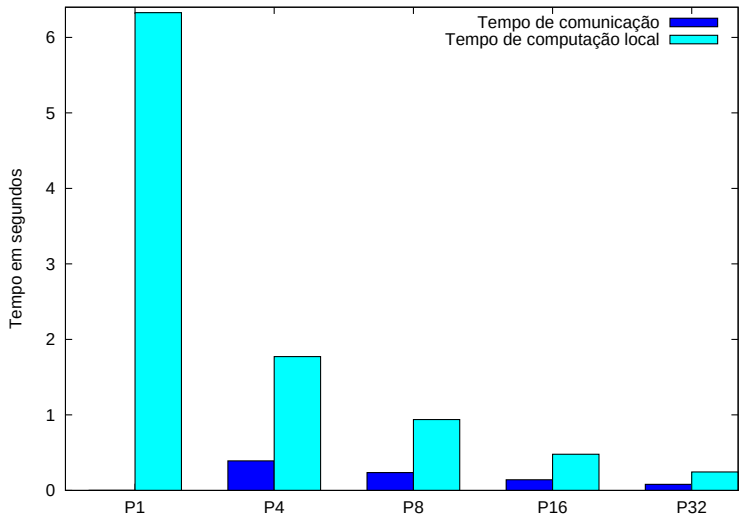


Figura D.17: Tempos de execução do algoritmo *Ordenação_CD*, para $n = 4194304$.

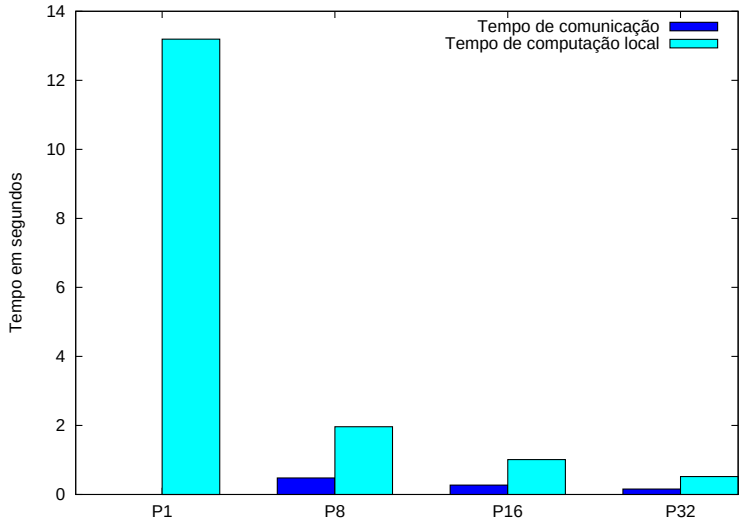


Figura D.18: Tempos de execução do algoritmo *Ordenação_CD*, para $n = 8388608$.

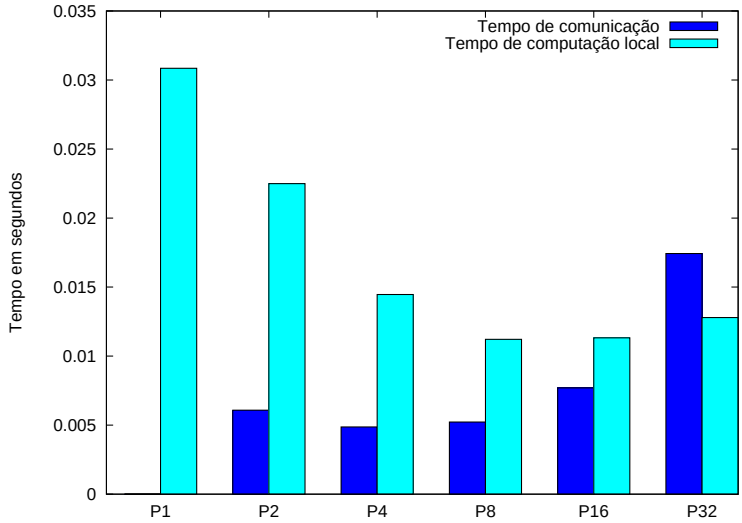


Figura D.19: Tempos de execução do algoritmo *Ordenação_por_Divisão*, para $n = 32768$.

