CENTRO UNIVERSITÁRIO VILA VELHA

CURSO DE CIÊNCIA DA COMPUTAÇÃO

Christyan Brandão Oliveira

Felipe Nogueira Gáudio

ORDENAÇÃO DE DADOS EM MEMÓRIA EXTERNA UTILIZANDO PROGRAMAÇÃO PARALELA

VILA VELHA

2008

Christyan Brandão Oliveira

Felipe Nogueira Gáudio

ORDENAÇÃO DE DADOS EM MEMÓRIA EXTERNA UTILIZANDO PROGRAMAÇÃO PARALELA

Trabalho de Conclusão de Curso apresentado ao Centro Universitário de Vila Velha como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.
Orientador: Leonardo Muniz de Lima

VILA VELHA

2008

Christyan Brandão Oliveira

Felipe Nogueira Gáudio

ORDENAÇÃO DE DADOS EM MEMÓRIA EXTERNA UTILIZANDO PROGRAMAÇÃO PARALELA

BANCA EXAMINADORA

Prof. Msc. Leonardo Muniz de Lima
Centro Universitário Vila Velha
Orientador

Prof. Msc. Cristiano Biancardi
Centro Universitário Vila Velha

Prof. Msc. Vinicius Rosalén da Silva
Centro Universitário Vila Velha

Trabalho de Conclusão de Curso

aprovado em 26/11/2008.

Aos nossos pais, amigos e professores que nos apoiaram e ajudaram nesta caminhada...

AGRADECIMENTOS

Agradecemos às nossas famílias e amigos pelo apoio durante esses quatro anos de luta e saudamos o professor e orientador Leonardo Muniz de Lima, pelo empenho durante o desenvolvimento deste trabalho.

"O sábio envergonha-se dos seus defeitos, mas não se envergonha de os corrigir."

Confúcio.

LISTA DE TABELAS

LISTA DE FIGURAS

SUMÁRIO

RESUMO

Este trabalho apresenta um algoritmo desenvolvido a partir de um estudo sobre ordenação de dados em memória externa utilizando a computação paralela. O algoritmo utiliza um tipo de ordenação com base na arquitetura de memória distribuída, mais precisamente *clusters*, onde na criação do mesmo é utilizada a biblioteca MPI. São apresentadas fundamentações teóricas sobre o assunto que passam por métodos de ordenação, computação paralela e programação paralela. Ao final do desenvolvimento deste trabalho, são mostrados testes de desempenho do algoritmo desenvolvido em comparação a dois outros implementados, seguido de suas considerações finais.

Palavras-chave: ordenação em memória externa, *clusters*, programação paralela e MPI

1 INTRODUÇÃO

A busca do poder computacional vem sendo uma das maiores questões desde o início da computação. Dia após dia, o homem necessita de mais ciclos de CPU[1] para resolver problemas, como cálculos matriciais, previsões do tempo, ordenação de grandes massas de dados, entre outros. Segundo Tanembaum[1], não importa quanta tecnologia há, nunca será suficiente.

primeiramente, investiu-se na computação seqüencial, que era a execução de tarefas a partir de um único processador. Quanto maior a freqüência que esse processador operava, mais poder computacional existia. Com a evolução da tecnologia, os *chips* diminuíam, e com eles um problema: a dissipação do calor. Quanto maior a freqüência, maior o calor gerado. E, quanto menor o *chip*, mais difícil de se livrar desse calor. Além disso, o maior fator para a busca de novas tecnologias era a latência[2] entre o processador e a memória.

Com o passar dos anos, os processadores evoluíram muito mais que as memórias, o barramento para alimentação era o mesmo e, independente da freqüência operada, o processador tinha que esperar esses dados chegarem para processá-los. Mesmo usando *pipeline*[3], o poder computacional gerado não era suficiente. Um meio de aumentar a velocidade foi o uso de computadores em paralelo. Máquinas essas, constituídas de muitas CPU's, cada qual executando em velocidade 'normal', porém coletivamente com uma velocidade maior do que uma única CPU.

Dentre as aplicações citadas anteriormente, a ordenação de dados tem sido um grande desafio da programação. Seja ela utilizando computação sequencial ou computação paralela, principalmente, se a massa de dados for extensa. Dessa forma, tais informações não caberiam na memória principal forçando o sistema a fazer acessos aos dispositivos de memória secundária e tornando o procedimento de ordenação muito mais lento. Esse tipo de ordenação é conhecida como ordenação em memória externa[4], e para obter melhores resultados é necessário reduzir o número de tais

acessos. Na computação sequencial essa diminuição se torna dificil, uma vez que a quantidade de memória primária é limitada seja por custo ou por capacidade. Por outro lado, a computação parelela permite que cada máquina ordene uma quantidade de dados que teoricamente, caberiam na memória principal. Tornando assim o processo relativamente mais rápido.

Dentre os diversos trabalhos encontrados na literatura que apresentam soluções para a ordenção de dados em paralelo, podemos destacar o estudo desenvolvido por Pawlowski e Bayer[4]. Neste trabalho, é apresentada uma forma eficiente de escrever programas em paralelo para o problema da ordenação de uma grande quantidade de informações utilizando o paradigma MPI. Essa mesma forma é abordada no presente trabalho.

Outro trabalho correlato é o artigo de Hammao e Jonathan[5], que mostra como o *speedup* depende largamente da latência da memória onde a escalabilidade e sincronização do *overhead* pode ser minimizada. Um ponto comum entre os dois trabalhos é a utilização do *speedup* como principal medida de análise. Esta medida também é utilizada neste trabalho.

Com outro exemplo similar, podemos citar o experimento apresentado por Daniel Cleary[6]. Ele descreve que a biblioteca de programação paralela LAM-MPI[18] é mais eficiente para esse tipo de operação do que a MPICH[18]. Mesmo que ambas possuam o mesmo paradigma *Message Passing*. Este experimento mostrou tal eficiência levando em consideração alguns testes. Sendo assim, não pode ser considerado que esta implementação sempre será mais rápida na troca de mensagens do que a MPICH.

Tendo como base teórica os artigos anteriores, o presente trabalho visa apresentar a programação paralela como ferramenta para alcançar um melhor desempenho na ordenação de dados em memória externa. Desta forma, o objetivo do mesmo é apresentar o algoritmo paralelo desenvolvido e os resultados da execução deste na ordenação de uma grande massa de dados em memória externa, para mostrar com isso que a utilização de ambientes paralelos torna tal ordenação mais eficaz. Inicialmente, apresenta-se um resumo sobre a ordenação de dados. Em seguida, realiza-se um pequeno estudo sobre as arquiteturas paralelas e alguns paradigmas no contexto da programação paralela. Por fim, são apresentados os resultados da ordenação de dados utilizando computação seqüencial e paralela em memória interna e externa.

2 INTRODUÇÃO AOS ALGORITMOS DE ORDENAÇÃO

A ordenação é utilizada para organizar elementos de uma seqüência em uma determinada ordem, sendo um dos problemas fundamentais no estudo dos algoritmos [7]. Segundo Donald Knuth, "Mesmo se a ordenação de dados fosse quase inútil, haveria motivos gratificantes para estudá-la assim mesmo! Engenhosos algoritmos que têm sido descobertos, revelam que este é um tópico muito interessante para ser explorado por si mesmo. Ainda há muitos problemas que continuam sem ser resolvidos nesta área, assim como para um punhado deles já existe solução"[7].

As aplicações do processo, de organizar os dados de forma crescente ou decrescente, são bastante amplas. A pesquisa binária é um destes exemplos, visto que a mesma necessita que a seqüência de elementos esteja ordenada. Neste caso, a ordenação é utilizada para organizar os elementos e viabilizar a busca. Deve-se levar em consideração a finalidade da pesquisa, ou seja, caso necessite encontrar apenas um elemento, não seria vantajoso ordenar os dados para depois buscá-los. Seria mais fácil comparar um a um até achá-lo. Por outro lado, se a necessidade da busca for constante, a ordenação torna-se mais viável. Outra aplicação bem interessante seria a verificação da duplicidade de elementos. Parte-se do mesmo princípio da aplicação anterior, onde é mais rápida a varredura linear à verificação da combinação de cada par de elementos.

Apesar de que, alguns destes problemas podem ser resolvidos em tempo linear a partir dos mais sofisticados algoritmos e a ordenação fornece uma forma fácil e rápida para solucionar os problemas. Conforme mencionado anteriormente, se as operações forem repetidas muitas vezes, é vantajoso ordenar a seqüência primeiramente, uma vez que são raras as operações que, feitas sobre uma massa de dados desordenada, demandariam menos tempo para serem resolvidas se comparadas à mesma massa de dados ordenada.

Como pode ser verificado na Tabela.1, existem vários algoritmos de ordenação. Surgem perguntas do tipo: "por que existem tantos deles?"e "por que o melhor deles não é usado em todos os casos?". Vários fatores são levados em consideração, como a complexidade de tempo e espaço. Mas o 'verdadeiro' desempenho de um algoritmo depende do sistema de codificação utilizado, da quantidade e das características de seus dados. Precisa-se também considerar a facilidade de codificação, a compactação do código, a previsibilidade de consumo de tempo e o estado intermediário dos dados.

	Pior Caso	Caso Médio	Melhor Caso	Espaço Extra	Estável?
BublleSort	$O(n^2)$	$O(n^2)$?	$O(n)$	$O(1)$	sim
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	sim
InsertionSort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	sim
BitonicSort	$O(nlog^2n)$	$O(nlog^2n)$?	?	$O(1)$?	?
ShellSort	$O(n^2)$	$O(nlogn)$?	$O(n)$	$O(1)$	não
QuickSort	$O(n^2)$	$O(nlogn)$	$O(nlogn)$	$O(logn)$	não
HeapSort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$	não
SmoothSort	$O(nlogn)$	$(O(nlogn)$?	$O(n)$	$O(1)$	não
MergeSort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	sim
CountingSort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	sim
RadixSort	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	sim
BogoSort	não delimitado	$O(n!)$	$O(n)$	$O(1)$	não
SlowSort	$O(n^{(logn)}))$	$O(n^{(logn)})$	$O(n^{(logn)})$	$O(1)$	sim

Tabela 1: Tabela comparativa entre algoritmos de ordenação e parâmetros de configuração.

A maioria dos métodos não leva em consideração o tamanho do bloco de dados. Quando esta informação cabe na memória principal do computador, a operação é chamada de ordenação interna. Caso contrário, a organização é efetuada através da ordenação em memória externa.

2.1 Ordenação em memória interna

A ordenação em memória interna é caracterizada pelo armazenamento de todos os registros na memória principal, onde seus acessos são feitos diretamente pelo pro-

cessador. Isto é possível quando a quantidade de dados a ser ordenada é suficientemente pequena para o processo ser realizado basicamente na memora principal. Quase todos os algoritmos de ordenação estudados pertencem a esta classificação. Tais procedimentos podem ser categorizados através de três técnicas empregadas: os métodos de inserção, seleção e partição.

2.1.1 Ordenação por inserção

Os principais são o método da bolha[7] e o método de inserção direta[7]. O primeiro é baseado na troca de elementos entre posições consecutivas, sempre que um antecessor for maior que seu sucessor. É o mais conhecido e não é eficiente pelo fato de realizar uma quantidade excessiva de comparações. Já o algoritmo de inserção direta consiste em percorrer um vetor de elementos da esquerda para a direita, transportando os elementos maiores para o fim do vetor e ordenando-o crescente ou decrescentemente, se for o caso. Esta estratégia é indicada para vetores quase ordenados. Ambos possuem uma implementação simples e são métodos estáveis, ou seja, a ordem relativa dos itens com chaves iguais se mantém inalterada após a ordenação. A complexidade do algoritmo de inserção direta é, no melhor caso: $O(n)$, e, no pior caso: $O(n^2)$. Se os dados já estiverem praticamente ordenados, serão feitas comparações em uma única passagem pela lista de elementos, o que implica numa complexidade linear. É um dos poucos exemplos onde um método realiza suas etapas abaixo do *lower bound* da ordenação, que é $\Omega(nlogn)$.

2.1.2 Ordenação por seleção

Em uma ordenação por seleção, os elementos sucessivos são selecionados na ordem e inseridos em suas respectivas posições. Os elementos de entrada são pré-processados para tornar a seleção ordenada possível. Os métodos de seleção direta e o *heapsort* serão apresentados a seguir:

Seleção direta

É um dos mais simples algoritmos de ordenação[7]. Ele percorre todos os elementos identificando o menor e troca-o com o da primeira posição, repete o mesmo processo e troca com o da segunda posição, até chegar ao fim. Não é um algoritmo

estável e sua ordenação parcial não interfere no processo pelo fato dele procurar o menor elemento até o final. Sua ordem de complexidade é $O(n^2)$, assim como no método da bolha, embora seja mais rápido.

Heapsort

O *heapsort* foi desenvolvido em 1964 por Robert W. Floyd e J.W.J. Williams. Ele utiliza uma estrutura de dados chamada *heap*, que pode ser uma árvore ou um vetor, para a ordenação dos elementos à medida que são inseridos. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada. Para uma ordenação crescente, o maior elemento deve ficar na raiz e, para uma ordenação decrescente, o menor. É um método não estável e sua ordem de complexidade é de $O(nlogn)$. É um dos únicos algoritmos que possui esta complexidade e não parte do princípio de "dividir para conquistar".

2.1.3 Ordenação por partição

Os algoritmos que utilizam esta abordagem partem do princípio de "dividir para conquistar". Este princípio é uma técnica de projeto de algoritmos utilizada pela primeira vez por Anatolii Karatsuba em 1960 e consiste em dividir um problema maior em problemas pequenos, e sucessivamente até que o mesmo seja resolvido diretamente. Esta técnica realiza-se em três fases:

- **Divisão:** o problema maior é dividido em problemas menores e os problemas menores obtidos são novamente divididos sucessivamente de maneira recursiva.
- **Conquista:** o resultado do problema é calculado quando o problema é pequeno o suficiente.
- **Combinação:** o resultado dos problemas menores são combinados até que seja obtida a solução do problema maior.

Algoritmos que utilizam o método de partição são caracterizados por serem os mais rápidos dentre os outros algoritmos pelo fato de sua complexidade ser, na maioria das situações, $O(nlogn)$. Os dois representantes mais ilustres desta classe são o *quicksort* e o *mergesort*.

Quicksort

O algoritmo *quicksort* é um método de ordenação muito rápido e eficiente, inventado por C.A.R. Hoare em 1960, quando visitou a Universidade de Moscovo como estudante. Foi publicado em 1962 após uma série de refinamentos. Ele seleciona um elemento aleatório como pivô e particiona o resto colocando os menores à esquerda e os maiores à direita. Faz-se o mesmo com os elementos da esquerda e direita até não ser mais possível particioná-los, obtendo-se os elementos ordenados. Sua implementação pode ser estruturada ou recursiva. onde a recursiva é mais custosa por fazer várias chamadas à mesma função e necessitar de uma pequena pilha como memória auxiliar. É um método não-estável e sua melhor eficiência depende da escolha do pivô, onde uma boa escolha pode variar de uma complexidade de $O(nlogn)$ a $O(n^2)$. Muito utilizado na prática, é o algoritmo mais rápido para ordenações em memória interna. Seus passos estão definidos abaixo, a partir de seu paradigma de "dividir para conquistar".

- **Dividir:** rearranje a lista de forma que todos os elementos anteriores ao pivô sejam menores ou iguais a ele, e todos os elementos posteriores ao pivô sejam maiores ou iguais a ele. Ao fim do processo o pivô estará em sua posição final.
- **Conquistar:** recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores.
- **Combinar:** junte as listas ordenadas e o pivô.

Mergesort

O algoritmo *mergesort* foi inventado em 1945 por John Von Neumann[8] e funciona basicamente em 3 passos, como mostrado na Figura.1 abaixo.

- **Dividir:** dividir a lista em duas listas com cerca da metade do tamanho.
- **Conquistar:** dividir cada uma das duas sublistas recursivamente até que tenham tamanho um.
- **Combinar:** fundir as duas sublistas de volta em uma lista ordenada.

