Quicksort, Samplesort e Splitsort

Diego S. Cintra[1], Jainor Souza[1]

[1]*Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)*
Caixa Postal 549 – 79.070-900 – Campo Grande – MS – Brasil
diego_2337@hotmail.com, jainorsouza@gmail.com

10 de junho de 2015

Abstract. *It was requested to us the implementation of the parallel versions of the quicksort, samplesort and splitsort, having as components for execution classes to support the understanding of these algorithms and the use of MPI. Executing the sorting programs (written in C) should serve as comparison between not only the speedup among themselves but also to the sequential version of the quicksort. For this, it's essential to understand the foundations of parallel programming and main communication procedures. These requests aim to broaden the importance of the explicit parallelization in computing applications.*

Resumo. *Foi-nos requisitado a implementação das versões paralelas dos algoritmos quicksort, samplesort e splitsort, tendo como componentes para execução aulas ministradas para apoiar o entendimento desses algoritmos e a utilização do MPI. A execução dos algoritmos de ordenação (escritos em C) deve servir como comparação entre não somente o speedup deles como também da versão sequencial do quicksort. Para isso, é essencial entender os fundamentos da programação paralela e os principais procedimentos de comunicação. Essas requisições visam ampliar a importância da paralelização explícita em aplicações de computação.*

1 Introdução

Algoritmos de ordenação são um dos fundamentos da computação, devido a quase todas as operações computacionais que possuímos atualmente utilizarem, pelo menos parcialmente, alguma espécie de ordenação. Dessa maneira, o ramo da programação paralela também tenta, na medida do possível, explorar as possíveis soluções de paralelismo nessa área, adotando não só o gasto computacional envolvido na execução dos algoritmos de ordenação como também o tempo de comunicação necessário. Portanto, nesse documento, abordaremos três algoritmos de ordenação paralelos, explicando seu funcionamento e apresentando seus resultados - o Samplesort, Splitsort e Quicksort.

2 Desenvolvimento do projeto

2.1 Splitsort

Antes de adentrarmos à formulação do Splitsort, é importante entender de onde ele veio. O algoritmo Bucketsort busca ordenar os elementos de um vetor atribuindo um conjunto deles a diferentes "baldes" (*buckets*), que são ordenados serialmente e depois atribuídos ao vetor original. É fácil ver que esse algoritmo é facilmente paralelizável, entretanto o grande gargalo dele é a complexidade: se os elementos são uniformemente distribuídos, o algoritmo apresenta

um tempo de execução excelente; do contrário o tempo não é dos melhores. Logicamente, nem sempre é tão simples de se obter uma entrada uniformemente distribuída.

O algoritmo Splitsort é uma melhoria em relação ao Bucketsort, na tentativa de prover uma distribuição mais uniforme dos dados quando esse cenário não é possível com a entrada fornecida. Para sua execução em paralelo, a seguinte sequência de passos foi realizada:

1. Em um primeiro momento, todos os dados estão contidos no processador P0; daqui, calcula-se o tamanho de dados que cada processador irá receber e realiza-se um `scatter` desses;

2. Após isso, cada processador realiza o cálculo de seu quartil (que, neste trabalho, foi generalizado para uma quantidade arbitrária de processadores, não somente 4) através da ordenação de seu subconjunto - utilizamos a função `qsort` da biblioteca padrão de C para tal ordenação;

3. O processador P0 recebe o quartil de cada um dos outros contidos em seu domínio de comunicação e repete a mesma operação: ordena-os e acha seus quartis;

4. Realizando novamente a distribuição desse conjunto de quartis, cada processador agora será capaz de realizar uma comunicação personalizada (`all-to-all`), receber uma distribuição correspondente e ordenar corretamente ela, assim ordenando o conjunto todo.

Além do que foi explicitado acima, é importante notar alguns detalhes especiais para a execução desse algoritmo neste trabalho. Todos os testes foram feitos com números de processadores e quantidade de números múltiplos entre si; isso quer dizer que, no evento de executá-lo sem seguir essa restrição, o resultado pode ser incorreto e o comportamento do programa inesperado. Assim como foi requisitado, a função de comparação implementada para a execução do `qsort` divide o dado em três partes, de acordo com a quantidade de bits extraída para cada uma delas. Uma das seções do algoritmo (especificamente na determinação da quantidade de elementos pertencentes a cada *bucket*) foi levemente alterada em relação ao programa fornecido em sala, visto que o funcionamento dele não foi completamente correto para diversas instâncias; mesmo assim, a complexidade original do algoritmo ainda foi mantida.