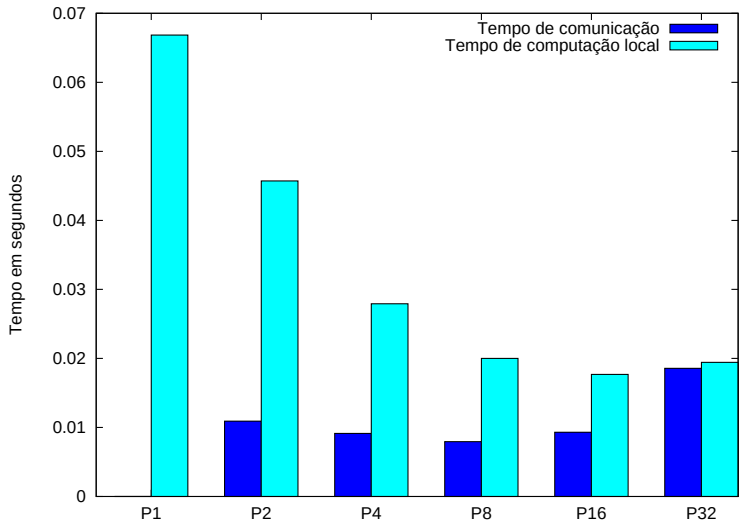


Figura D.20: Tempos de execução do algoritmo *Ordenação_por_Divisão*, para $n = 65536$.

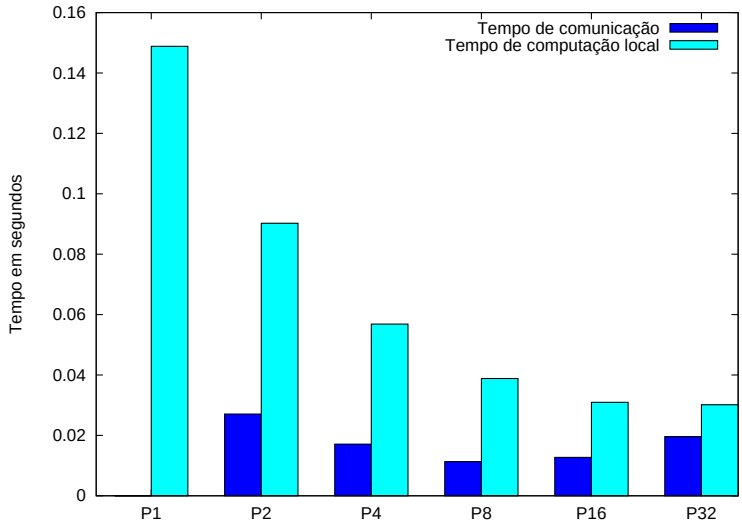


Figura D.21: Tempos de execução do algoritmo *Ordenação_por_Divisão*, para $n = 131072$.

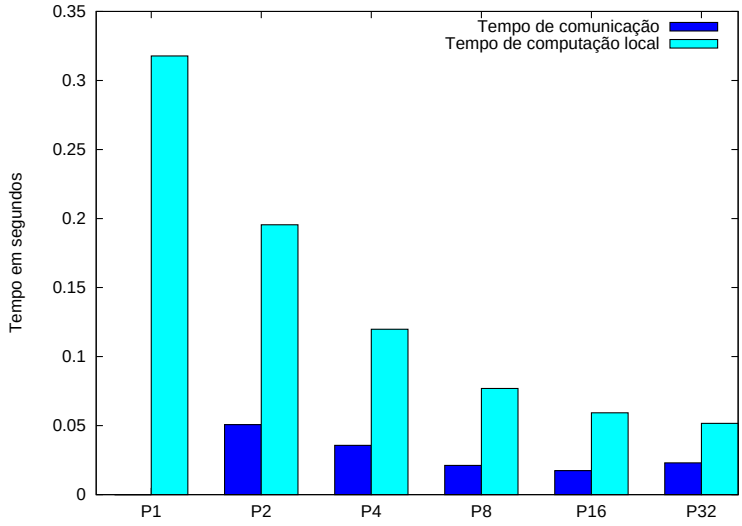


Figura D.22: Tempos de execução do algoritmo *Ordenação_por_Divisão*, para $n = 262144$.

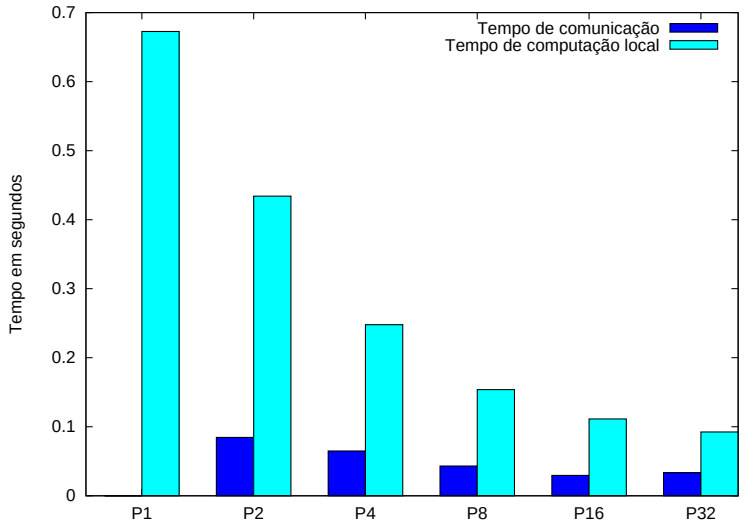


Figura D.23: Tempos de execução do algoritmo *Ordenação_por_Divisão*, para $n = 524288$.