O mergesort é um algoritmo estável na maioria de suas implementações, onde estas podem ser iterativas ou recursivas. Sua desvantagem é o fato de utilizar uma

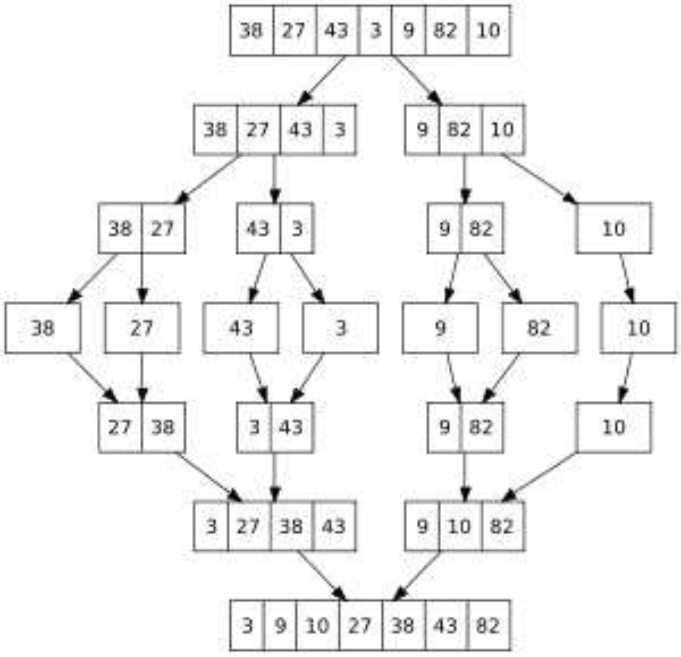

Figura 1: Funcionamento do algoritmo *mergesort*.

estrutura auxiliar, ocupando o dobro de memória. É interessante destacar suas características em cima do paradigma de "divisão para conquista":

- **Dividir:** se a seqüência tiver mais de um elemento, divida em duas partes.
- **Conquistar:** ordene cada subseqüência em separado usando *mergesort*.
- **Combinar:** junte as duas subseqüências em uma seqüência ordenada.

É comprovado cientificamente que é praticamente impossível fazer um algoritmo mais eficiente [7], sendo sua ordem de complexidade $O(nlogn)$. Seu pseudocódigo pode ser definido como mostra a Figura.2.

A operação de fusão *merge*, do *mergesort*, é muito utilizada na busca *online*, onde os dados chegam de blocos em blocos, são ordenados por qualquer método e depois fundidos pela mesma. No entanto, esta abordagem pode demandar muito tempo e espaço de armazenamento se os blocos recebidos forem pequenos em comparação com os dados ordenados[7].

```
1  function mergesort(m)
2     var lista esquerda, direita, resultado
3     if tamanho(m) = 1
4        return m
5     else
6        var meio = tamanho(m) / 2
7     for each x in m up to meio
8        adicionar x to esquerda
9     for each x in m after meio
10       adicionar x to direita
11    esquerda = mergesort(esquerda)
12    direita = mergesort(direita)
13    resultado = merge(esquerda, direita)
14    return resultado
15
16 function merge(esquerda,direita)
17    var lista resultado
18    while tamanho(esquerda) > 0 and tamanho(direita) > 0
19       if primeiro(esquerda) = primeiro(direita)
20          concatena primeiro(esquerda) to resultado
21          esquerda = resto(esquerda)
22       else
23          concatena primeiro(direita) to resultado
24          direita = resto(direita)
25    end while
26    if tamanho(esquerda) > 0
27       concatena resto(esquerda) to resultado
28    if tamanho(direita) > 0
29       concatena resto(direita) to resultado
30    return resultado
```

Figura 2: Pseudo-código do algoritmo *mergesort*.

2.2 Ordenação em Memória Externa

Quando o número de registros a ser ordenado é maior do que o computador pode suportar em sua memória principal, surgem problemas interessantes a serem estudados. Embora o fato de a ordenação em memória interna e externa possuírem o mesmo objetivo, elas são muito diferentes, uma vez que o acesso eficiente aos arquivos muito extensos é bastante limitado. As estruturas de dados devem ser organizadas de modo que lentos dispositivos periféricos de memória, geralmente discos rígidos, possam rapidamente lidar com as exigências do algoritmo. Conseqüentemente, a maioria dos algoritmos de ordenação interna é praticamente inútil para a ordenação externa. Por conseguinte, se torna necessário repensar toda a questão[7].

Supondo um arquivo a ordenar de 1 *gigabyte* de dados e uma máquina com 256 *megabytes* de memória *RAM*. Um algoritmo normal criaria um vetor para colocar todos os dados e faria comparações até o mesmo ficar ordenado. Como a máquina só possui 256 *megabytes* de memória interna, não caberia o vetor inteiro, então parte ficaria em sua memória *RAM* e o restante em disco. Ao efetuar as comparações, os dados que não estivessem na *RAM* deveriam ser buscados em disco, trazidos para a

memória principal e comparados. Este processo se repetiria inúmeras vezes, onde o processador ficaria a maior parte do tempo ocioso esperando os dados chegarem à *RAM* para serem processados. Com isso, o desempenho deixaria muito a desejar.

Uma forma de tratar este problema seria controlar o acesso à disco. A utilização do algoritmo de ordenação *mergesort* é bastante eficaz pela sua característica de *merge*(junção) de blocos ordenados. O arquivo a ordenar seria dividido em vários outros, em tamanhos possiveis para o armazenamento completo de seus dados em memória principal. Desta forma, ordenar-se-ia todos os dados fazendo menos acessos à disco. Em seguida, seria efetuado o *merge* entre estes dados ordenados. Agindo dessa maneira, o resultado seria satisfatório, dado que, reduzindo as leituras e escritas em memória secundária, o processamento aconteceria mais rapidamente.

Basicamente a ordenação em memória externa é utilizada para grandes arquivos que não cabem na memória principal e utilizam uma memória auxiliar[9] para armazená-los. Imagine, por exemplo, a ordenação de 900 *megabytes* de dados com apenas 100 *megabytes* de memória *RAM*, usando o *mergesort*, como pode ser visto na Figura.3. De acordo com Donald Knuth[7], é apresentado um exemplo como se segue:

Figura 3: Representação da memória e do arquivo a ordenar.

1. Leia os 100MB de dados que estão na memória principal, ordene-os. A Figura.4 mostra a divisão do arquivo em tamanhos de 100MB, o tamanho da memória, para não ocorrer acesso a disco. Neste caso, é lido o vetor 1.

2. Escreva os dados em disco. Os arquivos ordenados, representados de amarelo na Figura.5 já passaram pela memória principal, foram ordenados, e agora encontram-se em disco.

3. Repita o procedimento de 100MB em 100MB até completar os 900MB de dados, que agora precisam ser fundidos em um único arquivo de saída.

4. Leia os 10 primeiros *megabytes* de dados de cada pedaço ordenado, totalizando 90MB, chamados de *buffers* de entrada. Os 10MB restantes são alocados para

o *buffer* de saída.

5. Realiza a função *merge* nos 9 *buffers* de entrada para fundir e armazenar o resultado no *buffer* de saída. Quando estiver cheio, escreva-o no final do arquivo ordenado. Se qualquer um dos 9 *buffers* de entrada estiver vazio, preencha-o com os próximos 10MB dos 100MB associados, ou então marque-o como esgotado e a função *merge* não irá utilizá-lo. Esta representação pode ser vista na Figura.6 onde a memória é composta por 10MB de cada arquivo em disco e o *buffer* de vermelho.

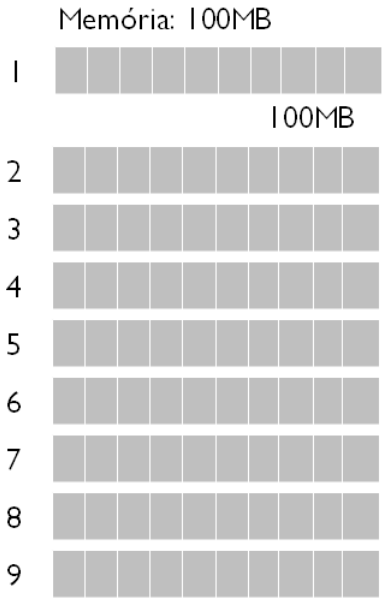

Figura 4: Memória e o arquivo a ordenar particionado.

Mesmo utilizando técnicas de ordenação em memória externa, em duas situações típicas apresentam problemas: quando a quantidade de memória física disponível é pequena ou quando a massa de dados é tão grande que a redução de acessos à disco não é suficiente para diminuir significativamente o tempo de processamento. Neste exemplo, é possível utilizar a programação paralela para otimização do processo. Cada computador ficaria com uma parte dos dados e os ordenaria parcialmente, fundidos no final por um computador mestre determinado. Para um resultado satisfatório, é necessário o uso somente da memória interna, para não ter de haver a busca em disco dos dados. Quanto maior o número de computadores, melhor. Assim a quantidade de dados na memória externa tende a diminuir.

Uma implementação deste algoritmo paralelo pode ser vista na Figura.7. Seu funcionamento segue cinco passos fundamentais:

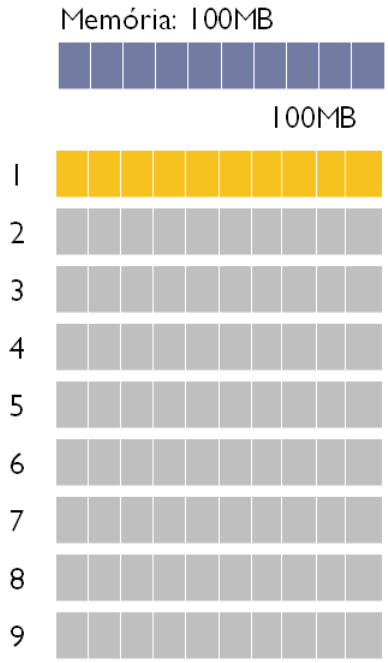

Figura 5: Primeiro arquivo ordenado.

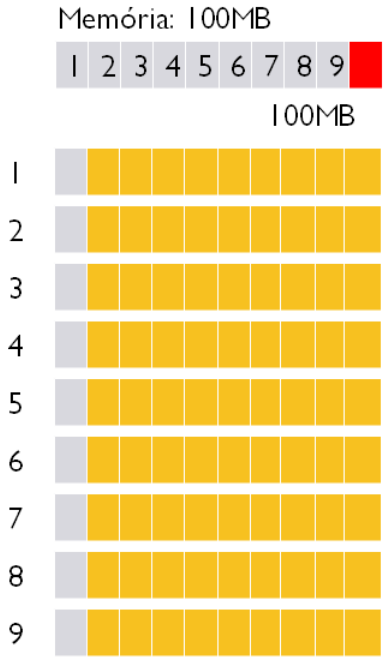

Figura 6: Memória contendo a primeira parte de cada arquivo ordenado.

1. Dividir o arquivo à ordenar no número de processadores envolvidos.
2. Cada processador ordena sua parte do arquivo respectivo.
3. O processador nomeado como 1 envia sua parte ordenada para o processador 0, este junta sua parte ordenada com a recebida e faz a fusão (*merge*). O processador 1 finaliza sua participação no algoritmo. Isto é, feito para todos os processadores, onde n envia e n-1 recebe.
4. O passo 3 é repetido até sobrar somente o processador 0 com o vetor ordenado.
5. O vetor ordenado é gravado no arquivo e se tem a ordenação completa.

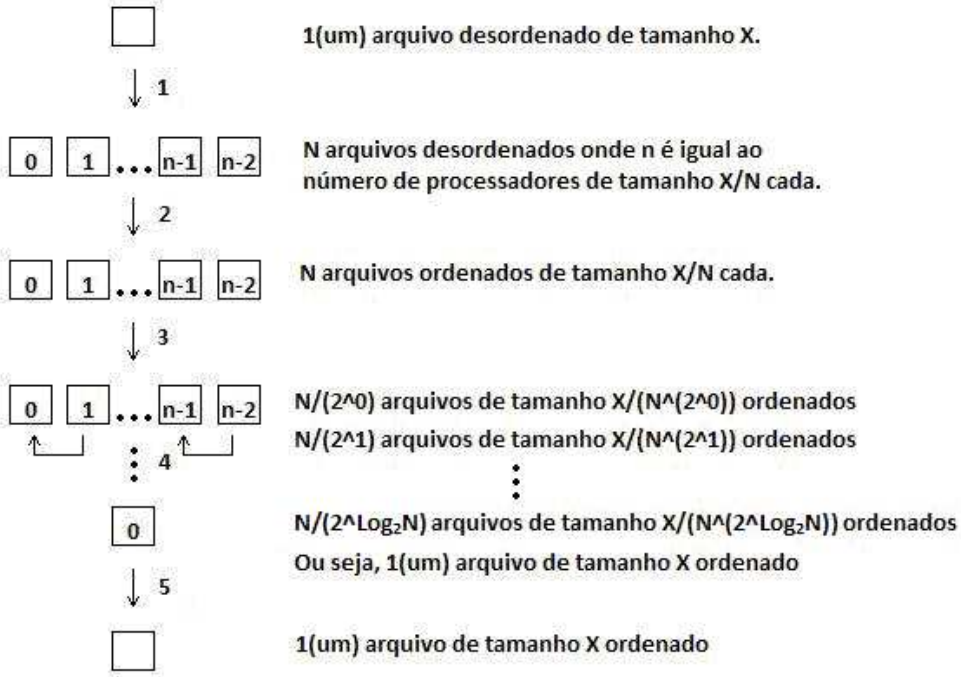

Figura 7: Fluxograma do algoritmo paralelo.

Parece óbvio dizer que um algoritmo de ordenação operando em um sistema paralelo será mais rápido que em um seqüencial. Existem vários fatores que poderiam fazer esta afirmativa falsa. Um primeiro seria a rede de interconexão dos computadores em paralelo. Esta pode ser o gargalo da operação, onde a troca de mensagens é fundamental entre os computadores. Se esta troca for lenta, não importa o número de computadores que operam juntos, eles precisam enviar e receber dados para continuarem o processo, e isto poderia fazer o algoritmo seqüencial mais rápido. Outro fator é a quantidade de dados a ordenar. Caso seja uma pequena massa, a troca de

mensagens pela rede, mesmo que seja de alta velocidade, é mais lenta que a transferência de dados em um barramento, portanto um sistema seqüencial poderia ter um melhor desempenho também.

3 INTRODUÇÃO Á COMPUTAÇÃO PARALELA

De forma geral, pode-se destacar 3 tipos de paralelismo computacional: Multiprocessadores[10], Multicomputadores[11] e Sistemas Distribuídos[12], veja a Figura.8.

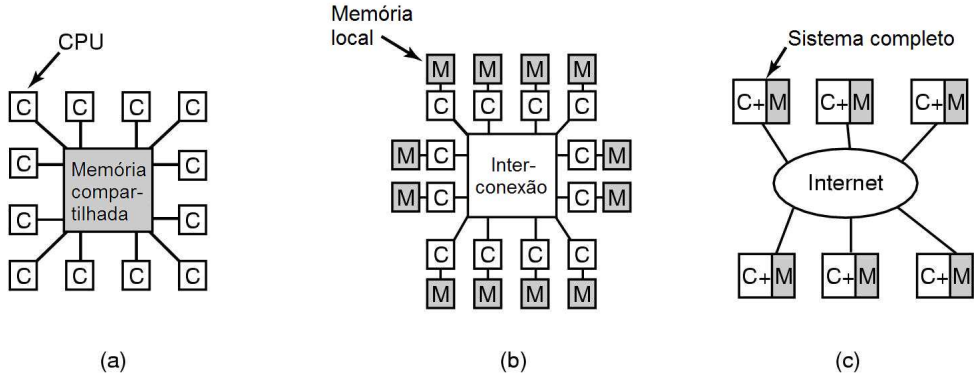

Figura 8: Organização da computação paralela. (a) multiprocessador de memória compartilhada. (b) multicomputador com troca de mensagens. (c) sistema distribuído fracamente acoplado.

