Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

PROJETO E ANÁLISE DE ALGORITMOS

QuickSort Paralelo

Quarto Trabalho Prático - disponível em: http://www.dcc.ufmg.br/~menoti/paa/tp4/

David Menoti Gomes

Professor - Nivio Ziviani

Belo Horizonte
28 de junho de 2004

Sumário

1 Introdução **1**

2 Algoritmo **2**
2.1 Descrição do Algoritmo 2
2.2 Análise de Complexidade de Tempo do Algoritmo 4

3 Estruturas de Dados **8**

4 Aspectos de Implementação **12**
4.1 1ª Fase 12
4.2 2ª Fase 13
4.3 3ª Fase 13
4.4 4ª Fase 15
4.5 Geral 20

5 Experimentos **21**

6 Conclusões **27**

7 Anexos **28**
7.1 Tabelas dos Experimentos 28
7.2 Códigos fontes 34

1 Introdução

O problema de ordenação corresponde ao processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente [7]. O objetivo principal da ordenação é facilitar a recuperação posterior de itens do conjunto ordenado. Imagine como seria difícil utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética. A atividade de colocar as coisas em ordem está presente na maioria das aplicações em que os objetos armazenados têm de ser pesquisados e recuperados, tais como dicionários, índices de livros, tabelas e arquivos.

A ordenação é um das atividades mais comuns em computadores seriais [6]. Muitos algoritmos incorporam uma ordenação de forma que a informação possa ser acessada posteriormente de forma eficiente. A ordenação tem mais importância para desenvolvedores de algoritmos paralelos; ela é frequentemente usada para realizar permutações de dados genéricos em computadores com memória distribuída. Essas operações de movimentos de dados podem ser usadas para resolver problemas em Teoria de Grafos, Geometria Computacional e Processamento de Imagens em tempo ótimo ou próximo do ótimo.

Neste trabalho é implementado o algoritmo de ordenação paralela de dados através de amostras regulares - de agora em diante chamado, simplesmente, de PSRS (*Parallel Sorting by Regular Sampling*)- descrito em [6] e inicialmente apresentado em [5], que toma por base o algoritmo de ordenação *QuickSort*, implementado em um computador MIMD, mais precisamente em uma uma rede de estações de trabalho [2]. O Algoritmo *QuickSort* utiliza a metodologia de comparações de chaves e um paradigma de divisão e conquista. Além disso ele é classificado como um algoritmo de ordenação interna, isto é, ele ordena conjuntos de dados que caibam na memória principal do computador.

O trabalho é organizado da seguinte forma. Na Seção 2 o algoritmo PSRS é apresentado, sendo discutido suas fases e também uma análise de complexidade de tempo do mesmo. As estruturas de dados utilizadas na implementação do Algoritmo são apresentadas na Seção 3. Na Secão 4 são apresentados alguns aspectos de implementação, detalhando a implementação do mestre e dos escravos. Os experimentos são apresentados na Seção 5. E, finalmente, na Seção 6 conclusões são apontadas.

2 Algoritmo

Nesta seção é apresentada uma descrição do Algoritmo PSRS (*Parallel Sorting by Regular Samples*) que foi desenvolvido por [5] e uma análise da complexidade do algoritmo.

2.1 Descrição do Algoritmo

Considere que cada processador p envolvido gerencia somente seus elementos, ou seja, não existe um compartilhamento de memória entre os processadores, visto que a arquitetura usado é uma Rede de Estações de Trabalho - NOW (Network of *Workstation*).

O algoritmo PSRS tem quatro frases. Assuma que o conjunto de dados a ser ordenado tem n elementos. Na fase um, cada um dos p processadores usa o algoritmo sequencial *QuickSort* (Algoritmo 1) para ordenar um conjunto de dados contínuo de não mais do que $\lceil n/p \rceil$ elementos. Agora, os n elementos formam p conjuntos de dados independentes de tamanho não maior do que $\lceil n/p \rceil$ elementos. Cada processador seleciona elementos nas posições 1, $(n/p^2) + 1$, $(2np^2) + 1$, ..., $(p - 1)(n/p^2) + 1$ como amostras regulares do seu conjunto ordenado, selecionando p amostras regulares.