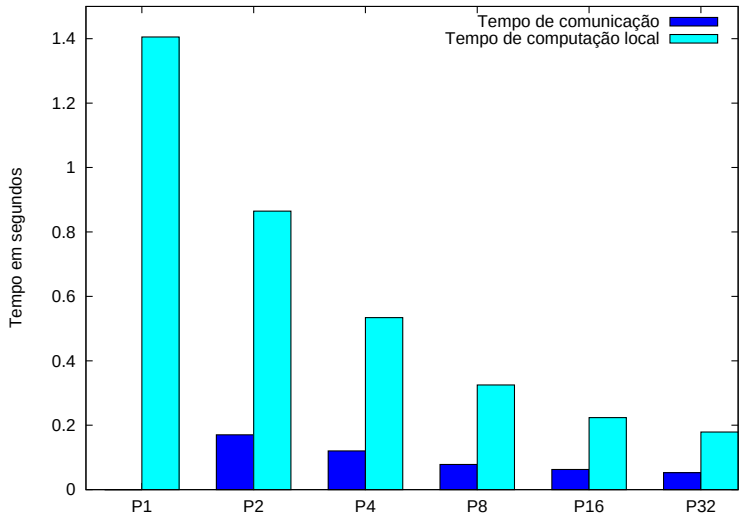


Figura D.24: Tempos de execução do algoritmo *Ordenação_por_Divisão*, para $n = 1048576$.

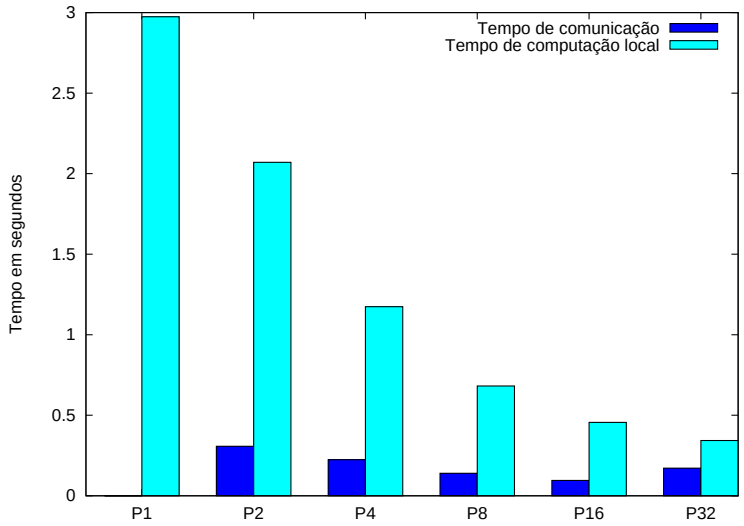


Figura D.25: Tempos de execução do algoritmo *Ordenação_por_Divisão*, para $n = 2097152$.

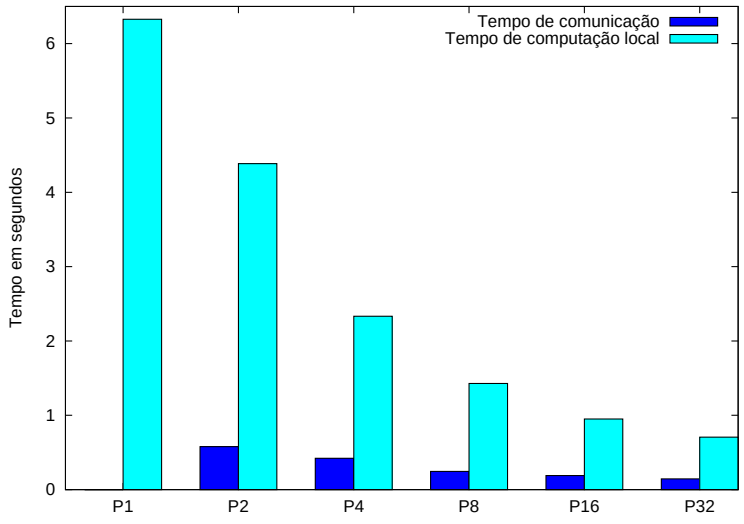


Figura D.26: Tempos de execução do algoritmo *Ordenação_por_Divisão*, para $n = 4194304$.