3.1 Multiprocessadores

Um multiprocessador de memória compartilhada, ou somente multiprocessador, é um sistema de computadores em que duas ou mais CPU's compartilham acesso total à uma memória comum[1], como mostra a Figura.9.

Além de possuir características de processadores normais, a sincronização de processos, o gerenciamento de recursos e o escalonamento são únicos. Alguns multiprocessadores possuem propriedades especiais, pode-se citar o acesso uniforme à memória (UMA)[1] e o acesso não uniforme à memória (NUMA)[1]. Os mais simples utilizam o mesmo barramento para acesso à memória, apresentado pela Figura.10(a).

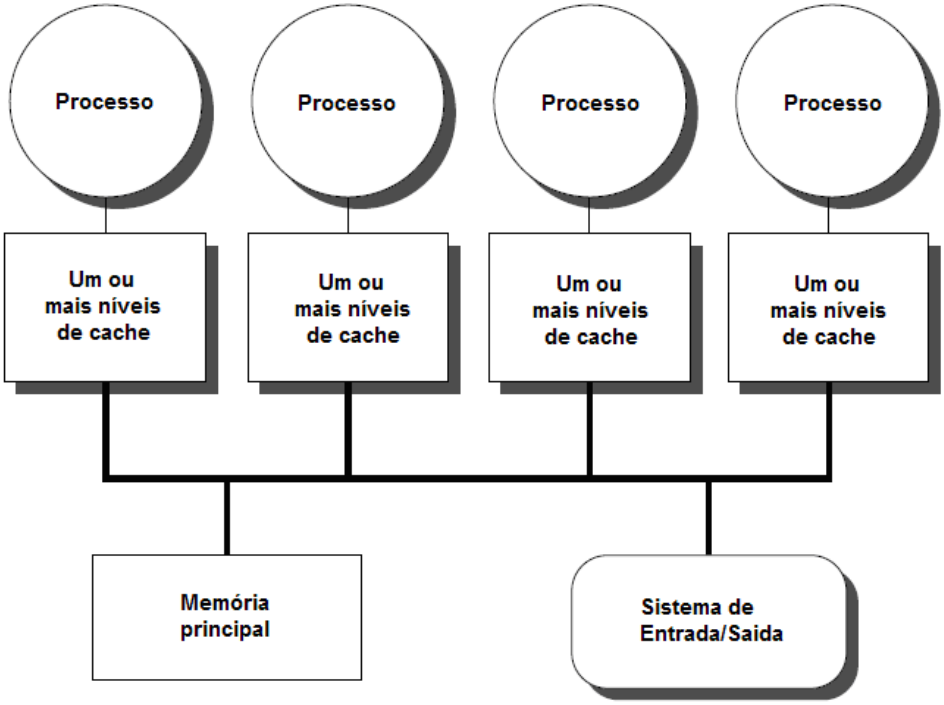

Figura 9: Multiprocessador com memória compartilhada.

Quando se tem poucos processadores o sistema funciona bem. Porém quando a quantidade processadores é menor, o barramento fica congestionado pelo fato destes processadores tentarem utilizá-lo ao mesmo tempo, deixando assim o sistema inviável. Uma primeira solução seria o uso de uma cache para cada processador, diminuindo o tráfego no barramento, conforme pode ser visto na Figura.10(b). Neste modelo cada cache conteria um bloco de dados da memória fazendo com que o processador utilizasse menos o barramento. Esses blocos seriam marcados como somente leitura (vários processadores) ou leitura e escrita (no máximo um). Quando um processador alterasse um determinado bloco de dados, avisaria aos outros que o estavam utilizando como somente leitura, que esses dados tinham sido atualizados. Outra possibilidade era o uso além de uma cache, de uma memória local e privada para cada processador, caso apresentado na Figura.10(c). Essa memória conteria informações de somente leitura, o código do programa, entre outros. Fazendo assim com que a memória compartilhada fossi usada somente para variáveis compartilhadas que poderiam ser escritas, reduzindo seu uso. Mesmo com o melhor sistema de cache, um barramento só limitaria a quantidade de processadores para esse sistema.

Foi criado outro meio de comunicação entre esses processadores com o nome de chave *crossbar*, ilustrado pela Figura.11, que era basicamente uma matriz onde se podia ligar CPU's (linhas horizontais) à memórias(linhas verticais) a partir de um

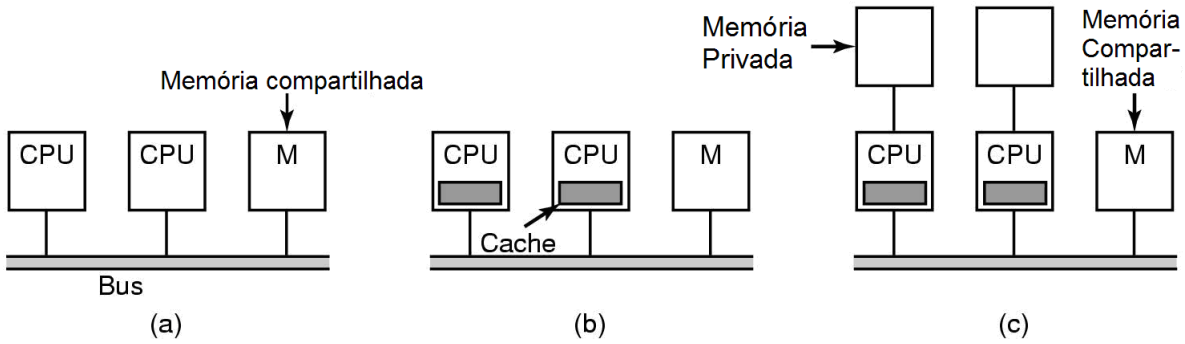

Figura 10: Modelos de multiprocessadores baseados em barramentos. (a) Sem a utilização de cache. (b) Com utilização de cache. (c) Com memórias privadas e utilização de caches.

cruzamento (*crosspoint*) elétrico.

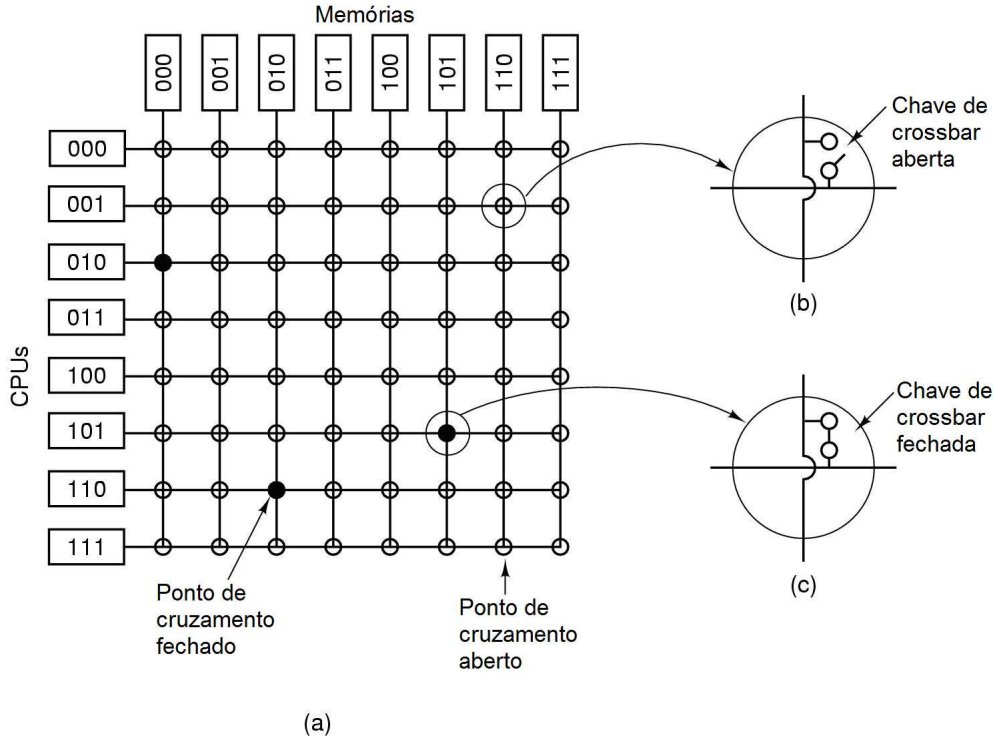

Figura 11: Multiprocessador UMA que utiliza chave *crossbar*. (a) Uma chave *crossbar* 8x8. (b) Um ponto de cruzamento aberto. (c) Um ponto de cruzamento fechado.

Uma vantagem é o fato de que ela é uma rede não bloqueante, ou seja, cada CPU pode se conectar à cada memória independente se ela já esteja sendo usada. A maior desvantagem é que, por ser uma matriz, o ponto de cruzamento cresce exponencialmente, ficando inviável para grandes sistemas, mas funcional para sistemas de médio porte. Já para sistemas de grande porte, utiliza-se o conceito de processadores NUMA, que são semelhantes aos UMA, mas todos os módulos de memória têm o mesmo tempo de acesso[1]. Ou seja, embora todos os processadores possam

acessar todas as posições de memória, os tempos de acesso variam de acordo com o endereço acessado[13][14][15]. Com isso, há a diminuição do tráfego na rede de interconexão e o acesso se torna mais rápido por fazer referência à uma memória local e não remota. As memórias são visíveis à todas CPU's. Uma desvantagem é que a comunicação entre processadores é muito complexa e tem maior latência, pelo fato de acessarem memórias remotas. Quando não existe cache, o sistema é chamado de NC-NUMA[1], e quando possue coerência de cache, CC-NUMA[1]. O CC-NUMA[1], mais usado nos sistemas multiprocessadores, é também chamado de multiprocessador baseado em diretório, exemplificado na Figura.12. No mesmo local o diretório funciona como uma base de dados, informando a localização de várias linhas de cache de memória e seu *status*.

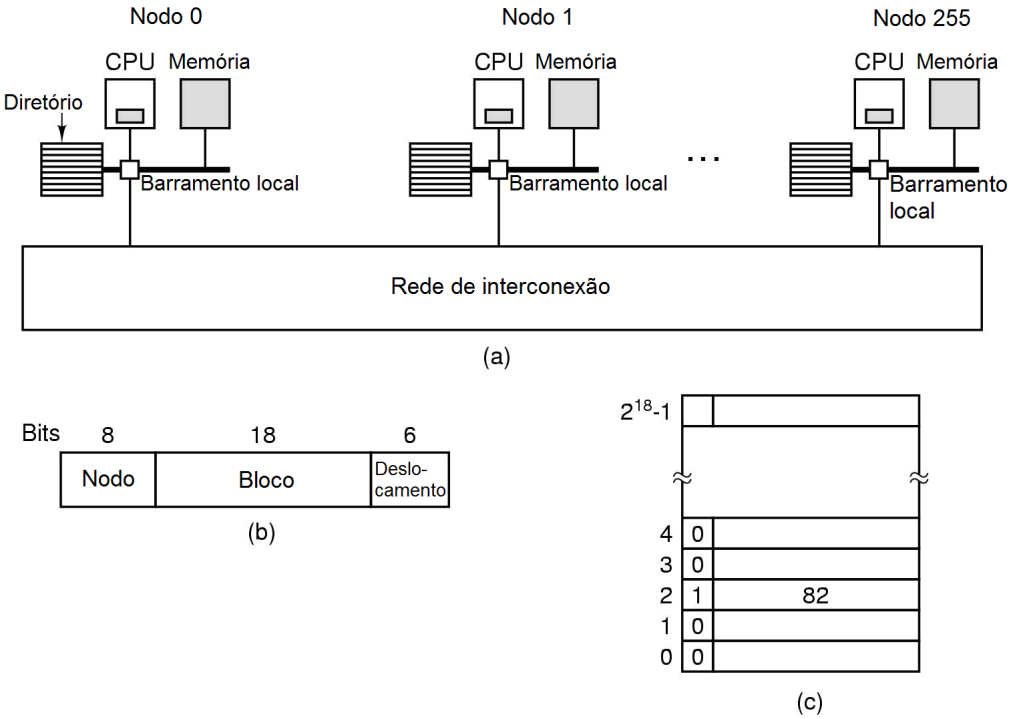

Figura 12: CC-NUMA. (a) Multiprocessador de 256 nodos com base em diretório. (b) Divisão de um endereço de memória de 32 bits em campos. (c) O diretório no nodo 36.

3.2 Multicomputadores

Os multicomputadores são CPU's fortemente acopladas, que possuem sua própria memória local, não havendo compartilhamento de uma memória como nos multiprocessadores. Basicamente, são computadores normais com uma placa de rede.

Quanto melhor essa rede de interconexão, melhor o desempenho. As topologias de interconexão podem ser vistas na Figura.13.

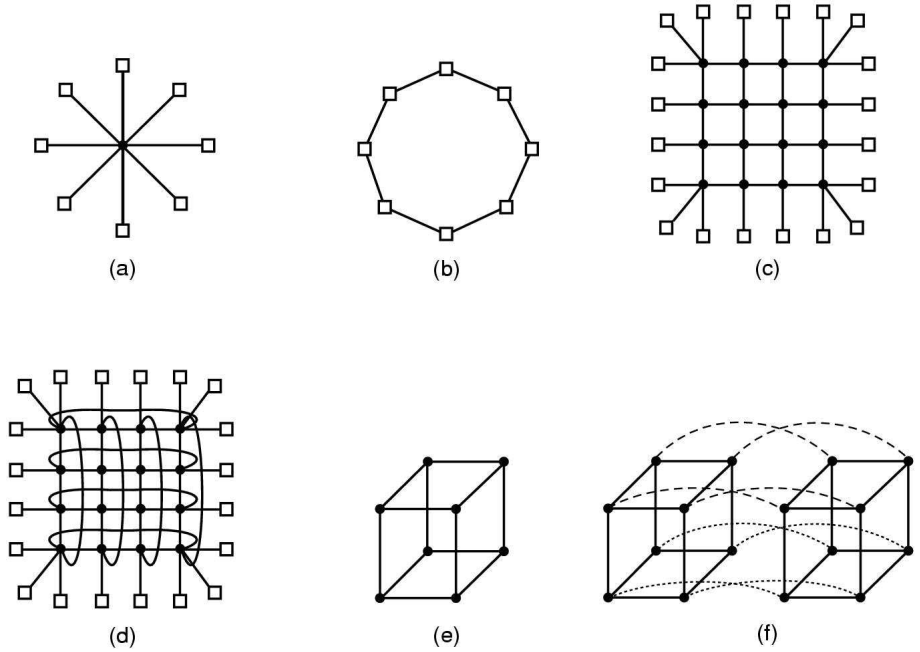

Figura 13: Várias topologias de interconexão. (a) Um *switch* apenas. (b) Um anel. (c) Uma grade. (d) Um toro duplo. (e) Um cubo. (f) Um hipercubo 4D.

Um exemplo bem conhecido é o *cluster*[4] (agregado), onde existem vários computadores trabalhando em conjunto para executar um processo de modo que o usuário tenha a impressão de que há apenas um computador trabalhando. Esses agregados proporcionam: um alto desempenho, pelo fato de vários computadores estarem executando um mesmo processo; escalabilidade, por estarem conectados a partir de uma rede; baixo custo, por serem computadores normais; e tolerantes a falhas, no caso de um nó (cada computador do cluster) parar de funcionar, outro pode assumir. Existe também a independência de fornecedores, pois são usados *hardwares* abertos e *softwares* livres. Entre os *clusters* destacam-se o *cluster Beowulf*[5] e o *COW*[1] (*Cluster of Workstation*). O cluster Beowulf era constituído de um nó mestre controlando nós escravos. Funcionavam todos em Linux e eram interligados por uma rede *Fast Ethernet*. Este *cluster* é composto basicamente de várias CPUs conectadas e controladas por um nó mestre, tendo como objetivo único o processamento em paralelo. Os *COW's*, bem parecidos com o *Beowulf*, também possuíam estações de trabalho de alto desempenho, mas cada uma com seus periféricos(monitor, teclado, mouse), ligados por uma rede *Ethernet*. Quando um nó não estava participando do processamento, poderia ser usado como posto de trabalho[16]. Um exemplo prático pode ser observado em um laboratório de informática. Este poderia ser utilizado como um clus-

ter, uma vez que as máquinas estão conectadas por uma rede fortemente acoplada e, quando não estivessem processando em paralelo, poderiam ser utilizadas pelos usuários do laboratório para qualquer outro fim. Não limitando assim, seu uso apenas para o processamento de alto desempenho, como é feito no cluster *Beowulf*.