2.2 Samplesort

Esse algoritmo é equivalente ao supracitado, entretanto, ao invés de selecionarmos os separadores com a ideia de quartis, aqui eles simplesmente são selecionados de acordo com sua espacialidade - "igualmente espaçados". A sequência de passos é basicamente a mesma:

1. Como visto anteriormente, todos os dados estão contidos no processador P0; calcula-se aqui o tamanho aproximado de dados que cada processador irá receber e realiza-se um `scatter` desses;

2. Após isso, cada processador realiza a obtenção de seus separadores através da ordenação de seu subconjunto, tal como havia sido feita no Splitsort;

3. O processador P0 recebe todos os separadores contidos em seu domínio de comunicação e repete a mesma operação: ordena-os e acha os separadores dos separadores;

4. Realizando novamente a distribuição desse conjunto de *splitters*, cada processador agora será capaz de realizar uma comunicação personalizada (`all-to-all`)), receber uma distribuição correspondente e ordenar corretamente ela, assim ordenando o conjunto todo.

Com isso, podemos verificar a complexidade de ambos os algoritmos, que é praticamente idêntica. Em um primeiro momento, cada processador dispende $\theta((n/p)log(n/p))$ para ordenar sua porção local; depois, uma operação de *gather* é realizada para unir os separadores (ou quartis), gastando $\theta(p^2)$. Ordenar esses elementos em P0 irá gastar $\theta(p^2logp)$, e o *one-to-all broadcast* que esse processo realizará consumirá $\theta(p^2)$. $\theta(plog(n/p))$ é o tempo necessário

para a partição local dos buckets, aonde os separadores serão inseridos no bloco. Por fim, a comunicação e reorganização entre os processos e seus elementos consumirá $O(n/p)$. Abaixo tem-se a fórmula da complexidade desse algoritmo:

$$T_p = \theta((n/p)log(n/p)) + \theta(p^2) + \theta(p^2 logp) + \theta(p^2) + \theta(plog(n/p))$$

2.3 Quicksort

Talvez o mais famoso de todos os algoritmos de ordenação, o Quicksort usa a técnica da divisão e conquista para ordenar elementos da maneira que se queira. Inventado por C. A. R. Hoare em 1962, o Quicksort pode ser tão lento quanto os algoritmos elementares ($O(n^2)$), mas, em geral, ele é rápido. Seu caso médio possui tempo $O(n \lg n)$, mas, no pior caso, seu tempo é proporcionalmente quadrático ao tamanho da entrada.

O principal problema do Quicksort é o problema da separação. Ele consiste em definir um pivô, de tal forma que todos os elementos menores que ele se encontrem em seu lado esquerdo e todos os elementos maiores que ele se encontrem em seu lado direito. Assim, definimos um pivô da seguinte forma:

$$v[i..k-1] <= v[k] < v[k+1..n], \forall i < k, k < n$$

Note que, caso acima, o elemento k foi escolhido como pivô e a sequência foi arranjada de forma a manter os elementos menores que $v[k]$ na parte esquerda e os maiores na parte direita. A partir desta divisão, o algoritmo executa uma recursão e as partes esquerda e direita são tratadas como uma sequência independente e então um o algoritmo da separação é executado novamente.

2.3.1 Formulação Sequencial

O algoritmo Quicksort, em uma sua formulação sequencial, pode ser assim:

Algoritmo 1 Quicksort

```
1: if Ainda há sequência para percorrer then
2:     Defina o pivô para a sequência ou subsequência;
3:     Execute recursivamente para a parte esquerda;
4:     Execute recursivamente para a parte direita;
5: end if
```

2.3.2 Formulação Paralela

Para este trabalho, o paradigma usado foi o de troca de mensagens. O objetivo almejado é um algoritmo mais eficiente que o sequencial. No entanto, a ideia da formulação paralela possui a necessidade de definirmos conjuntos de processos. Isto ocorre pois a entrada não está apenas em um processador, o que torna a definição das partes esquerda e direita um conceito global entre todos os processadores que estão executando o algoritmo.