Na segunda fase do algoritmo um processador mestre recebe e ordena as p^2 amostras regulares - cada processador enviou p amostras regulares. Dado esse conjunto de p^2 amostras regulares ordenado, o algoritmo seleciona $p-1$ valores de pivôs. Esse pivôs são escolhidos pelos índices $p + \lfloor p/2 \rfloor, 2p + \lfloor p/2 \rfloor, \ldots, (p-1)p + \lfloor p/2 \rfloor$ do conjunto ordenado. Nesse instante, os pivôs são enviados aos escravos que particionam seus conjuntos em não mais do que $\lceil n/p \rceil$ elementos em partições disjuntas de p pedaços, usando os valores dos pivôs como divisores entre os pedaços.

Na terceira fase do algoritmo, cada processo i mantém sua i-ésima partição e envia a j-ésima partição ao processor j. Em outras palavras, cada processador mantém uma partição e enviam, ou distribuem, $p - 1$ partições para os outros processadores.

Durante a quarta fase do algoritmo cada processador *merge* suas p partições em um único conjunto. Os valores do conjunto de cada processador são disjuntos aos valores dos conjuntos de outros processadores. Portanto, a concatenação dos conjuntos individuais de cada processador formam o conjunto final ordenado dos dados. Os dados devem ser concatenados respeitando a ordem dos processadores, conjunto oriundo do primeiro processador 1, depois conjunto oriundo do processador 2, até o conjunto vindo do processador p. Isto porque os menores valores estão no processador 1, o segundo menor conjunto está no processador 2, e assim por diante.

Uma ilustração da execução desse algoritmo para 40 elementos e 4 processadores pode ser vista na Figura 1. Na Figura 1(a) é apresentado o conjunto de dados não ordenado. Esse conjunto é divido em 4 partes, uma para cada processador, gerando assim 4 subconjuntos. Esses subconjuntos são ordenados com o algoritmo *QuickSort* sequencial gerando quatro subconjuntos ordenados, como apresentado na Figura 1(b). De cada um dos subconjuntos ordenados, são escolhidos p amostras regulares, segundo uma ordem previamente estabelecida em função do número p. Estas amostras são enviadas a um processador mestre, gerando um pequeno conjunto de p^2 amostras regulares (Figura 1(c) - Amostras Regulares). Então, essas amostras

Algorítmo 1 Algoritmo *sequencial* de ordenação *QuickSort*.

$procedure$ QuickSort ($Vector : A, integer : Left, Right$)
$integer : Lb, Ub$; {Lower and Upper boundary}
Partition($A, Left, Dir, lb, ub$);
if $Left < Lb$ **then**
  Quicksort($A, Left, lb$)
end if
if $Ub < Right$ **then**
  Quicksort($A, ub, Right$)
end if

$procedure$ Partition($Vector : A, integer : Left, Right, Lb, Ub$)
$integer : pivot$;
$Lb \leftarrow Left$;
$Ub \leftarrow Right$;
$pivot \leftarrow (Lb + Ub)/2$;
repeat
  while $A[pivot] > A[Lb]$ **do**
    $Lb \leftarrow Lb + 1$
  end while
  while $A[pivot] < A[Ub]$ **do**
    $Ub \leftarrow Ub - 1$
  end while
  if $Lb \leq Ub$ **then**
    $swap(A[Lb], A[Ub])$;
    $Lb \leftarrow Lb + 1$;
    $Ub \leftarrow Ub - 1$;
  end if
until $Lb > Ub$;