3.3 Sistemas distribuídos

Já os sistemas distribuídos são multicomputadores fracamente acoplados, ou seja, uma "coleção"de computadores independentes que se apresentam ao usuário como um único computador[1]. Cada nó desse sistema consiste em um computador completo que pode estar em qualquer lugar do mundo e ter sistemas operacionais diferentes. Pode-se citar milhares de máquinas cooperando fracamente pela internet como um exemplo. Esses sistemas são organizados por meio do *middleware*, que é uma camada de *software* localizada entre o alto e o baixo nível para que os sistemas distribuídos possam ser uniformes na presença de diferentes sistemas operacionais e *hardwares*, como mostra a Figura.14. Dentre as vantagens, pode-se citar o baixo custo, pois o conjunto de computadores normais é mais barato que um supercomputador; o compartilhamento de recursos pela rede; o desempenho, por estarem operando juntos para um mesmo fim; a escalabilidade, pela facilidade de adicionar nós no sistema; a confiabilidade, caso um computador pare, outro pode continuar seu trabalho. O fato de serem pouco acoplados pode ser uma vantagem, pois são usados por várias aplicações, e, ao mesmo tempo uma desvantagem, pois sua programação é muito complexa. Outras desvantagens são: a comunicação exclusiva por troca de mensagens, que pode gerar erros caso as mensagens não sejam entregues; a complexidade do modelo de falhas, pela variedade de sistemas operacionais suportados; a segurança, pela possibilidade de mensagens serem interceptadas e o desempenho, pois a rede pode saturar causando lentidão.

3.4 Classificação de Flynn

Uma forma de classificar os processos computacionais em geral, criada na década de 70, e ainda muito válida e definida, é a de classificação de Flynn[17]. Esta classificação é baseada no fluxo de instruções e dados dentro do processador e é definida conforme a Tabela.2.

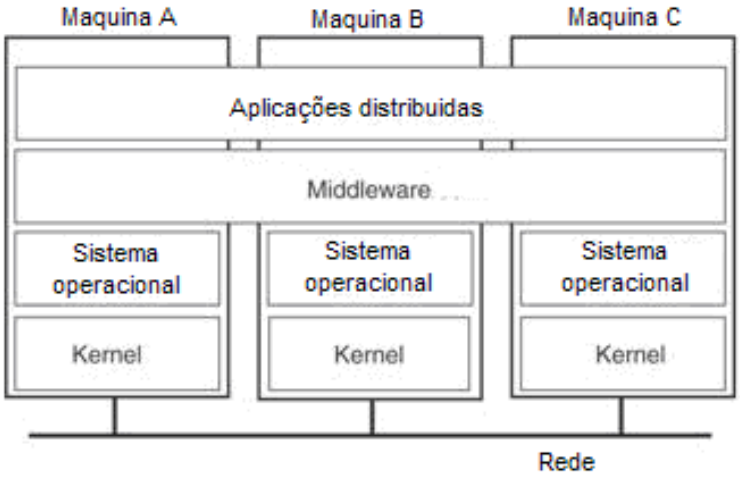

Figura 14: Posicionamento da *middleware* em um sistema distribuído.

		Fluxo de Instruções	
Fluxo de Dados		Single Instruction	Multiple Instruction
	Single Data	SISD	MISD
	Multiple Data	SIMD	MIMD

Tabela 2: Classificação de Flynn.

SISD - Single Instruction Single Data

São computadores seqüenciais que utilizam o princípio de Von Neumann[8] (processador, memória e dispositivos de entrada e saída) e que podem realizar o paralelismo virtual usando *pipeline*. É, basicamente, um único processador operando sobre um único fluxo de dados. Microcomputadores pessoais e estações de trabalho fazem parte deste grupo.

SIMD - Single Instruction Multiple Data

Pode ser definido como vários elementos de processamento controlados por uma unidade de controle, agindo sobre um mesmo bloco de dados, cada um em uma parte distinta. Pode-se citar os computadores vetoriais como exemplo.

MISD - Multiple Instruction Single Data

São várias unidades de processamento recebendo instruções diferentes sobre o mesmo conjunto de dados, ou seja, vários processadores operando sobre um único fluxo de dados. Não existe implementação desta classe.

MIMD - Multiple Instruction Multiple Data

Os processadores são controlados por uma unidade de controle e executam instruções independentemente sobre diferentes fluxos de dados. Como exemplos, pode-se citar multiprocessadores e multicomputadores.

4 INTRODUÇÃO Á PROGRAMAÇÃO PARALELA

O desenvolvimento de um algoritmo é um componente crítico para a resolução de problemas utilizando computadores. De acordo com Ananth Grama[18], existem dois tipos de algoritmos: os algoritmos seqüenciais e os algoritmos paralelos.

Um algoritmo seqüencial pode ser definido como uma lista de passos básicos com a finalidade de resolver problemas da computação serial. Similarmente, um algoritmo paralelo é um conjunto de instruções utilizadas para resolver problemas com múltiplos processadores. No entanto, desenvolver um algoritmo paralelo invoca especificamente alguns passos, como se preocupar com a concorrência e a sincronização. Essas preocupações são essenciais para obter algum ganho de desempenho.

Vários paradigmas de programação foram desenvolvidos para explicitar a programação paralela, o que difere é a maneira que elas fazem uso do espaço de memória. Assim, fica visível para o programador a forma de sincronização imposta em atividades concorrentes e a multiplicidade dos programas. Neste trabalho serão referenciados três paradigmas: *Parallel Data*[19], *Distributed Shared Memory (DSM)*[19] e *Message Passing* (Troca de mensagens)[20].

4.1 *Parallel Data* (PD)

Parallel Data é um paradigma de programação constituído em um simples método de desenvolver algoritmos com controle de distribuição de operações e dados para todos os processadores. Tipicamente suporta um *array* de operações e aceita qualquer *array* que seja usado nas expressões. A responsabilidade de distribuir o *array* de elementos, isto é, dividir o *array* em vários menores e entregar cada pedaço a um processador disponível no *cluster*, fica, portanto, a cargo do compilador assim como

a sincronização que não é feita pelo programador. Sendo assim, cada processador é responsável por uma parte do *array* e irá utilizar sua memória local para executar o processo. Uma exemplo de linguagem de programação usando *Parallel Data* é o *High Performance Fortran (HPF)*[21], que é baseado na linguagem Fortran 90.

4.2 *Distributed Shared Memory* (DSM)

O paradigma *Distributed Shared Memory* (DSM) tem um acesso presumidamente infinito ao banco de memória, ou seja, todos os processos podem referenciar pedaços da memória. As operações na memória, sejam elas remotas ou locais, são assumidas teoricamente com o custo de uma unidade, isto é, cada processador acessa a memória e executa as operações como se a memória estivesse reservada somente para o mesmo. Esta é a diferença entre os computadores normais em que o acesso à memória local é mais rápido que o acesso à memória remota. Um exemplo de uma *API* de programação que utiliza DSM é *TreadMarks*[22].

O *TreadMarks* é simples, porém poderoso. Ele provê facilidades no processo de criação, destruição, sincronização e alocação de memória compartilhada. A alocação de memória compartilhada é feita a partir do comando *tmk_malloc()*. Somente memória alocada pelo *tmk_malloc()* é compartilhada, as memórias alocadas estaticamente ou por chamada do comando *malloc()* são privativas de cada processador. O *TreadMarks* possui sincronização que é feita pelo programador, usando comandos para, de forma expressa, ordenar o acesso à memória compartilhada de diferentes processos. A simples forma de sincronização ocorre na sessão crítica, onde a mesma sessão garante que somente um processo execute em um tempo neste local.

4.3 *Message Passing (MP)*

O paradigma *Message Passing* (troca de mensagem) é o mais antigo e o mais apropriadamente usado para a programação paralela. Existem dois atributos chave que caracterizam o paradigma de *Message Passing.* O primeiro é o que assume o espaço de memória particionado, ou seja, este atributo indica que o espaço de memória utilizado pelo algoritmo pode ser particionado (o que define a memória distribuida), e o segundo é o que suporta somente paralelização explícita, ou seja, o programador deve efetuar toda a paralelização por meio de diretivas de programação paralela. A

vantagem desse tipo de paradigma de programação é que pode ser implementado facilmente sobre as mais variadas arquiteturas. Uma visão lógica de uma máquina que suporta *Message Passing* consiste em N processos, cada um com exclusividade de espaço de memória.

O paradigma *Message Passing* requer que a paralelização seja explicita para o programador, isto é, o programador é responsável por analisar o algoritmo serial, identificar como ele pode decompor a computação e extrair a concorrência. Como resultado, a programação com base neste paradigma tende a ser mais difícil e demandar muita inteligência. Por outro lado, desenvolver programas com *Message Passing* pode oferecer um desempenho muito alto e uma escalabilidade para um grande número de processos.

Os programas de *Message Passing* são escritos usando o paradigma assíncrono ou o paradigma *Loosely Synchronous* (Parcialmente Síncrono). No paradigma assíncrono todas as tarefas de concorrência são executadas aleatoriamente, possibilitando a implementação de qualquer algoritmo paralelo. Porém, alguns programas podem entrar em condição de corrida. programas que usam *Loosely Synchronous* possuem um compromisso entre esses dois extremos. Os programas possuem tarefas síncronas e iterações entre si, sendo essas iterações executam tarefas completamente assíncronas. Muitos sabem que algoritmos paralelos podem ser naturalmente implementados utilizando o *Loosely Synchronous*. De uma forma geral, o paradigma Message Passing suporta execução de diferentes programas, sendo um para cada processo. Dentro deste paradigma existem duas bibliotecas: PVM (*Parallel Virtual Machine*)[23] e MPI (*Message Passing Interface*)[23].

4.3.1 PVM - *Parallel Virtual Machine*

PVM é uma ferramenta muito potente para explorar virtualmente qualquer rede de computadores ou computadores paralelos. Com esta ferramenta, o usuário pode criar e controlar uma máquina paralela virtual de qualquer computador conectado na rede, sendo assim é possível rodar um numero de *jobs* em paralelo em uma simples máquina, na rede ou em uma máquina paralela. Ao contrário da MPI (será explicada mais adiante), PVM é muito mais do que uma interface de troca de mensagem. Com ela é possível emular um sistema paralelo e debugar programas paralelos sobre um simples processador. A interface é identificada virtualmente em todas as máquinas, possibilitando adição dinamica de processos ou *jobs*. Resumindo, a habilidade de

rodar programas paralelos sobre esta ferramenta garante um aumento na flexibilidade durante o desenvolvimento do programa.

4.3.2 MPI - *Message Passing Interface*

Um grande problema da *Message Passing* é que, se for necessário trocar o programa de uma biblioteca para outra, será necessária a reestruturação de todo o código. A *Message Passing Interface*, ou MPI como é popularmente conhecida, foi criada essencialmente para resolver esse problema. Essa *interface* define um padrão de biblioteca para *Message Passing* que pode ser usado para desenvolver programas portáteis entre as bibliotecas de *Message Passing* usando C ou *Fortran*. A MPI foi desenvolvida por um grupo de pesquisadores de universidades e indústrias, e é utilizada como suporte para os mais diversos fornecedores de *hardware*. Essa biblioteca contém mais de 125 (cento e vinte e cinco) rotinas, mas o número de conceitos chaves é muito pequeno. De fato é possível escrever um programa de *Message Passing* a partir de 6 (seis) rotinas, como mostra a Tabela.3

MPI_Init	Inicializa a MPI.
MPI_Finalize	Termina a MPI.
MPI_Comm_size MPI_Comm_rank	Determina o número de processos. Determina o rótulo para chamar os processos.
MPI_Send	Envia uma mensagem.
MPI_Recv	Recebe uma mensagem.

Tabela 3: Rotinas básicas para escrever um programa. MPI

Estas rotinas são usadas para inicializar ou terminar a biblioteca, obter informações sobre os processos de computação paralela, além de enviar e receber mensagens. A maior observação que pode ser notada sobre MPI é que “a *Message Passing Interface* não é completamente um ambiente de programação para computação paralela”[24]. Isto não é um problema se for trabalhar somente com uma máquina, mas pode se tornar um problema se forem utilizados vários sistemas.

Cada implementação de MPI requer diferentes especificações de ambiente, diferentes procedimentos de submissão e concorrência de *jobs*. Sobre estas implementações, destacam-se a MPICH[6] e *LAM MPI*[6].

De acordo com um experimento realizado por Daniel Cleary[6], que examinou o tempo requerido para fazer a chamada de mensagens na programação paralela em

ambientes de computação distribuída, as implementações MPICH e *LAM MPI* possuem essencialmente os mesmos métodos e funções, mas o pouco que as diferencia é que a administração, o processamento e o tempo da mensagem do programa podem ser melhorados pela escolha de uma implementação ao invés da outra. Para isto basta que o programador e o administrador tenham conhecimento dos pontos fortes e fracos de cada implementação, escolhendo a melhor biblioteca baseado em suas necessidades.

Para ambas as bibliotecas citadas no parágrafo anterior, o experimento mediu as funções em tempo de execução nos ambientes de *clusters*, através de métodos de comunicação MPI ponto a ponto e global. Em uma comparação direta, nenhuma biblioteca se mostrou dominante em qualquer categoria, mas ambas possuem diferentes vantagens na programação. Daniel concluiu também que cada biblioteca possui benefícios significativos e áreas em que a transferência de dados é mais rápida. Uma notável observação deve ser levada em consideração: em alguns casos, a função *MPI_REDUCE* em *LAM MPI* foi até quatro vezes mais rápida que a respectiva função pertencente à implementação MPICH.

5 ALGORITMO PARALELO

Para demonstrar os fatos citados anteriormente, foi desenvolvido um algoritmo baseado no *mergesort* que irá trabalhar com várias máquinas em paralelo. Este algoritmo foi codificado com base na ferramenta MPI, uma ferramenta especial para este tipo de operação e pensado de uma forma a diminuir uma possível utilização da memória externa, esperando assim um melhor desempenho.

Primeiramente será apresentada uma descrição deste algoritmo e as ferramentas utilizadas. Em seguida, será conhecido o ambiente de testes, um *cluster Beowulf*. Logo após, os testes são feitos, nos quais serão dispostos os resultados das ordenações feitas em comparação com outros dois algoritmos, implementados especialmente para o experimento, e dados de tamanhos variados. Além disso, serão apresentadas formas de análise dos resultados, como *speedup*, eficiência e tempo de processamento com seus respectivos gráficos para cada caso. Por fim, uma conclusão baseada nos resultados.

5.1 Definição

O algoritmo desenvolvido é chamado de *mergesort* paralelo, pelo fato de utilizar a função *merge*(junção) como principal característica de funcionamento. Foi codificado a partir da ferramenta MPI, que dá suporte ao trabalho em paralelo. Seu funcionamento pode ser descrito pelo fluxograma da Figura.17.