A formulação paralela do Quicksort usada como modelo para o desenvolvido do contido e analisado neste trabalho segue a linha de pensamento acima e está descrita na figura e no algoritmo abaixo:

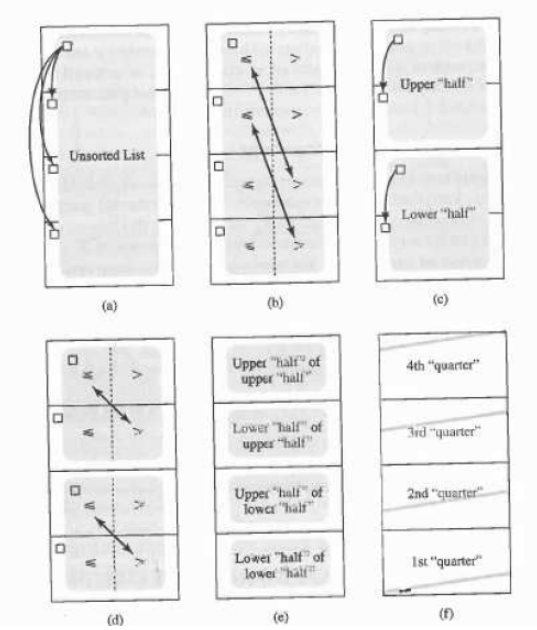

Figura 1: Funcionamento da formulação paralela do Quicksort

Algoritmo 2 ParallelQuicksort

1: Divida a entrada em p partes iguais, uma para cada processo;
2: Ordene a subsequência de cada processo localmente;
3: Ordene globalmente:
4: Escolha um pivô para cada conjunto de processos;
5: Localmente para cada processo, divida sua sequência em duas partes, maior e menor que o pivô;
6: Divida o conjunto de processos em duas partes e troque informações entre processos, dois a dois, de grupos diferentes, de forma que um conjunto de processos fique com toda a parte esquerda em relação ao pivô e outro conjunto com toda a parte direita;
7: Junte as duas sequências de cada processo em uma sequência ordenada;
8: Repita a ordenação global recursivamente para cada conjunto de processos, enquanto houver mais de um processo por conjunto.

A figura acima foi obtida a partir do capítulo 14 do livro "Parallel Programming in C with MPI and OpenMP"(Michael J. Quinn), e ilustra o funcionamento da formulação paralela do Quicksort.

Percebe-se então que o Quicksort paralelo converge para $\lg p$ passos.

2.3.3 Detalhes de Implementação

Ao longo da implementação feita do Quicksort paralelo para este trabalho, alguns detalhes **importantes** devem ser considerados.

- **NENHUMA** estratégia de escolha de pivô foi usada;
- Conforme sugestão do professor, a função MPI_Comm_split foi utilizada para definir os conjuntos de processos, **PORÉM**, verifica-se que o tempo desta operação leva tempo $O(p^2)$, o que faz com que ela dite o tempo de execução do Quicksort paralelo como um todo;
- O processador de rank 0 lê a entrada e a distribui igualmente entre os outros processadores;

- Existem barreiras ao longo do código, apenas para conferir a corretude do algoritmo.

Usando do OpenMPI, o algoritmo foi implementado em ansi-C em sua forma recursiva. Para diminuir a quantidade de argumentos nas assinaturas das funções, estruturas auxiliares foram criadas, mas isto não interfere no funcionamento do algoritmo. Existem outras implementações do Quicksort paralelo, como o *hyperquicksort*, mas o foco foi implementar a mais básica.

Finalmente, conforme a implementação contida **neste trabalho**, sua análise assintótica é:

$$T_{quick} = O(\tfrac{n}{p} \lg \tfrac{n}{p}) + O(\tfrac{n}{p} \lg p) + O(\lg^2 p) + O(p^2 \lg p)$$

3 Experimentos/Resultados

Dadas as classes ministradas pelo professor Marco Aurélio Stefanes de algoritmos paralelos, no primeiro semestre de 2015 na FACOM- UFMS, pudemos iniciar a implementação desses algoritmos. Além da parte teórica, um momento foi reservado para a implementação do Splitsort, durante aulas práticas. Apesar dessa implementação ter nos ajudado de maneira substancial, alguns pequenos problemas ocorreram na execução desse programa para diferentes casos de teste, portanto foi necessário uma recodificação de uma porção do código para seu funcionamento correto, bem como a adição da leitura do arquivo de entrada e divisão do dado em três inteiros menores, de acordo com a quantidade de bits comparada.