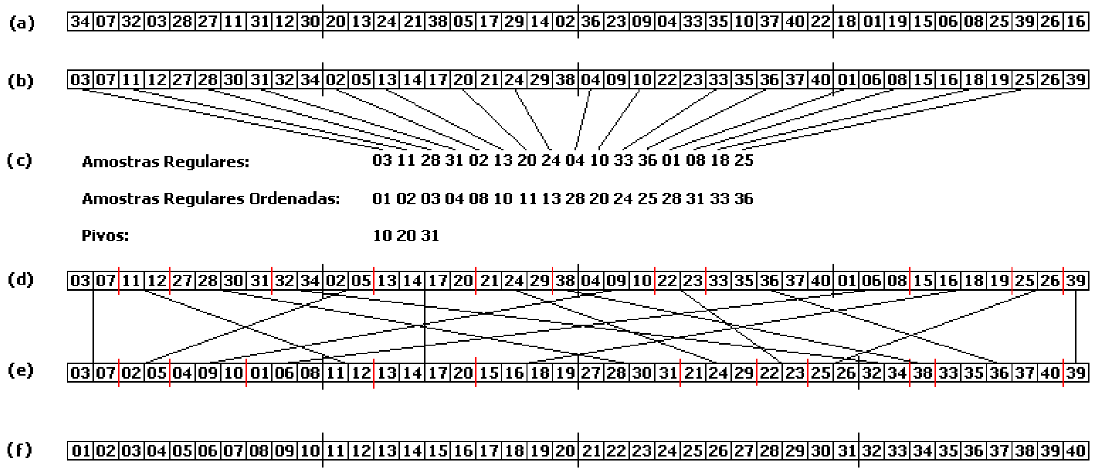

Figura 1: Exemplo da execução do algoritmo PSRS para um conjunto de 40 elementos com 4 processadores. (a) Conjunto de 40 elementos desordenados; (b) Cada processador ordena seus elementos compartilhados do conjunto (10 elementos) usando o algoritmo *QuickSort* sequencial; (c) Seleciona as amostras regulares de cada conjunto ordenado (primeiro, terceiro, sexto e oitavo elementos). O processador mestre ordena essas p^2 amostras regulares. Escolhe os elementos pivôs (sexto, décimo e décimo quarto elementos); (d) Usa os pivôs computados no passo (c) para dividir cada subconjunto ordenado em quatro partições. (e) Copia os subconjuntos ordenados em um novo vetor de forma que os elementos que tem a mesma relação entre os pivos são colocados juntos. (f) cada processador *merge* o seu subconjunto ordenado, gerando, finalmente, o conjunto de dados ordenado.

são ordenadas por um algoritmo de ordenação qualquer (Figura 1(c) - Amostras Regulares Ordenadas) - deve-se levar em conta o fato de que esse número é pequeno. Desse pequeno conjunto de amostras regulares ordenado, são selecionados $p-1$ elementos pivôs (Figura 1(c) - Pivôs), também, através de uma regra previamente estabelecida. Esses $p-1$ pivos são utilizados por cada processador p para dividir cada subconjunto em p partições, conforme apresentado em 1(d). Neste ponto, ocorre uma troca de partições entre os processadores escravos, de forma que cada processador i mantém somente a i-ésima partição enviando e recebendo as outras partições aos outros processadores escravos. O resultado desse processo é ilustrado na Figura 1(e). Nesses novos subconjuntos é efetuado um processo de *merge* de forma que os elementos de cada subconjunto estejam ordenados e dessa forma todos os elementos do conjunto também o estejam, conforme apresentando na Figura 1(f).

2.2 Análise de Complexidade de Tempo do Algoritmo

Para simplificar a análise do algoritmo PSRS, assume-se que p processadores ordenam n elementos distintos, onde p é par, $n = p^2k$ e k é um inteiro positivo. Para uma análise mais rigorosa, veja [5].

Definição 1. *Seja $X = \{X_1, \ldots, X_n\}$ o conjunto de n elementos a ser ordenado.*

Definição 2. *Seja $Y = \{Y_1, Y_2 \ldots, Y_{p^2}\}$ o conjunto ordenado de p^2 amostras regulares selecionadas dos p subconjuntos ordenados de X.*

Definição 3. *Seja $N(cond)$ o número de elementos de X que satisfaçam uma certa condição booleana cond.*

Por exemplo, $N(\leq Y_2)$ é o número de elementos de X menores ou iguais à Y_2, a segunda amostra regular selecionada dos subconjuntos ordenadas de X.

Primeiro, estabelece-se um limite inferior sobre o número de elementos em X menores ou iguais que um certo número de elementos do conjunto de amostras regulares Y.