É interessante observar a interação feita entre as máquinas pela ferramenta MPI. Cada máquina recebe um nome, e a partir deste, se sabe qual parte do arquivo ficará sob sua responsabilidade no decorrer do processo. O nome do arquivo é gerado no formato file"nome_da_máquina_responsável".dat, por exemplo, file0.dat, sendo 0 o nome dado pela ferramenta MPI a um micro no processo, como pode ser observado na Figura.15. Durante este processo, as máquinas ordenam seus arquivos e as que

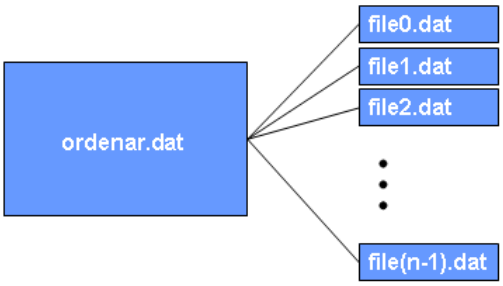

Figura 15: Divisão do arquivo a ordenar para as máquinas envolvidas no processo.

são selecionadas para enviarem seus arquivos terminam seu trabalho, enquanto as que recebem continuam, este processo pode ser verificano na Figura.16.

Para efeito de comparação, foram implementados outros dois algoritmos. O primeiro deles é o *mergesort* seqüencial, feito para dados que utilizam basicamente a memória interna, e o outro é o *mergesort* seqüencial com memória externa, onde é definido um tamanho de memória e, a partir dele, o algoritmo trata os acessos a disco.

O *mergesort* seqüencial é o algoritmo convencional feito de forma iterativa ao invés de recursiva, pois, ao tratar de arquivos grandes, o sistema operacional não consegue suportar tantas chamadas à mesma função. A partir de um certo momento, o sistema operacional não consegue controlar as chamadas recursivas e acaba perdendo a referência destas chamadas, não sendo possível a obtenção do resultado. O tamanho do arquivo foi limitado à 1GB pois o algoritmo requer que o mesmo esteja completamente alocado na memória. Foram realizadas tentativas de alocar arquivos maiores de 2GB e não foi obtido sucesso. O tamanho máximo conseguido para alocação foi de 2GB - 1*byte*. O seu pseudo-código pode ser visto na Figura.2.

O *mergesort* seqüencial com memória externa foi baseado no livro de Donald Knuth, um dos maiores nomes da computação atual, *The Art of Computer Programming*[7]. Ele controla os acessos à disco, deixando o processador menos ocioso. Seu pseudo-código foi descrito no capítulo de algoritmos de ordenação, em ordenação em memória externa.

Os algoritmos foram desenvolvidos na linguagem C, em ambiente Linux, a partir do Fedora Core 9, kernel 2.6, onde o compilador utilizado foi o gcc em sua versão 4.3.0-8.i386. A distribuição Linux foi instalada no *VMWare Workstation* 6.0.4 build-93057, uma máquina virtual, em um computador de configuração Core 2 Duo 2.33, 2GB de *RAM*, 250GB de HD, placa de vídeo 512MB 256bits, mas foi limitado ao uso de 1 (um) único processador, com 1GB de *RAM* e 15GB de HD. A ferramenta utilizada

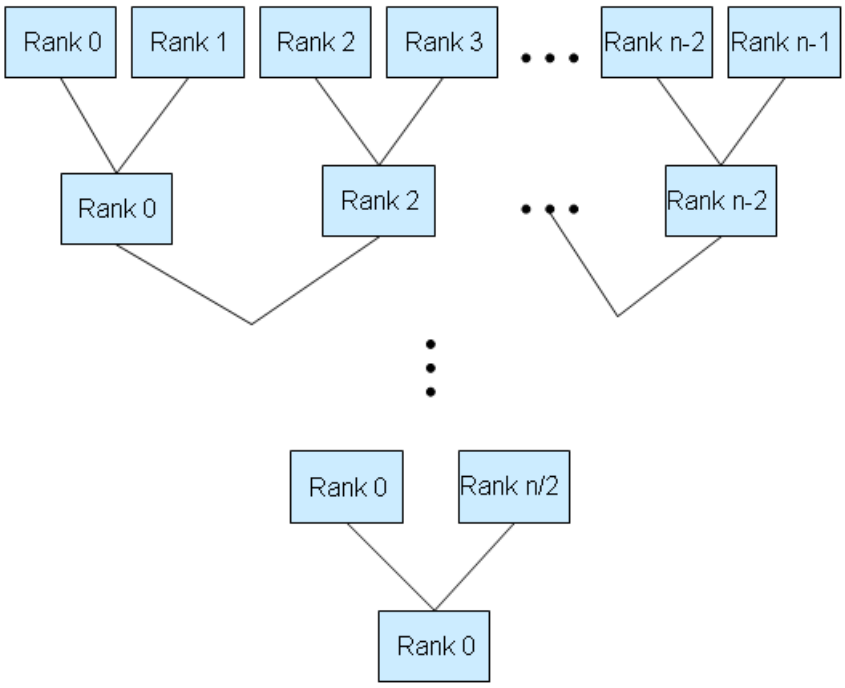

Figura 16: Funcionamento do algoritmo *mergesort*.

para a programação em modo paralelo foi o *LAM/MPI* 7.1.4. O código do programa desenvolvido se encontra no anexo A deste trabalho.

5.2 *Clusters*

Todos os testes foram realizados no cluster Enterprise do Laboratório de Computação de Alto Desempenho do Centro Tecnológico da Universidade Federal do Espirito Santo (LCAD - CT - UFES). O cluster é composto por 64 nós de processamento e um servidor. Todos os nós utilizam o sistema operacional Linux Fedora (kernel 2.4-20). Em relação as ferramentas de programação disponíveis no Enterprise, foram utilizadas o compilador gcc-2.96-112.7.1 da linguagem C para GNU/Linux, parte da distribuição Fedora e a biblioteca do padrão *MPI lam*- 6.5.9-tcp.1. Os códigos sequenciais foram compilados utilizando “-o file file.c ”e os códigos paralelos “mpicc -o file file.c ”.

A rede de interconexão é composta basicamente por dois *switches* 3COM modelo 68 4300. Cada *switch* possui 48 portas e cada porta tem uma capacitadade teórica de 100Mb/s para comunicação com os nós de processamento e são interligados entre si através de um módulo *Gibabit Ethernet* (1000Mb/s). Para uma melhor utilização, a quantidade de nós de processamento é dividida igualmente entre os dois *switches* e

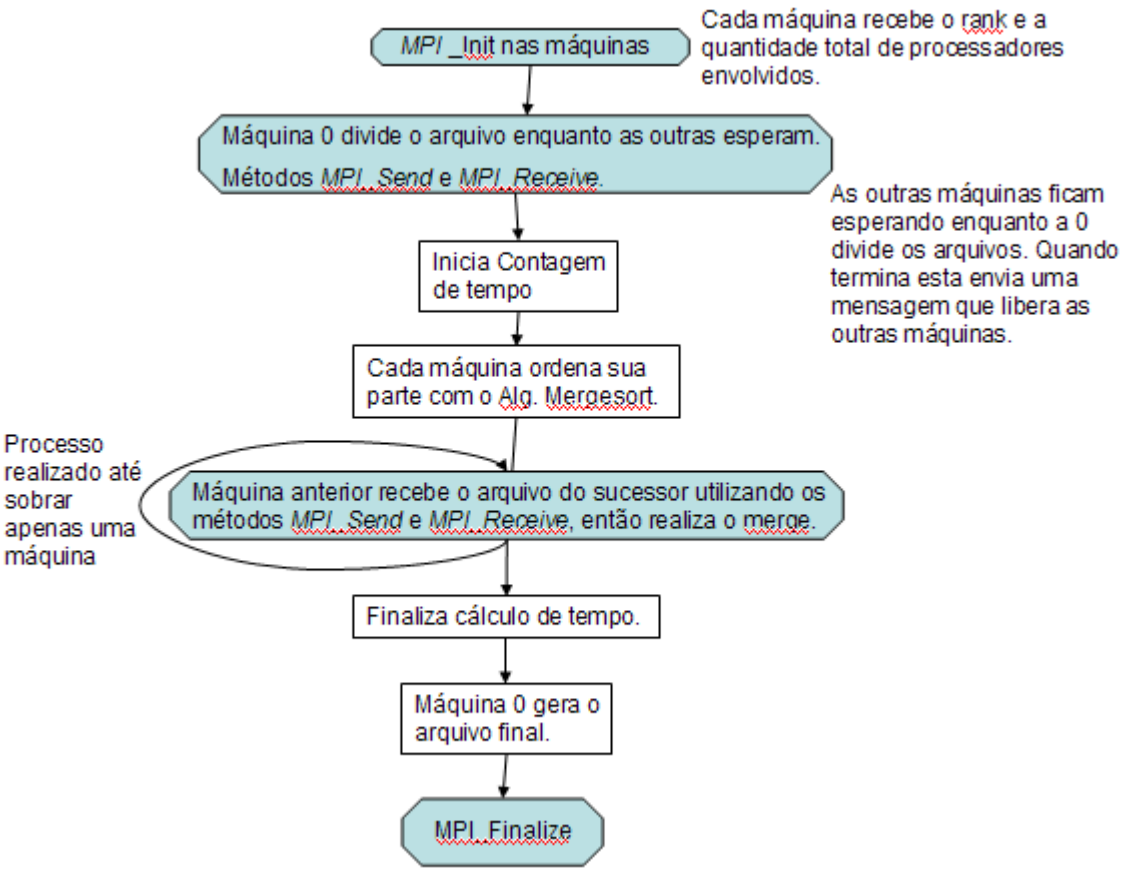

Figura 17: Fluxograma de funcionamento do *mergesort* paralelo.

um deles possui uma porta *Gibabit* a mais para conexão com o servidor.

Cada um dos 64 nós de processamento do *Enterprise* possui memoria *SDRAM* de 256MB e disco rígido Ultra *ATA* de 20GB. O processador de cada nó é o *ATHLON* XP 1800+ fabricado pela *AMD*. Visto que cada processador possui duas unidades de ponto flutuante e frequência de operação de 1,53 GHz, tem-se um desempenho teórico de pico igual a 2 x 1,53 = 3,06 GFLOP/s por nó de processamento.

A máquina servidora possui praticamente a mesma configuração dos nós de processamento, exceto que conta com 512MB de memória *RAM*, 80GB de disco e duas placas de rede, uma *3COM Gigabit Ethernet* para conexão com um dos *switches* do *cluster* e uma *3COM Fast-Ethernet* para conexão externa. Esta máquina também é responsável pela distribuição dos processos nos nós de processamento (a gerência de filas de processos é feita pelo *SGE - Sun Grid Engine* [25]), pelo armazenamento das contas dos usuários, além das atividades de confuguração, atualização e monitoramento dos nós de processamento. A Figura.18 ilustra as conexões de rede entre as máquinas externas, o servidor e os nós de processamento.

Para medir o desempenho das implementações das estratégias de armazenamento executadas no *cluster Enterprise*, foram utilizadas algumas medidas de desempenho. Na seção seguinte é detalhado como tais medidas foram delidas e como foram utilizadas.

5.3 Medidas de desempenho adotadas

O desempenho dos algoritmos desenvolvidos foi avaliado através de três medidas: tempo de processamento, *speedup* e eficiência. O tempo de processamento foi medido com o auxílio de algumas funções padrão da linguagem C, que utilizam a biblioteca "*time*.h". Quando o processo começa sua execução, a função *time* inicializa uma variável com um formato de tempo especial, formato "*time* t"e antes que a função de finalização do MPI seja acionada, uma segunda variável é marcada com o mesmo formato de tempo da variável inicial. Depois disso, é chamada a função "*difftime*"que calcula a diferença entre essas duas variáveis e retorna uma resposta em segundos. Assim, o tempo de execução total é calculado e escrito em um arquivo.

O *speedup* tem como objetivo calcular quantas vezes um processo executado com n processadores envolvidos é melhor que o executado em somente um. O *speedup*

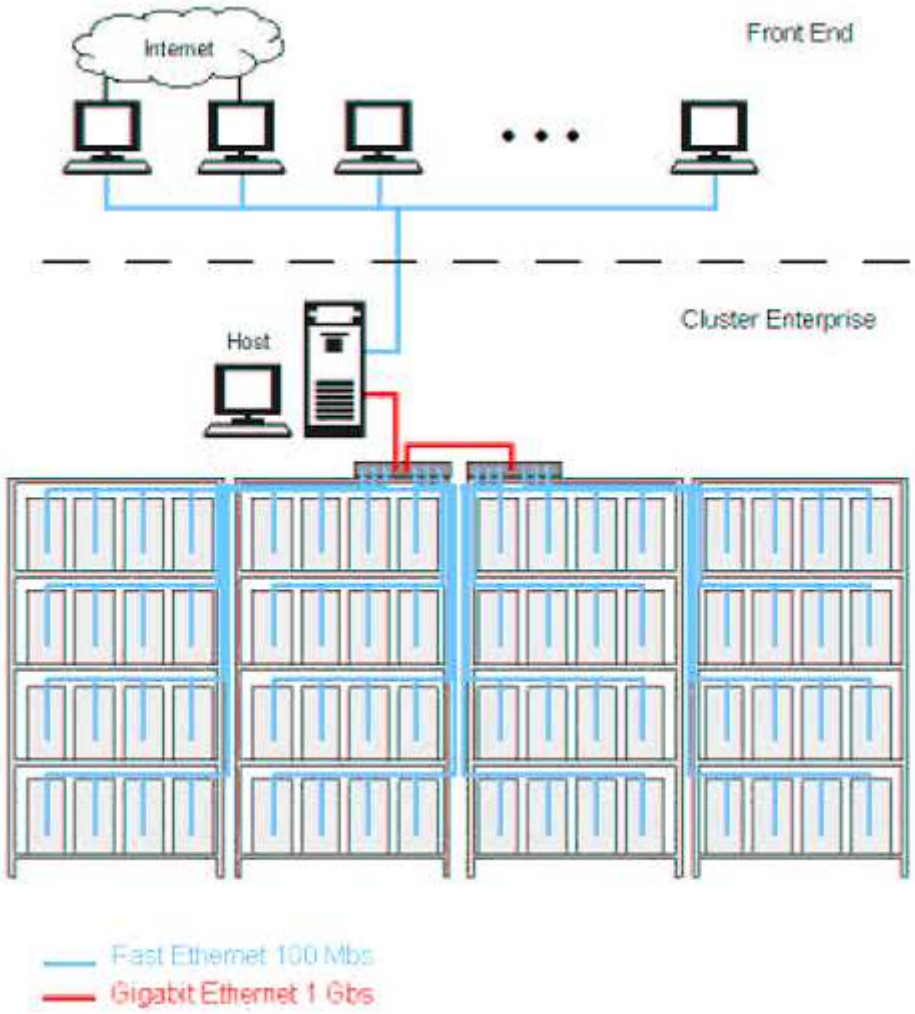

Figura 18: Conexões de rede do cluster Enterprise

ideal é sempre n, sendo um processo executado com estes n processadores, n vezes mais rápido que o mesmo processo executado em um microcomputador somente. A eficiência é a porcentagem do *speedup*. Se o mesmo for o ideal, então a eficiência é de 100%.

O *speedup* (S_i) e a eficiência (E_i) foram definidos conforme as expressões

$$S_i = \frac{t_1}{t_i} \tag{5.1}$$

$$E_i = \frac{S_i}{i} \tag{5.2}$$

onde t1 representa o tempo de processamento do algoritmo sequencial implementado e ti, com i = 1; ...; p, o tempo de processamento do algoritmo paralelo executado utilizando i processadores, sendo p o número de processadores envolvidos no processamento paralelo.

5.4 Testes

Esta seção apresentará testes de desempenho realizados no *cluster*. Sua finalidade é verificar, em diferentes parâmetros, qual algoritmo levaria menos tempo para ordenar os dados. Os arquivos foram gerados em formato binário, pois cada número no arquivo ocupa 4 *bytes*, pelo fato de serem do tipo inteiro. Caso fosse utilizado arquivo texto, o elemento 1000 não teria o mesmo tamanho, em *bytes*, que o elemento 1 dentro deste arquivo. Estes elementos foram gerados de forma aleatória, de 0 à 9999. Os parâmetros analisados foram os seguintes:

Tamanho do arquivo	Quantidade de elementos
32MB	(2^{23})
64MB	(2^{24})
128MB	(2^{25})
256MB	(2^{26})
512MB	(2^{27})
1024MB	(2^{28})

Tabela 4: Tamanho dos arquivos a serem ordenados.