É importante atentar para o fato de que o tempo para contagem foi iniciado logo após a distribuição dos dados pelo processador P0 e terminado após a ordenação de cada bucket por cada processador distinto. Os testes foram realizados utilizando o cluster do CTEI (Centro Tecnológico de Eletrônica e Informática), disponível para os alunos de graduação da UFMS.

Algumas outras considerações também dizem respeito à forma de execução: a biblioteca MPI (padrão) foi utilizada, juntamente com o argumento "-hostfile", apontando um arquivo que contém os endereços de todas as máquinas utilizadas para a execução dos algoritmos (de acordo com os exemplos explicitados abaixo, foram utilizadas 32 máquinas, no máximo).

3.1 Splitsort

Abaixo, seguem os tempos de execução do Splitsort, considerando a variação do número de processadores em 32 e da quantidade de dados de entrada em 10^6, $5 * 10^6$ e 10^7:

Nº processadores	10^6	$5 * 10^6$	10^7
1	0,73251	4,15146	8.83256
2	0,34439	1,91657	4.55325
4	0,22772	1,22570	2.44510
8	0,12836	0,70039	1.40191
16	0,06379	0,36593	0.81229
32	0,04133	0,21223	0.44083

Tabela 1: Tabela de tempo de execução em segundos por número de processadores

O gráfico abaixo foi gerado com os valores obtidos da tabela:

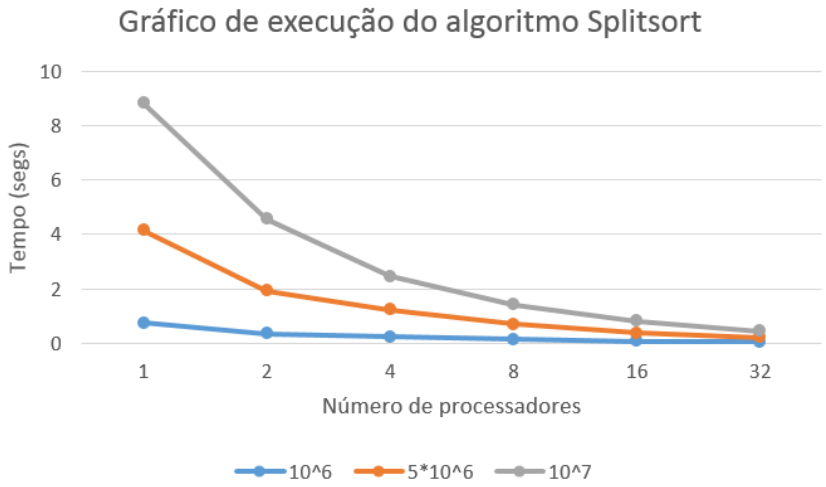

Figura 2: Gráfico de tempos de execução do Splitsort

Após uma análise desses resultados, pudemos verificar que o tempo de execução melhorou significativamente a partir da utilização de 8 processadores para o processamento dos dados, especialmente com 5 milhões e 10 milhões de números. Como utilizamos um arquivo de host, a distribuição foi feita de maneira real, e dada a propriedade do algoritmo de explorar ao máximo a amostragem para tentar obter distribuições balanceadas, isso realmente pôde ser visto com a diminuição drástica do tempo ao instanciar-se mais processadores para cada execução.

3.2 Samplesort

Abaixo, seguem os tempos de execução do Samplesort, considerando a variação do número de processadores em 32 e da quantidade de dados de entrada em 10^6, $5 * 10^6$ e 10^7:

Nº processadores	10^6	$5 * 10^6$	10^7
1	0,73435	4,18620	8.94633
2	0,39589	2,18825	4.54556
4	0,21490	1,15509	2.40008
8	0,10872	0,58840	1.22018
16	0,06419	0,42253	0.70214
32	0,03638	0,30702	0.53366

Tabela 2: Tabela de tempo de execução em segundos por número de processadores

O gráfico abaixo foi gerado com os valores obtidos da tabela:

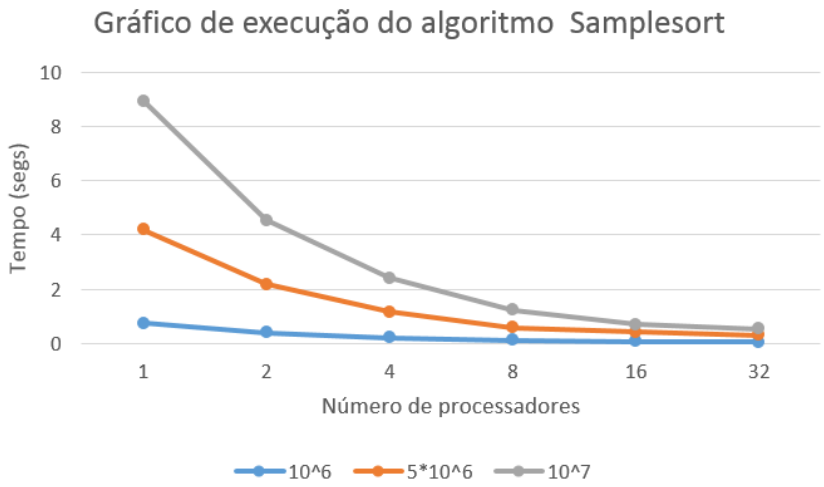

Figura 3: Gráfico de tempos de execução do Samplesort

Em geral, os resultados obtidos com esse algoritmo foram menos eficientes do que com o Splitsort, provavelmente devido a forma como os *buckets* foram organizados - no primeiro, a generalização de "quartis" foi adotada, ao passo que no Samplesort uma distribuição mais arbitrária foi utilizada; porém, apenas pela visualização do gráfico, essa mudança foi quase que imperceptível. A execução melhorou, também, drasticamente com 8 processadores instanciados.

3.3 Quicksort

Como o tempo de execução do comando MPI_Comm_split é da ordem de $O(p^2)$, seu tempo foi contado para comparar com o tempo total gasto do algoritmo. Serviu como lição para os alunos entenderem a complexidade do comando e procurarem algumas análises de comandos recorrentes de bibliotecas que usem o paradigma de troca de mensagens. Os tempos exibidos na tabela abaixo desconsideram o tempo adjunto da separação de comunicador e foram contados usando a função MPI_Wtime() da biblioteca OpenMPI:

Nº processadores	10^6	$5 * 10^6$	10^7
1	0.48484	2.70725	5.70536
2	0.45116	2.20545	5.17826
4	0.42194	1.89635	3.05781
8	0.26971	1.73421	2.55171
16	0.25219	0.57080	1.30995
32	0.12318	0.44918	1.18692

Tabela 3: Tempo de execução do algoritmo Quicksort com variação de entrada e processos, em segundos

Com esses dados, pudemos formular o seguinte gráfico:

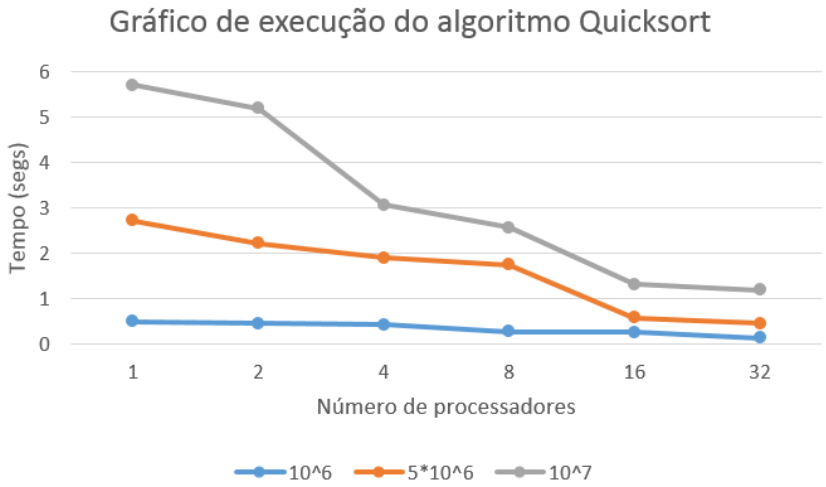

Figura 4: Gráfico de tempos de execução do Quicksort

Os resultados variaram bastante, pois, como explicado, não há nenhuma técnica de escolha do pivô. Por conta desta variação, os resultados escolhidos foram os melhores de uma bateria de testes, apenas por fins de demonstração. Vê-se uma melhora significativa de performance para entradas maiores com um número maior de processos – para entradas pequenas, o overhead de comunicação é a parte mais pesada do algoritmo, o que torna a paralelização dispensável.