Lema 1. *Para $1 \leq i \leq p$*

$$N(\leq Y_{(i-1)p+p/2} \geq \begin{cases} p/2, & se\ i = 1 \\ \frac{n}{p^2}((i-1)p - p/2) + p, & se\ i > 1 \end{cases}$$

Prova 1. *Quando $i = 1$, $Y_{(i-1)p+p/2}$ é $Y_{p/2}$. Visto que os elementos de Y são ordenados, existem $p/2$ elementos de Y que são menores ou iguais a $Y_{p/2}$. Os elementos de Y são selecionados de X. Portanto, existem no mínimo $p/2$ elementos de X menores ou iguals a $Y_{p/2}$.*

Cada elemento de Y representa um conjunto de n/p^2 elementos de X com valor igual ou maior. Quando $i > 1$, a soma $(i-1)p + p/2$ é maior que p. Isto significa que quando considera-se as amostras $Y_1, Y_2, \ldots, Y_{(i-1)p+p/2}$, algumas amostras devem ter vindo de algum processador. Se Y_j e Y_k vem do mesmo processador e $j < k$, então todos os n/p^2 elementos de X representados por Y_j devem ser menores ou iguais a Y_k. Isso por sua vez, é menor ou igual a $Y_{(i-1)p+p/2}$. Para as p amostras representando os maiores elementos de Y associados com cada processador, sabe-se somente que as próprias amostras são menores ou iguais à $Y_{(i-1)p+p/2}$. Portanto, o número de elementos de X menores ou iguais aos elementos de Y é

$$\begin{aligned} N(\leq Y_{(i-1)p+p/2}) &= \frac{n}{p^2}((i-1)p + \frac{p}{2} - p) + (1)p \\ &= \frac{n}{p^2}((i-1)p + \frac{p}{2}) + p \end{aligned}$$

Agora, determina-se um limite superior sobre o número de elementos de X maiores que um certo número de elementos do conjunto de amostras regulares Y.

Lema 2. *Para $1 \leq i \leq p$*

$$N(> Y_{ip+p/2}) \geq \frac{n}{p^2}((p-i)p - \frac{p}{2} + 1) - 1$$

Prova 2. *Existem $p^2 - (ip+p/2) = (p-i)p - p/2$ amostras maiores que $Y_{ip+p/2}$. Isto significa que $(n/p^2)((p-i)p - p/2)$ elementos de X associados com estas amostras são maiores que $Y_{ip+p/2}$. Os $n/p^2 - 1$ elementos de X imediatamente seguintes à $Y_{ip+p/2}$ também são maiores que $Y_{ip+p/2}$. Portanto,*

$$\begin{aligned} N(\leq Y_{(ip+p/2)}) &\geq \frac{n}{p^2}((p-i)p - \frac{p}{2}) + (\frac{n}{p^2} - 1) \\ &\geq \frac{n}{p^2}((p-i)p - \frac{p}{2}) + 1) - 1 \end{aligned}$$

Definição 4. *Seja Φ_i o número de elementos de X intercalados pelo processador i na fase quatro do algoritmo PSRS.*

Teorema 1. *Um limite superior sobre $max_{1 \leq i \leq p} \Phi_i$ é $2n/p - n/p^2 - p + 1$.*

Prova 3. *Começa-se devirando o limite superior de Φ_i para cada um dos três casos $i = 1$, $i = p$ e $1 < i < p$.*
Caso 1*: $i = 1$.*

Todos os X elementos intercalados pelo processador 1 devem ser menores ou iguais a $Y_{p+p/2}$. Do Lema 2 vem que

$$N(> Y_{p+p/2}) \geq \frac{n}{p^2}((p-1)p - \frac{p}{2} + 1) - 1$$

e, portanto

$$\begin{aligned} \Phi_i = N(\leq Y_{p+p/2}) &\leq n - \frac{n}{p^2}((p-1)p - \frac{p}{2}) + 1) - 1 \\ &\leq \frac{n}{p^2}(p + \frac{p}{2} - 1) + 1 \end{aligned}$$

Caso 2*: $i = p$*

Todos os X elementos intercalados pelo processador p devem ser maiores que $Y_{(p-1)p+p/2}$. Do Lema 1 vem que

$$N(\leq Y_{(p-1)p+p/2}) \geq \frac{n}{p^2}((p-1)p - p/2) + p$$

e, portanto,

$$\Phi_p = n - N(\leq Y_{(p-1)p+p/2}) \leq \frac{n}{p^2}(p + \frac{p}{2}) - p$$

Caso 3*: $1 < i < p$*

O número de elementos intercalados pelo processador i é igual a $n - N(\leq Y_{(i-1)p+p/2}) - N(> Y_{ip+p/2})$. Aplicando-se os Lemas 1 e Lema 2, vem que

$$\begin{aligned} \Phi_i &\leq n - \left(\frac{n}{p^2}((i-1)p - \frac{p}{2}) + p\right) - \left(\frac{n}{p^2}((p-i)p - \frac{p}{2} + 1) - 1\right) \\ &\leq \frac{n}{p^2}(2p - 1) - p + 1 \end{aligned}$$

O maior limite superior para Φ_i vem do Caso 3. Portanto, um limite superior sobre o número de elementos de X intercalados por qualquer processador na fase quatro do algoritmo PSRS é $2n/p - n/p^2 - p + 1$.