Os algoritmos utilizados nos testes foram o *mergesort* seqüencial, *mergesort* seqüencial com memória externa (memória principal limitada a 128MB), *mergesort* paralelo

operando com 2 processadores, *mergesort* paralelo operando com 4 processadores, *mergesort* paralelo operando com 8 processadores e o *mergesort* paralelo operando com 16 processadores.

A partir destes parâmetros, foi feito um teste para cada combinação possível. Os resultados foram medidos na ordem de segundos, a fim de facilitar a comparação entre os números obtidos. Segue abaixo uma tabela para cada caso de teste:

5.4.1 Algoritmos x 32MB de elementos

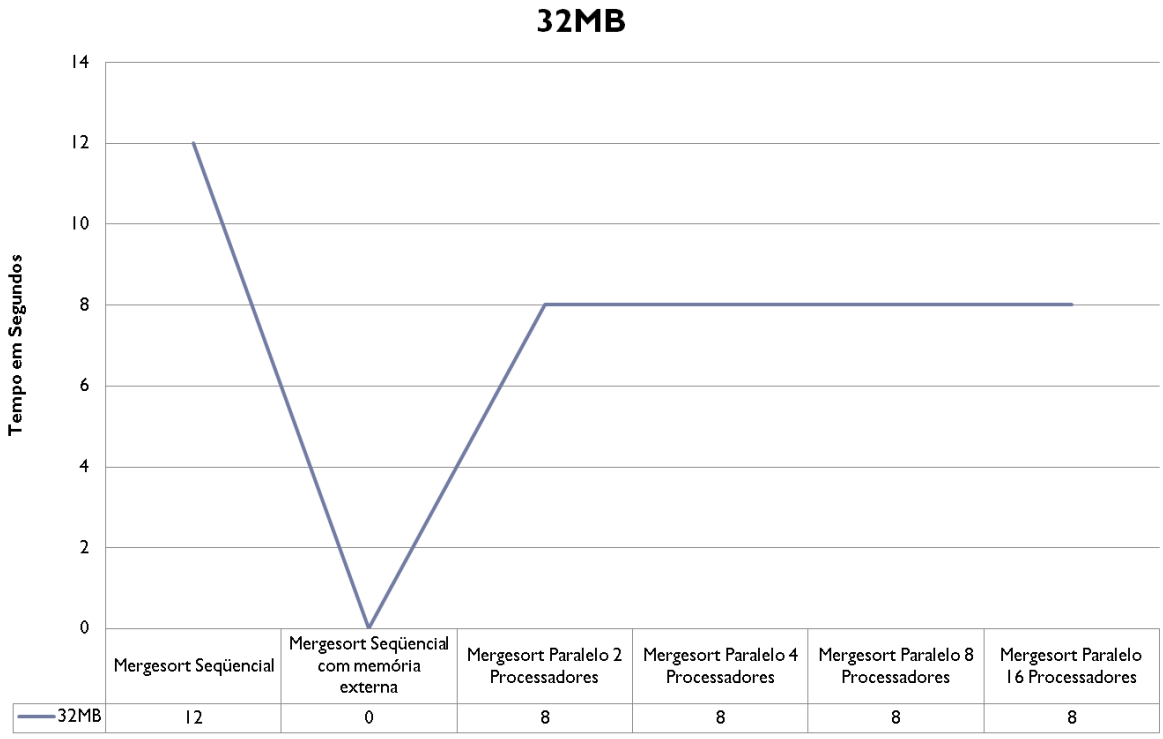

Figura 19: Tempo dos algoritmos, em segundos, ordenando 32MB de dados.

Como pode ser observado na Figura.19, o *mergesort* paralelo obteve 8 segundos em todos os seus casos. O número de processadores não interferiu no resultado pelo fato do arquivo ser muito pequeno. Ao se pensar no fato de 16 processadores executarem no mesmo tempo que 2, o instante entre a troca de mensagens dos 16 computadores pela rede, neste caso, compensou o de processamento de apenas duas máquinas. Para ficar mais claro, enquanto duas máquinas processam seus arquivos de 16MB e realizam apenas um envio e recebimento pela rede, 16 máquinas processam arquivos de 1MB e realizam 15 trocas de mensagem, onde o tempo destas trocas, compensaria o processamento de mais dados por um único computador com

menos comunicação.

O *mergesort* seqüencial com memória externa não seria aplicável neste caso de teste pelo fato de sua memória ser limitada a 128MB de dados e o arquivo a ordenar ter apenas 32MB. O algoritmo não teria efeito, pois é feito especialmente para tratar o acesso à disco, ou seja, deve-se ter um arquivo maior ou pelo menos igual ao tamanho da memória para realizar o processo. Obviamente, levando em conta que esta memória de 128MB seria reservada apenas para os dados do algoritmo. Uma linha do algoritmo, vista abaixo, pode explicar o fato da impossibilidade de sua utilização neste caso.

```
92//n_arquivos = número de arquivos.
93//tam_arq_ini = tamanho do arquivo a ordenar.
94//tam_mem = tamanho da memória.
95    n_arquivos=tam_arq_ini/tam_mem;
```

Figura 20: Parte do código do algoritmo *mergesort* seqüencial com memória externa.

Suponha o caso da Figura.20, "tam_arq_ini"= 32MB e "tam_mem"= 128MB. O número de arquivos seria 0,25. Este número de arquivos é utilizado para alocar vetores úteis no decorrer no algoritmo. Sendo assim, não seria possível alocar um vetor menor ou igual a 1 (um).

No caso do *mergesort* seqüencial, o resultado foi 4 segundos a mais que os outros algoritmos, exceto o *mergesort* seqüencial com memória externa que não se aplica. Ao se pensar no fato de utilizar várias máquinas ao invés de somente uma, é preferível a utilização, neste caso, do seqüencial do que o paralelo, mesmo que o tempo de processamento seja maior. O trabalho e o custo de se montar um *cluster* não compensaria o investimento.

5.4.2 Algoritmos x 64MB de elementos

Neste teste, obteve-se praticamente o mesmo resultado anterior, a não ser pelo *mergesort* paralelo com 2 processadores, que levou 3 segundos a mais que os com mais processadores envolvidos. O processamento foi mais lento que a troca de mensagens, não foi compensado como visto anteriormente.

O algoritmo *mergesort* seqüencial com memória externa não se aplica a este teste, pois o tamanho do arquivo é menor que a memória definida anteriormente, de 128MB.

O *mergesort* seqüencial continua sendo o melhor em termos de custo x benefício,

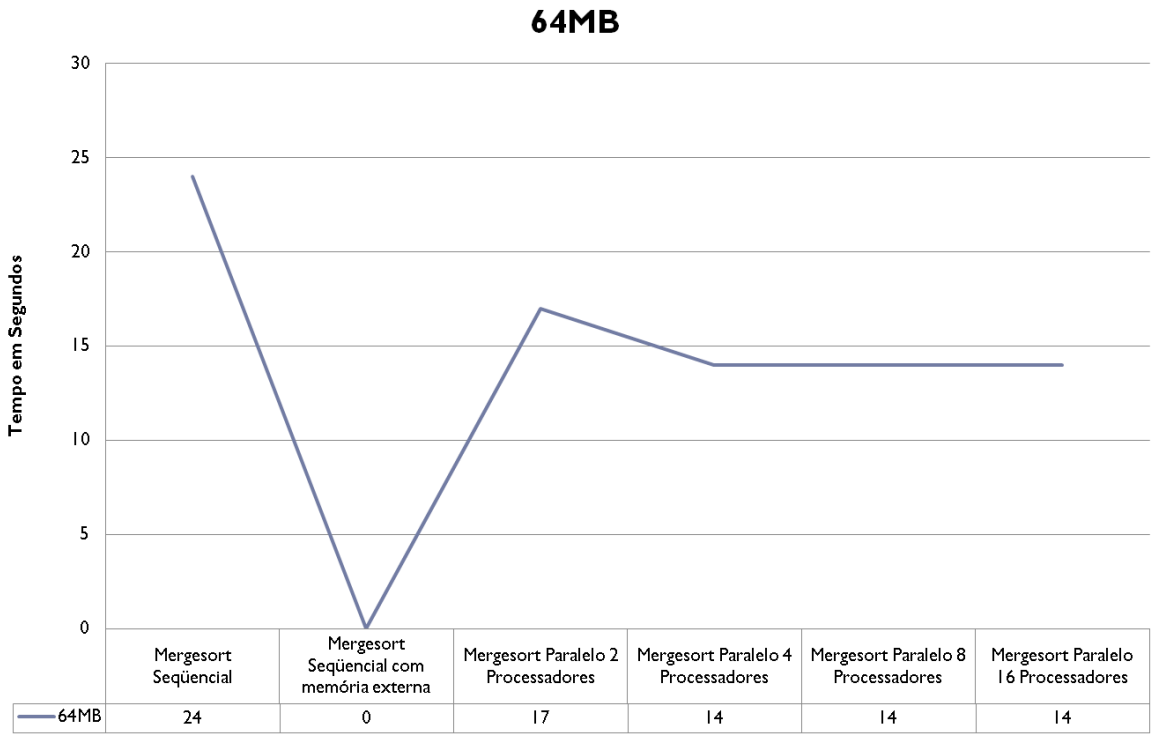

Figura 21: Tempo dos algoritmos, em segundos, ordenando 64MB de dados.

pois a diferença entre os tempo, neste caso, não é tão grande.

5.4.3 Algoritmos x 128MB de elementos

Este caso já é mais interessante de se analisar. Como era de se esperar, quanto mais processadores, melhor o resultado obtido. Esta afirmação é verdadeira neste teste. O *mergesort* paralelo com 16 processadores superou o com 8, que superou o 4, e que também foi melhor que o com 2.

O pior resultado obtido foi o do *mergesort* seqüencial com memória externa, que agora é aplicável, pois o arquivo é de tamanho igual ou maior à sua memória. O fato de ele tratar acessos a disco fez com que ele demorasse mais que os demais. Neste caso, ele não deveria se preocupar com a paginação, o arquivo cabe todo na memória principal, mas é do princípio do algoritmo.

O *mergesort* seqüencial teve um desempenho melhor que o com memória externa justamente pela razão citada acima. Como ele é feito para memória interna, não se preocupa com o acesso ao disco e o tamanho do arquivo cabe totalmente na memória principal, pois as máquinas de teste possuem 256MB de *RAM*.

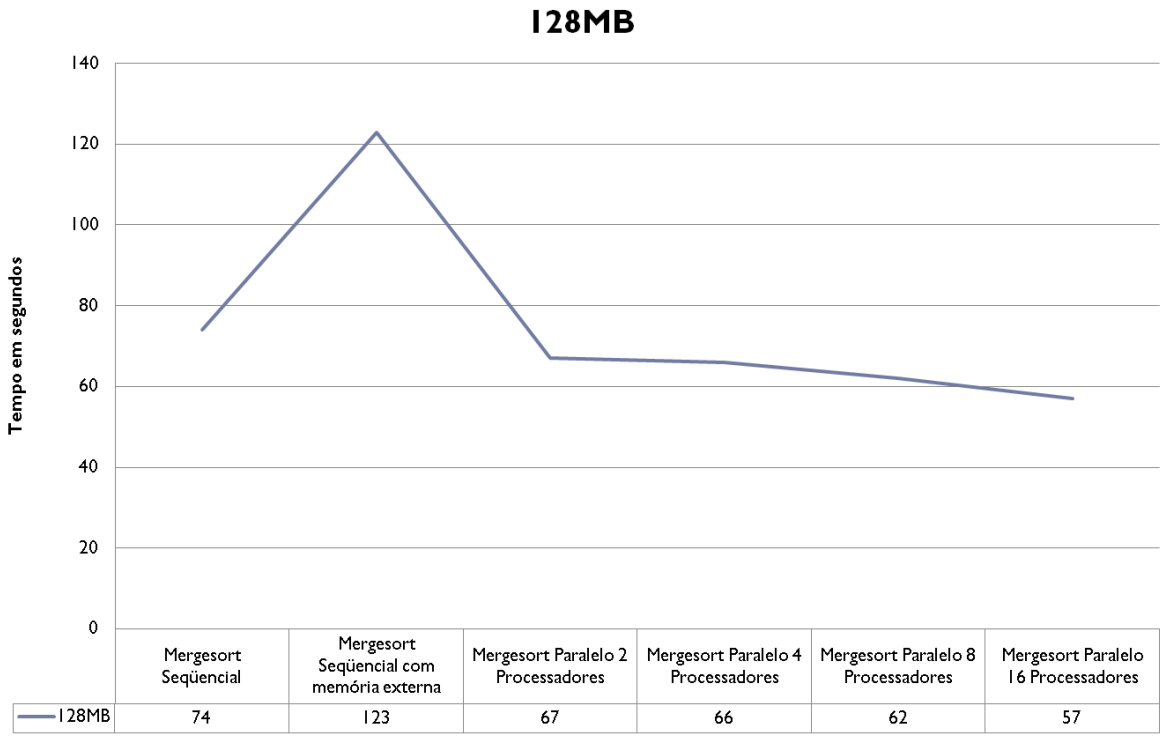

Figura 22: Tempo dos algoritmos, em segundos, ordenando 128MB de dados.

5.4.4 Algoritmos x 256MB de elementos

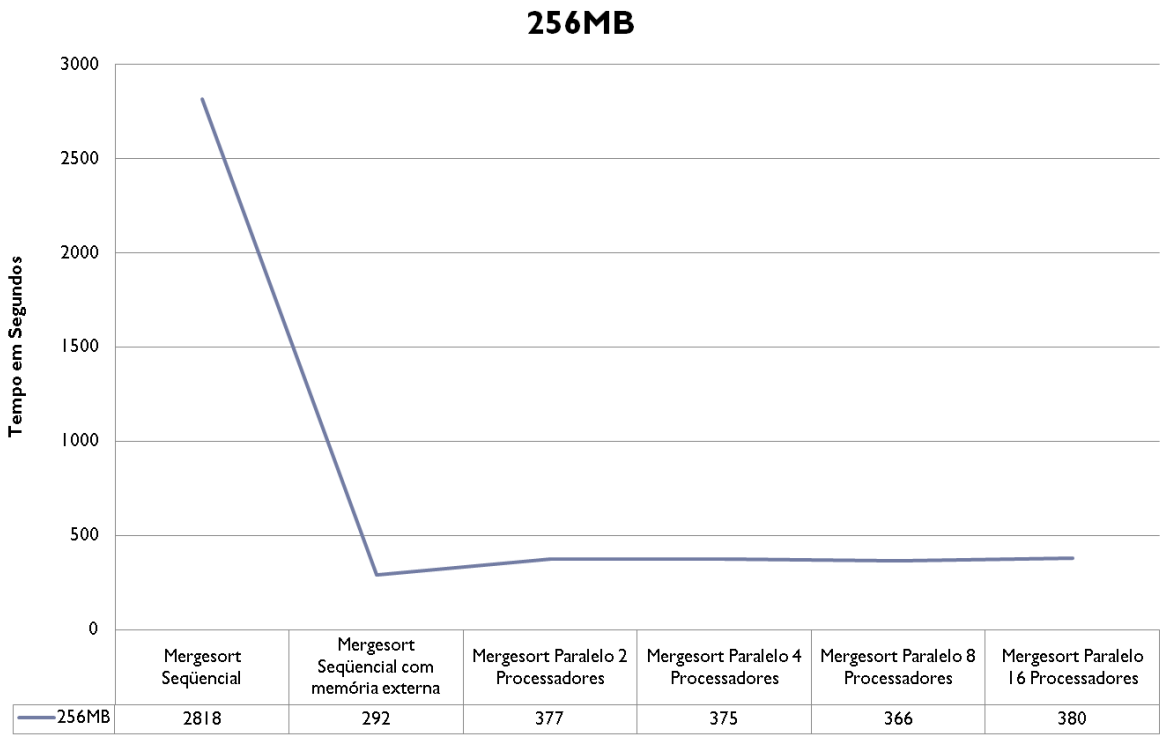

Figura 23: Tempo dos algoritmos, em segundos, ordenando 256MB de dados.