3.4 Speedup

Após a execução desses algoritmos, restou, como comparação adicional, testar como eles se comportaram em relação ao tempo de execução do quicksort sequencial; esse foi implementado utilizando a função `qsort`, disponível na biblioteca padrão da linguagem C. Abaixo, para fins de comparação, segue a tabela contendo os tempos de execução desse algoritmo:

Entrada	10^6	$5 * 10^6$	10^7
Quick Sequencial	0,23101	1,26966	2,66356

Tabela 4: Tabela de tempo de execução em segundos do quicksort sequencial

Com base nessas informações, seguem abaixo uma tabela e um gráfico demonstrando o *speedup* desses algoritmos em relação ao tempo de execução do quicksort sequencial:

Nº processadores	10^6	$5 * 10^6$	10^7
1	0,47646	0,46898	0,46685
2	0,67077	0,66246	0,58596
4	1,07496	1,09919	1,10977
8	2,12474	2,15779	2,18292
16	3,62143	3,46962	3,79347
32	6,34860	5,98237	6,04205

Tabela 5: Tabela de *speedup* obtido por número de processadores

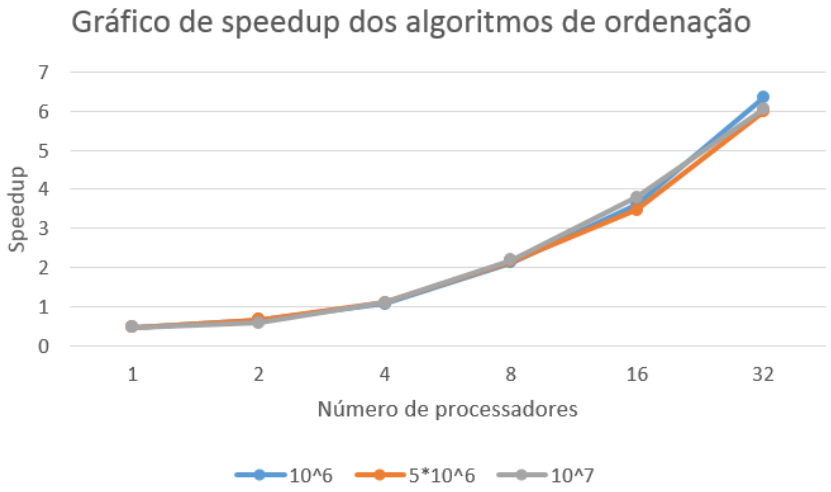

Figura 5: Gráfico de Speedup por número de processadores

Intuitivamente, o *speedup* obtido para as diferentes entradas é bastante similar. Para a geração desse gráfico, os melhores tempos serviram de divisor para o tempo de execução do quicksort sequencial, obtendo os valores supracitados. Também é importante notar que o *speedup* é uma unidade de medida *adimensional*, ou seja, não existe dimensão física aplicável.

Verificou-se, através da análise desse gráfico, que, com poucos processadores, houve pouca mudança em relação ao tempo de execução do quicksort sequencial; entretanto, a partir da execução com 4 processadores, uma melhora pode ser vista, chegando a um benefício muito drástico na execução com 32 processadores.

4 Conclusão

A partir do que nos foi ensinado durante a disciplina de Algoritmos Paralelos, pudemos aprender, na prática, como se dá o funcionamento de alguns algoritmos de ordenação tendo uma explícita paralelização embutida a eles, bem como o quão diferente o tempo de execução resultante é. A utilização de um *cluster* para obtenção dos resultados também só acrescentou ao aprendizado, visto que realmente a ordenação ocorreu entre diferentes máquinas, pondo em prática as fórmulas que envolviam o *startup time* (t_s) e o *per-word transfer time* (t_w).

O momento de maior percepção de que os algoritmos paralelos realmente são mais eficientes que as execuções sequenciais aconteceu no cálculo dos *speedups* - que chegaram a ser até 6 vezes mais rápido que o original. As análises dos gráficos e tabelas também permitiram com que víssemos com mais facilidade o quão rápido os algoritmos se tornaram ao adicionar-se mais processadores.

Portanto, fomos capazes de compreender como se dá o funcionamento de três algoritmos de ordenação com uma abordagem paralela, suas principais dificuldades (especialmente as relativas à comunicação) e o que é necessário para se chegar a tal implementação. Com isso, essa compreensão nos permite, na medida do possível, tomar decisões a respeito de explicitação de paralelização de algoritmos de ordenação.

5 Bibliografia

A. Grama, A. Gupta, G. Karpys and V. Kumar. Introduction to Parallel Computing. 2nd Ed. Addison Wesley, 2006. Slides disponíveis no EAD na disciplina de Algoritmos Paralelos.