Agora, que tem-se estabelecido um limite superior para o número de elementos que qualquer processador deve intercalar na fase quatro do algoritmo PSRS, pode-se analizar a complexidade computacional do algoritmo PSRS.

Na fase um cada processador executa o algoritmo sequencial *QuickSort* sobre n/p elementos. A complexidade computacional é $O((n/p)\log(n/p)) = O(n/p \log n)$. (Realmente, $O(n \log n)$ é a complexidade de tempo esperada para realizar o *QuickSort* em um lista de n elementos; a complexidade de tempo no pior caso é $O(n^2)$.

Entretanto, assume-se $O(n \log n)$ para esta análise. Caso seja desejado usar a complexidade de pior caso na análise, pode-se substituir o algoritmo pelo *MergeSort* ou *HeapSort* que tem complexidade de tempo de pior caso $O(n \log n)$.)

Na fase dois do algoritmo um processador ordena os p^2 elementos de Y (amostras regulares). Este passo tem complexidade $O(p^2 \log p^2) = O(p^2 \log p)$.

Na terceira fase do algoritmo cada processador envia porções de suas seções de X para outros processadores, e ao mesmo tempo recebe essas porções de outros processadores. Visto que cada processador tem n/p elementos, a complexidade deste passo é $O(n/p)$ (ignorando os tempos de inicialização de envio de mensagens em sistemas de memória distribuída).

Na fase quatro do algoritmo cada processador intercala (*merge*) as p sublistas ordenadas. Do Teorema 1 vem que nenhum processador tem mais que $2n/p$ elementos para intercalar (*to merge*). E, portanto, a complexidade de tempo desta fase é $O((n/p) \log p)$. (De fato, é possível implementar esse *merge* com custo linear $O(n)$, pois os subconjuntos já estão ordenados.).

A complexidade de tempo total do algoritmo PSRS é $O((n/p) \log n + p^2 \log p + n/p \log p)$. Se $n \geq p^3$, o que é fato para situações reais, o primeiro termo domina os outros, e, portanto, a complexidade de tempo do algoritmo PSRS é $O((n/p) \log n)$, cujo custo é ótimo.

3 Estruturas de Dados

Antes de ser apresentado os aspectos de implementação, faz-se necessário apresentar as estruturas de dados utilizadas na confecção do trabalho.

A princícpio é apresentado no Programa 1 as estruturas de dados básicas usadas no trabalho, que estão contidas no arquivo tda.h (Programa 5), anexado ao trabalho. Dentre as estruturas contidas no Programa 1, a estrutura *strArgs* é um *struct* que contém dois campo inteiros, *sockfd* e *slaveId*. Esta estrutura é utilizada para a passagem de parâmetros para as *threads* utilizadas tanto no programa mestre como escravo. Uma macro, *TCP_PORT* é utilizada para definir a porta a qual será usada para a comunição *socket* do processo mestre com os processos escravos. Um vetor de cadeias, *ips*, é utilizado para definir os *ip's* de cada uma das máquinas escravas. Uma matriz de inteiro, *tcp_ports* é utilizada para definir as portas de comunicação para o *socket* entre os processos escravos que serão utilizadas na terceira fase do algoritmo, onde cada elemento a_{ij} dessa matriz é usado como sendo a porta de comunicação entre o processo i e o j, de forma que o processo i enviará dados ao processo j somente através do *socket* estabelecido por esta porta. Observa-se que a_{ij} é diferente de a_{ji}, pois existe um *socket* para a comunicação em cada sentido entre dos processos i e j. O vetor de inteiro, *idx* é usado como sendo um índice para os processadores escravos saberem qual massa de dados utilizar. Esses índices referem-se as colunas do vetor de cadeias, *processorfiles* e as linhas aos processadores. De acordo com a massa de dados escolhida, cada processador escravo é encarregado de carregar um dado arquivo.

```
struct strArgs
{
  int sockfd;
  int slaveId;
};

#define TCP_PORT 1024

char ips[4][14]={"150.164.6.108",  /* traira 108*/
                 "150.164.6.102",  /* carpa */
                 "150.164.6.103",  /* lambari */
                 "150.164.6.65" }; /* niger */

int tcp_ports[4][4]={{  -1,2012,2013,2014},
                     {2021,  -1,2023,2024},
                     {2031,2032,  -1,2034},
                     {2041,2042,2042,  -1}};

int idx[] = {10,50,100};

char processorfiles[4][3][30] = {
{"../data/data-10-1.bin","/var/tmp/paa/dados50MB-1.bin",
"/var/tmp/paa/dados100MB-1.bin"},
{"../data/data-10-2.bin","/var/tmp/paa/dados50MB-2.bin",
"/var/tmp/paa/dados100MB-2.bin"},
{"../data/data-10-3.bin","/var/tmp/paa/dados50MB-3.bin",""},
{"../data/data-10-4.bin","/var/tmp/paa/dados50MB-4.bin",""}};
```