A partir da Figura.23, pode-se verificar um fato surpreendente. Um algoritmo seqüencial sendo mais rápido que um paralelo. Para muitos, isso é uma contradição. Foi mostrado que não é. Tendo em vista que o algoritmo seqüencial com memória externa conseguiu um resultado melhor que os demais testados, comprova-se que não há contradição.

A diferença entre os métodos seqüenciais passa a ser considerável. Mas, desta vez, o algoritmo em memória externa foi melhor que o seqüencial: o resultado contrário ao teste anterior. A característica de tratamento de acessos a memória secundária mostrou-se muito eficaz até mesmo comparada ao algoritmo em paralelo, em todos os seus casos.

Houve muita igualdade entre os casos do algoritmo paralelo, se comparados entre si, neste caso. Os tempos foram muito próximos e o aumento das máquinas não foi significativo.

Comparando o paralelo com o seqüencial com memória externa a diferença não foi tão grande, mas ao se tratar de desempenho, foi bastante considerável. A rede, provavelmente, foi o gargalo da operação, atrasando seu processo de ordenação.

5.4.5 Algoritmos x 512MB de elementos

O teste da Figura.24 mostra um resultado muito semelhando ao comentado anteriormente, a não ser pela igualdade entre os algoritmos em paralelo. O com 2 processadores foi consideravelmente mais custoso que os demais.

O algoritmo seqüencial comum tende somente a piorar seu desempenho, a medida que o arquivo a ordenar aumenta.

O *mergesort* seqüencial com memória externa se portou tão bem neste caso, que praticamente dava pra ordenar o arquivos duas vezes, enquanto máquinas em paralelo ordenariam uma única vez.

5.4.6 Algoritmos x 1024MB de elementos

A Figura.25 define o último teste realizado neste trabalho, a comparação entre algoritmos e um arquivo com 1024MB de dados. O *mergesort* seqüencial rodou por mais de 9 horas sem retornar resultado algum e foi abortado, pois o objetivo não é

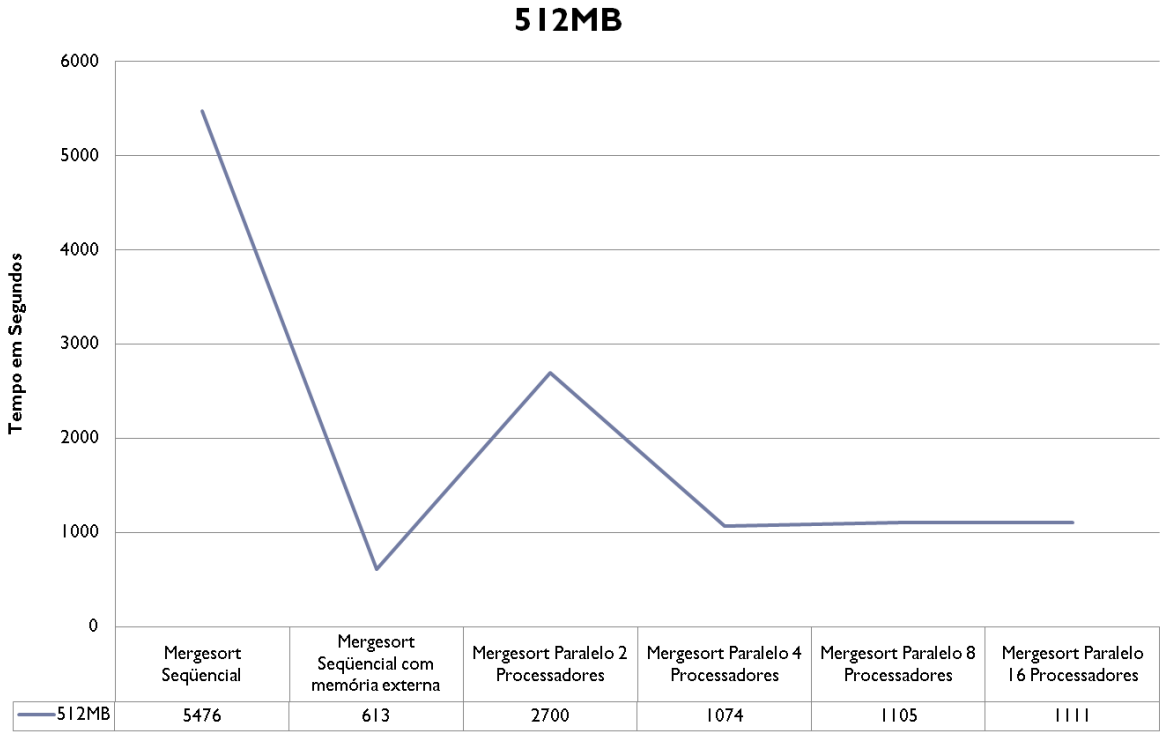

Figura 24: Tempo dos algoritmos, em segundos, ordenando 512MB de dados.

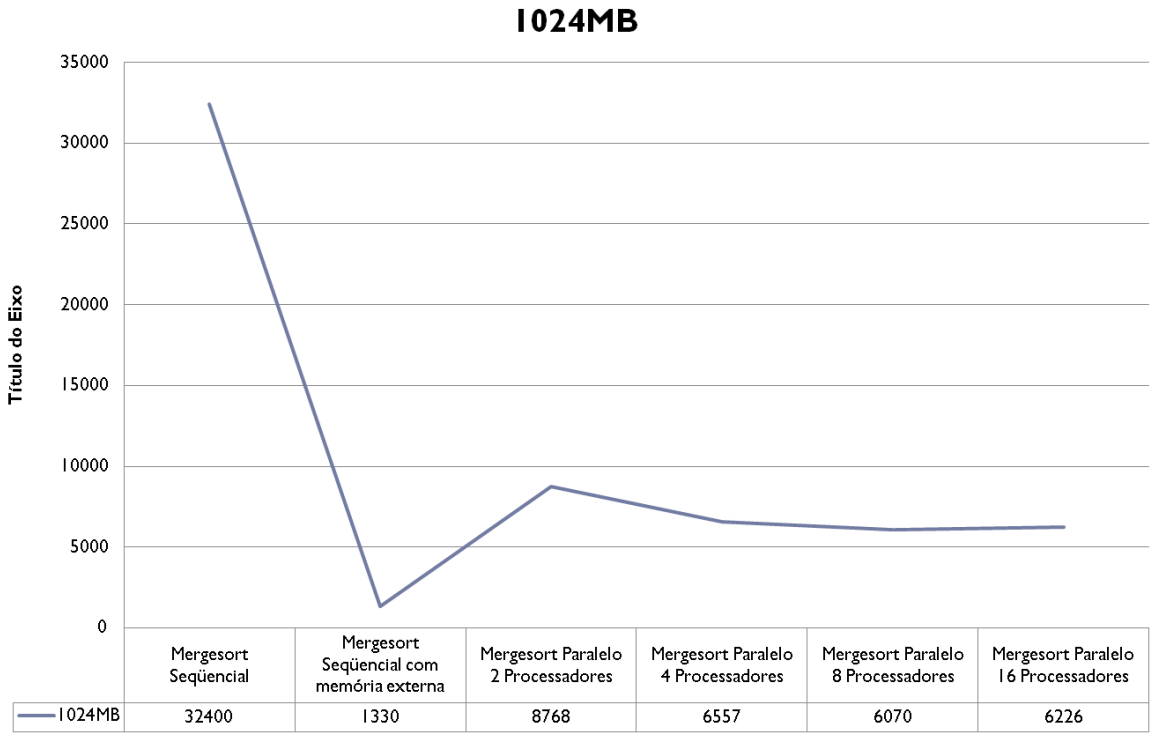

Figura 25: Tempo dos algoritmos, em segundos, ordenando 1024MB de dados.

medir o tempo real que o algoritmo leva, e sim compará-los entre si.

Os algoritmos em paralelo rodaram por mais de uma hora, para obterem um resultado. Mais uma vez a diferença, no contexto, não foi muito grande entre os que utilizam 4, 8 e 16 processadores. Já o com 2 processadores, sim.

O algoritmo *mergesort* seqüencial com memória externa obteve, novamente, o melhor resultado entre os testes. Desta vez a diferença foi tão grande que ele rodaria por quase 5 vezes enquanto o melhor dos paralelos ordenaria uma.

5.4.7 *Speedup* e Eficiência

De acordo com os dados que são mostrados na Figura.25, o tempo do *mergesort* seqüencial com memória externa foi de 1330 e o *mergesort* paralelo com 8 processadores foi de 6070, obtendo um *speedup* de, aproximadamente, 0,22. Sendo assim, o *speedup* e a eficiência do *mergesort* paralelo comparado ao *mergesort* seqüencial com memória externa não foram satisfatórios, tendo em vista que o valor esperado do *speedup* era de 8, uma vez que o processamento de 8 máquinas deveria ser maior 8 vezes mais rápido que o de uma.

Comparado o *mergesort* seqüencial com o *mergesort* paralelo, obtemos um bom speedup e uma boa eficiência. Como se pode ver na Figura.26, o speedup que mais vezes se aproximou ou foi maior que o ideal, foi na ordenação de arquivos com 256MB de dados. Isso ocorre porque as máquinas possuiam 256MB de memória e o algoritmo não trata os acessos à disco, sendo assim, a ordenação de 256MB de dados com o algoritmo de até 8 processadores fez nenhum ou um acesso à disco tornado o speedup melhor ou o ideal. Na Figura.27, pode-se verificar qual foi a eficiência dos algoritmos testados.

5.4.8 Limitações do Algoritmo Paralelo

Algumas limitações puderam ser observadas com os testes realizados. A principal delas seria em relação aos acessos a disco. Uma possível solução seria tentar reduzir o número de acessos a disco em cada processo. Supondo que 1 arquivo de 1GB de dados a ordenar, realize 100 acessos a disco em cada nível do algoritmo *mergesort* (Ver Figura.28 para melhor entendimento).

Imaginando também este arquivo sendo ordenado paralelamente em 10 máquinas.

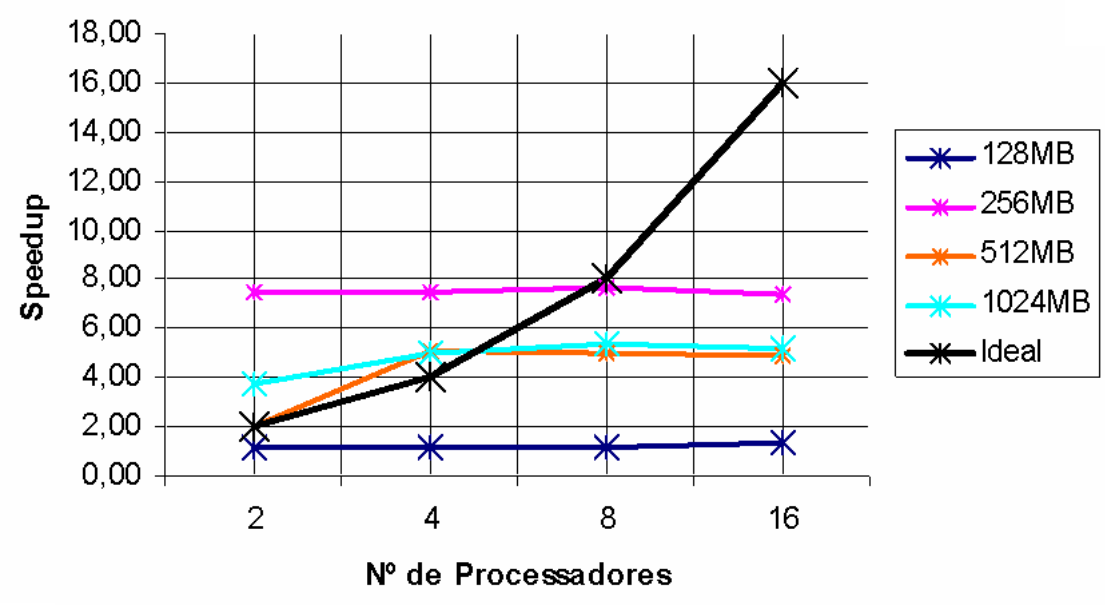

Figura 26: Grafico de *Speedup* - *MergeSort* Seqüencial X *MergeSort* Paralelo.

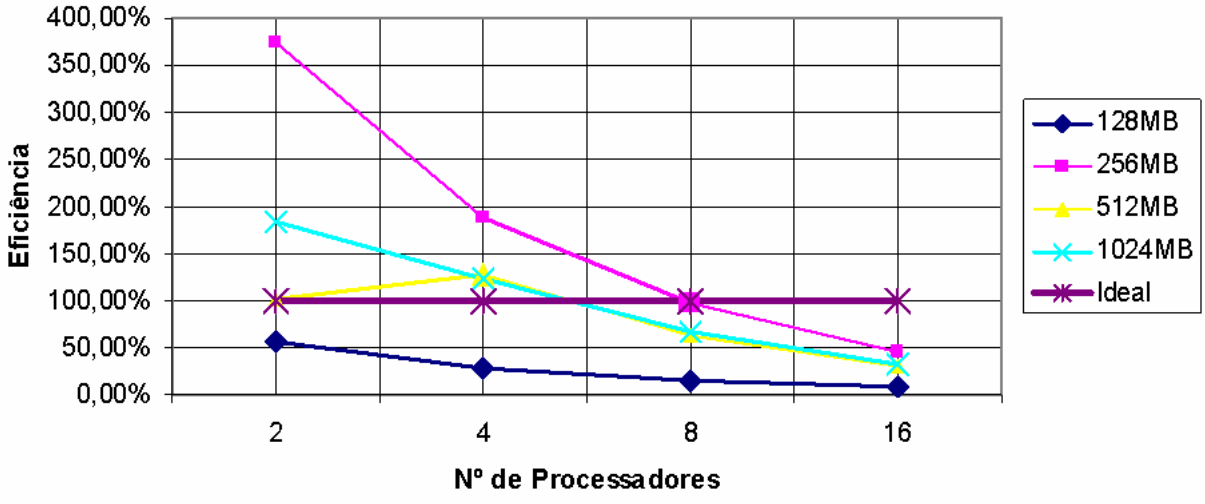

Figura 27: Gráfico de Eficiência - *MergeSort* Seqüencial X *MergeSort* Paralelo.

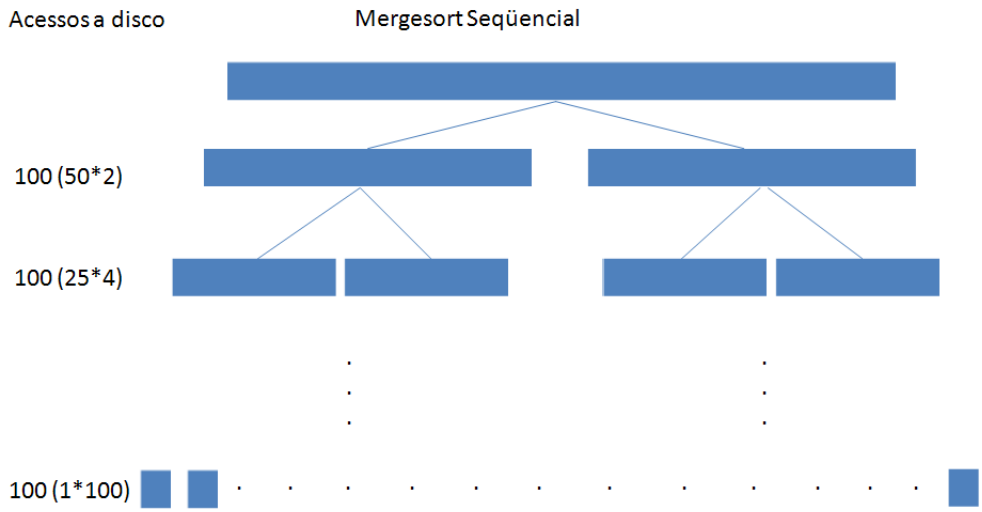

Figura 28: Acesso a disco no *MergeSort* Seqüencial.