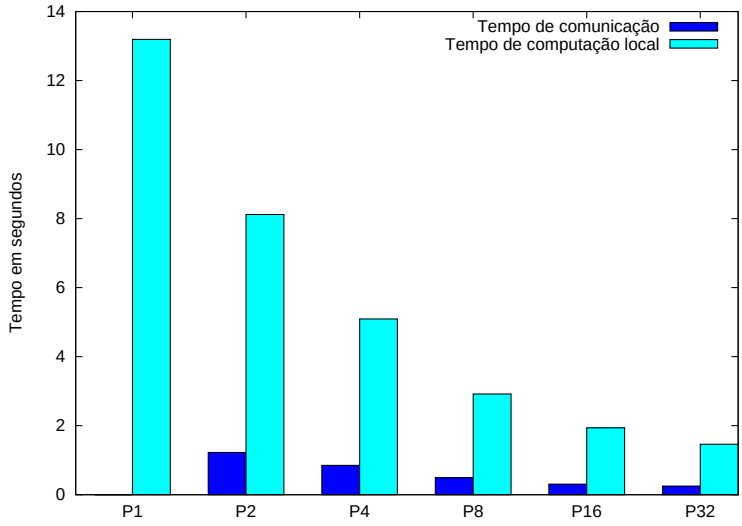


Figura D.27: Tempos de execução do algoritmo *Ordenação_por_Divisão*, para $n = 8388608$.

Referências Bibliográficas

[1] S. G. Akl. *Parallel Sorting Algorithms*. Academic Press, 1985.

[2] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, e M. Zagha. A comparision of sorting algorithms for the conection machine CM-2. In *ACM Symposium on Parallel Algorithms and Architectures*, páginas 3–16. 1991.

[3] E. N. Cáceres, H. Mongelli, e S. W. Song. *Algoritmos Paralelos Usando CGM/PVM: Uma Introdução*, páginas 219–278. XXI Congresso da Sociedade Brasileira de Computação, Julho 2001.

[4] A. Chan e F. Dehne. A note on coarsed grained parallel integer sorting. In *Parallel Processing Letter*, volume 9, páginas 533–538. 1999.

[5] T. H. Cormen, A. Fabri, e R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1996.

[6] D. A. Bader D. R. Helman, J. Jájá. A new deterministic parallel sorting algorithms with an experimental evaluation. *ACM Journal of Experimental Algorithms*, 3(4):1–24, 1998.

[7] D.Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, e T. von Eicken. LogP: Torwards a realistic model for parallel computation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 4, páginas 1–12. 1993.

[8] A. A. de Castro Júnior. *Implementação e Avaliação de Algoritmos BSP/CGM para o Fecho Transitivo e Problemas Relacionados*. Tese de Mestrado, Universidade Federal de Mato Grosso do Sul, Março 2003.

[9] F. Dehne, A. Fabri, e A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *Proceedings of the ACM 9th Annual Computational Geometry*, páginas 298–307. 1993.

[10] F. Dehne, A. Ferreira, E. N. Cáceres, S. W. Song, e A. Roncato. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. *Algorithmica*, 33(2):183–200, 2002.

[11] A. C. Dusseau, D. E. Culler, K. E. Schauser, e R. P. Martin. Fast parallel sorting under LogP: Experience with the CM-5. In *IEEE Transactions on Parallel and Distributed Systems*, volume 7, páginas 791–805. 1996.

[12] A. Ferreira. Discrete computing with pc clusters: an algorithmic approach. Tutorial, International School on Advanced Algorithmic Techniques for Parallel Computation with Applications, Setembro 1999.

[13] M. T. Goodrich. Communication-efficient parallel sorting. In *Proc. of the 28th annual ACM Symposium of Theory of Computing*. 1996.

[14] S. Götz. *Communication-efficient parallel algorithms for minimum spanning tree computation*. Tese de Doutoramento, University of Paderborn, 1998.

[15] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Publishing Company, 1973.

[16] V. Kumar, A. Gupta A. Grama, e G. Karypis. *Introdution to Parallel Computing: Design and Analysis of Algorithms*. Benjamim/Cummings Publishing Company, 1994.

[17] F. T. Leighton. *Introduction to Parallel Algorithms and Architeture: Arrays Trees Hypercubes*. Morgan Kaufmann Publishers, 1992.

[18] B.M. Maggs, L. R. Matheson, e R. E. Tarjan. Models of parallel computation: A survey and synthesis. In *Proc.28th Hawaii Internal of Conf. os System Sciences*, volume 2, páginas 61–70. 1995.

[19] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Publishing Company, 1989.

[20] H. Mongelli. *Algoritmos CGM para busca uni e bidimensional de padrões com e sem escala*. Tese de Doutoramento, Universidade de São Paulo, 2000.

[21] C. Nasu. *Algoritmos BSP/CGM para Computação de Circuitos de Euler em Grafos*. Tese de Mestrado, Universidade Federal de Mato Grosso do Sul, Setembro 2002.

[22] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[23] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, 1993.

[24] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.

[25] B. Wilkinson e M. Allen. *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computer*. Prentice Hall, 1999.