Cada máquina ficaria responsável por 100MB de dados, onde cada micro realizasse 10 acessos a disco no primeiro nível (Ver Figura.29).

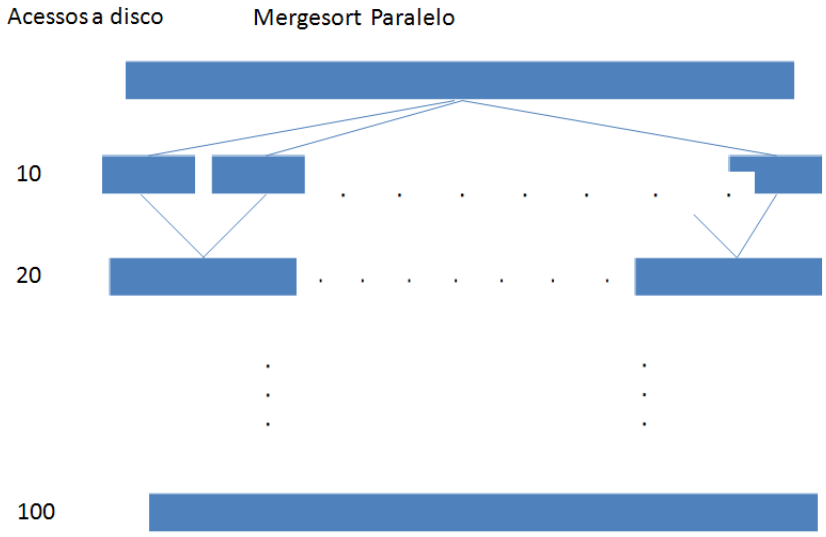

Figura 29: Acesso a disco no *MergeSort* Paralelo.

No segundo nível, como a máquina sucessora envia para a máquina antecessora seu arquivo ordenado, se passa a ter 5 máquinas envolvidas com arquvios de 200GB cada. Os acessos à disco são de 20, pois o arquivo foi dobrado de tamanho. Executando este processo até o final, a somatória de acesso à disco do processo em paralelo (10+20+...+100) deve ser menor que o seqüencial para, pelo menos, se ter a possibilidade de ser mais rápido que o mesmo. Deve-se considerar ainda a troca de mensagens pela rede, que é um pouco mais custosa que a transferência de dados por barramento.

6 CONCLUSÃO

Em face da viabilidade proposta, foi desenvolvido o algoritmo de ordenação *merge-sort* usando a biblioteca de programação paralela MPI, com o intuito de efetuar a ordenação de uma grande massa de dados com o mínimo de acesso à memória secundária. Para que desta forma fosse analisado qual tipo de computação possui melhor desempenho: computação paralela ou computação seqüencial.

Com o fim da análise, pode-se concluir que, mesmo parecendo óbvio, várias máquinas operando ao mesmo tempo nem sempre serão mais rápidas que uma. Esta variação depende de vários fatores como a rede, a forma da codificação do algoritmo, o cluster, o tamanho do arquivo e a quantidade de acesso à disco.

Através das limitações do algoritmo paralelo pode-ser perceber que o tratamento de acesso à disco é crucial durante um processo de ordenação. Os testes foram realizados com tamanhos de até 1GB de dados devido a uma limitação do ambiente que não permitia a criação de arquivos com 2GB de dados. Vale lembrar que para os testes é necessária a utilização de arquivos com tamanho de potência de 2. Estes testes com arquivos de 1GB não significam uma quantidade grande de informações na escala atual.

Como proposta para trabalhos futuros, deixa-se a idéia de uma melhoria no algoritmo de ordenação paralela, já que o resultado não foi o esperado. No primeiro nível poder-se-ia utilizar outro método de ordenação, ao invés do *mergesort*, como por exemplo, o *quicksort*. Além disso, em cada nível, uma verificação para saber se o tamanho do arquivo caberia na memória poderia ser feita, e, de acordo com o resultado, seria utilizado um algoritmo de memória interna ou externa. O tempo do primeiro passo do algoritmo, que dividia o arquivo no número de processadores envolvidos, foi computado sendo desnecessário, uma vez que a finalidade do trabalho era apenas verificar o desempenho da essência do algoritmo. Esta divisão poderia ter sido feita antes de iniciar o algoritmo paralelo. Outra proposta seria a tentativa de paralelização

de outros métodos de ordenação.

REFERÊNCIAS

[1] TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 2. ed. [S.l.]: Pearson Prentice Hall, 2006.

[2] KUROSE, R. *Redes de Computadores e a Internet uma nova abordagem*. [S.l.]: Ed. Makron Books.

[3] ABNOUS, N. B. A. *Nader. Pipeline and Bypassing in a VLIW Processor. IEEE Transactions on Parallel and Distributed Systems*. [S.l.: s.n.], 1994. (6, v. 5).

[4] PAWLOWSKI, R. B. M. Parallel sorting of large data volumes on distributed memory multiprocessors. *Parallel Computer Architectures: Theory, Hardware, Software, Applications*, 1993.

[5] SHI, J. S. H. Parallel sorting by regular sampling. *Department of Computing Science, University of Alberta*. Edmonton, Alberta. Canada.

[6] CLEARY, D. H. D. A comparison of lam-mpi and mpich messaging calls with cluster computing. *Journal of Young Investigators*, v. 15, 2008.

[7] KNUTH, D. *Sorting and Searching. The Art of Computer Programming*. [S.l.]: Addison Wesley Publ. Comp. Inc., 1973.

[8] LéVY, M. S. P. *Elementos para uma História das Ciências III: de Pasteur ao computador.* [S.l.]: Lisboa, Terramar, 1989.

[9] HOROWITZ, S. S. E. *Fundamentals of Data Structures*. [S.l.]: H. Freeman.

[10] DOWD, S. *High Performance Computing 2nd Ed, O´Reilly*. [S.l.: s.n.], 1998.

[11] CULLER, J. P. S. D. E. *Parallel Computer Architecture - A Hardware/Software Approach*. [S.l.]: Morgan Kaufmann, ISBN 1-55860-343-3, 1999. (ISBN 1-55860-343-3).

[12] DANTAS, M. *Computação Distribuída de Alto Desempenho: Redes, Clusters e Grids Computacionais*. [S.l.]: Axcel Books, 2005. (ISBN 85-7323-240-4).

[13] GOTTLIEB, G. S. A. A. *Highly parallel computing*. Redwood city: Benjamin/cummings. [S.l.]: 2ed, 1994.

[14] GUPTA, D. E. C. J. P. S. A. *Parallel computer architecture: a hardware/software approach.* [S.l.]: San Francisco: Morgan Kaufmann Publishers, 1999.

[15] FLYNN, M. J. *Some Computer Organizations and Their Effectiveness.* 9. ed. [S.l.]: IEEE Transactions on Computers, 1972. (948-960).

[16] HWANG, K. *Advanced computer architecture: parallelism, scalability, programmability.* [S.l.]: New York: Mcgraw-Hill, 1993.

[17] FLYNN, M. J. *Some Computer Organizations and Their Effectiveness.* [S.l.]: IEEE Transactions on Computers, 1972. (p.948-960).

[18] GRAMA, V. K. A. *Introduction to Parallel Computing - Design and Analysis of Algorithms*. [S.l.]: The Benjamin/Cummings Publishing, 2003.

[19] CORMEN CHARLES E LEISERSON, R. L. R. T. H. *Introduction to Algorithms*. The MIT Press Cambridge, Massachusetts - London, England: McGraw-Hill Book Company New York, St. Louis, San Francisco, Montreal and Toronto, 2000.

[20] IEEE COMPUTER SOCIETY PRESS. Mpi: A message passing interface. in proceedings of supercomputing '93. In: *Message Passing Interface Forum*. [S.l.], 1993. p. 878 – 883.

[21] RICHARDSON, H. High performance fortran: History, overview and current developments. *Thinking Machines Corporation*, 1996.

[22] AMZA ALAN L. COX, S. D. C. Treadmarks: Shared memory computing on networks of workstations. *Departamento de Ciência da Computação, Rice University.*

[23] BAKER, B. J. S. L. *Parallel programming*. [S.l.]: New York. Livros Horizonte, 1996.

[24] WALKER, D. W. *The design of a standard message passing interface for distriuted memory concurrent computers*. [S.l.]: Parallel Computing, 1994. (p.657-673, v. 20).

[25] SUN. *Sun Grid Engine: Enterprise Edition 5.3 Administration User's Guide*. [S.l.]: Sun Microsystems, 2002.

ANEXO A – Código desenvolvido do *MergeSort* Paralelo

```
# include <stdio.h>
# include <stdlib.h>
# include <string.h>
# include <time.h>
# include "mpi.h"

//função merge
int* intercala(int p, int q, int r, int v[])
{
   int i, j, k, *w;
   w =(int*)malloc((r-p)*sizeof(int));
   i = p; j = q;
   k = 0;

   while(i < q && j < r) {
      if(v[i] <= v[j]){
           w[k++] = v[i++];
      }
      else{
           w[k++] = v[j++];
      }
   }
   while(i < q){
      w[k++] = v[i++];
   }
   while(j < r){
```

```
      w[k++] = v[j++];
   }
   for(i = p; i < r; ++i){
       v[i] = w[i-p];
   }
   free(w);
   return v;
}

//mergesort
int* mergesort_i (int n, int v[])
{
   int p, r;
   int b = 1;
   while (b < n) {
      p = 0;
      while (p + b < n) {
         r = p + 2*b;
         if (r > n) r = n;
         intercala (p, p+b, r, v);
         p = p + 2*b;
      }
      b = 2*b;
   }
   return v;
}

// calcular tempo de execução
void calc_time(double Total_time)
{
     int h,m;
     double s;
     h=(int)Total_time;
     h=h/3600;
     m=(int)(Total_time-3600*h);
```

```
     m=m/60;
     s=Total_time-(3600*h+60*m);
     printf("O Tempo gasto foi: %dh %dm %lfs \n", h,m,s);
}

//a^b
int elevado (int a, int b)
{
  int temp=a,i;
  if(b==0)
    return 1;
  if(a==0)
    return 0;
  for(i=1; i<b; i++)
    temp*=a;
  return temp;
}

//v+v2
int* concatena(int* v,int* v2,int tam)
{
  int *v_aux=(int*)malloc(tam*sizeof(int));
  int i,j;
  for(i=0;i<tam/2;i++)
  {
    v_aux[i]=v[i];
  }
  for(j=0;j<tam/2;j++)
  {
    v_aux[i]=v2[j];
    i++;
  }
  free(v);
  free(v2);
  return v_aux;
```

```
}

// calcula tamanho do arquivo
long calcula_tamanho(FILE *arquivo) {

    // guarda o estado ante de chamar a função fseek
    long posicaoAtual = ftell(arquivo);
    // guarda tamanho do arquivo
    long tamanho;

    // calcula o tamanho
    fseek(arquivo, 0, SEEK_END);
    tamanho = ftell(arquivo);

    // recupera o estado antigo do arquivo
    fseek(arquivo, posicaoAtual, SEEK_SET);

    return tamanho;
}

//divide arquivo à ordenar em número-de-processos arquivos
void divide_arquivo(int qtd_processos)
{
  int qtd_elementos,qtd,i=0,arquivo=0,*v;
  char nome_arquivo[10];
  FILE *f=fopen("ordenar.dat","rb");
  if(f==NULL) exit(1);
  qtd_elementos=calcula_tamanho(f);
  v=(int*)malloc(qtd_elementos);
  fread(v,sizeof(int),qtd_elementos/sizeof(int),f);
  fclose(f);

  qtd=(qtd_elementos/sizeof(int))/qtd_processos;
  while(i<qtd_processos)
  {
```

```
    sprintf(nome_arquivo,"file%d.dat",arquivo);
    f=fopen(nome_arquivo,"wb");
    fwrite(&v[i*qtd],sizeof(int),qtd,f);
    arquivo++;
    i++;
    fclose(f);
  }
  free(v);
}

int main(int argc, char* argv[])
{
  //calcular tempo de execução
  double Total_time;
  time_t Start, End;
  // identificador do processo
  static int my_rank;
  // quantidade total de processos
  int p;
  // variável qualquer
  int i,j;
  // quantidade de elementos à ordenar por cada processo
  int qtd_elementos;
  //
  MPI_Status status;
  // inicia o MPI.
  MPI_Init(&argc, &argv);
  // descobre o identificador (rank) do processo.
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  // descobre a quantidade total de processos
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  // arquivo contendo dados à ordenar
  FILE *f;
  // nome do arquivo de dados
  char nome_arquivo[10];
```

```
printf("%d: dividindo arquivo\n",my_rank);

//divide arquivo no número de processos
if(my_rank==0)
{
  divide_arquivo(p);
  for(i=1;i<p;i++)
    MPI_Send(&i, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
}
else
  MPI_Recv(&j, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

  printf("%d: divididos\n",my_rank);

//Inicio da função de Calculo de tempo
if(my_rank==0)
Start=time(NULL);

// cada máquina pega sua parte do arquivo
sprintf(nome_arquivo,"file%d.dat",my_rank);


printf("%d: abrindo arquivo arquivo\n",my_rank);

// abre o arquivo binário
f=fopen(nome_arquivo,"rb");
if(f==NULL)
  return 0;


printf("%d: arquivo aberto\n",my_rank);

// calcula qtd de elementos no arquivo
qtd_elementos=calcula_tamanho(f)/sizeof(int);
```

```
  // aloca os vetores
  int *v=(int*)malloc(qtd_elementos*sizeof(int));
  int *v2=NULL;

  // lê dados do arquivo f e coloca-os no vetor v
  fread(v,sizeof(int),qtd_elementos,f);

  //fecha o arquivo
  fclose(f);


  printf("%d: ordenando mergesort\n",my_rank);

  // ordena vetor lido do arquivo
  v=mergesort_i(qtd_elementos,v);
  printf("%d: ordenado mergesort\n",my_rank);

  int enviou=0;
  i=1;

  while(elevado(2,i)<=p && enviou==0)
  {
    if(my_rank%elevado(2,i)==0)
    {
      //recebe vetor
      //merge vetores
      v2=(int*)malloc(qtd_elementos*elevado(2,i-1)*sizeof(int));
      MPI_Recv(v2, qtd_elementos*elevado(2,i-1), MPI_INT,
my_rank+elevado(2,i-1), 0, MPI_COMM_WORLD, &status);
      v=concatena(v,v2,2*qtd_elementos*elevado(2,i-1));
      v=intercala(0,qtd_elementos*elevado(2,i-1),
2*qtd_elementos*elevado(2,i-1),v);
      printf("rank: %d recebeu de : %d\n",my_rank, my_rank+elevado(2,i-1));
 /* for(j=0;j<qtd_elementos*elevado(2,i);j++)
```

```
        printf("%d: %d\n",my_rank,v[j]);*/
    }
    else
    {
      //envia vetor
      MPI_Send(v, qtd_elementos*elevado(2,i-1),
MPI_INT, my_rank-elevado(2,i-1), 0, MPI_COMM_WORLD);
      printf("rank: %d enviou para : %d\n",my_rank, my_rank-elevado(2,i-1));
      enviou=1;
      free(v);
    }
    i++;
  }

  printf("%d: gerando arquivo final\n",my_rank);

  if(my_rank==0)
  {
    End=time(NULL);
    printf("a %d\n", End-Start);
    Total_time=End-Start;
    f=fopen("ordenado.dat","wb");
    if(f==NULL) return 1;
    fwrite(v,sizeof(int),qtd_elementos*elevado(2,i-1),f);
    fclose(f);
    calc_time(Total_time);
  }

  printf("%d: gerado arquivo\n",my_rank);

  printf("Fim %d\n",my_rank);
  return 0;
  MPI_Finalize();
}
```