Programa 1: Estruturas de dados básicas usadas no trabalho.

Um conceito interessante utilizado nesse trabalho é o de *Thread*. Embora, não sabendo exatamente onde essa explicação seria mais conveniente, preferiu-se apresentar os conceitos utilizados de *thread* nessa seção. Mas o que realmente é uma *Thread*? Tecnicamente falando, uma *Thread* é definida como um conjunto independente de instruções que pode ser escalonada pelo sistema operacional [1].

Mais especialmente falando, nesse trabalho foi utilizado *POSIX threads* ou simplesmente *pthreads*. Considerou-se relevante apresentar, brevemente, dois tipos de variáveis relacionadas a *pthreads*: Variáveis de exclusão mútua (*mutex*) e Variáveis de condição (*cond*) que são largamente utilizadas nesse trabalho.

O uso de variáveis de exclusão mútua consiste em uma forma primária de implementar sincronização entre *threads* e para a proteção de dados compartilhados quando várias alterações (escritas) podem ocorrer.

As variáveis de condição fornecem outra forma de sincronizar *threads*. Enquanto as variáveis de exclusão mútua implementam a sincronização através do controle da *thread* ao acesso de dados, as variáveis de condição permitem que as *threads* sejam sincronizadas através de valores reais de dados.

Sem o uso de variáveis de condição, o programador precisaria de ter as *threads* executando continuamente (possivelmente em uma seção crítica), para verificar se uma dada condição é alcançada. Isto pode consumir muito recurso do sistema, pois a *thread* estaria continuamente ocupada com esta atividade. Uma variável de condição é uma forma de alcançar o mesmo ojetivo sem a necessidade de constante verificação. Uma variável de condição é sempre utilizada em junto com uma variável de exclusão mútua.

Um exemplo da utilização de variável de exclusão mútua e variável de condição pode ser visto no Programa 2. Este simples exemplo de código demonstra o uso de várias rotinas com variáveis de condição e variáveis de exclusão mútua em *pthread*. O programa principal cria três *threads*. Duas das *threads* realizam o trabalho de atualizar a variável *count*. A terceira *thread* aguarda até que a variável *count* alcance um valor específico.

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int       count = 0;
int       thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *idp)
{
  int j,i;
  double result=0.0;
  int *my_id = idp;

  for (i=0; i<TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when condition is
```

```
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %d, count = %d  Threshold reached.\n",
             *my_id, count);
      }
    printf("inc_count(): thread %d, count = %d, unlocking mutex\n",
     *my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
    for (j=0; j < 1000; j++)
      result = result + (double)random();
    }
  pthread_exit(NULL);
}

void *watch_count(void *idp)
{
  int *my_id = idp;

  printf("Starting watch_count(): thread %d\n", *my_id);

  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait
  routine will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %d Condition signal
           received.\n", *my_id);
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  int i, rc;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /*
  For portability, explicitly create threads in a joinable state
  so that they can be joined later.
  */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, inc_count, (void *)&thread_ids[0]);
```

```
    pthread_create(&threads[1], &attr, inc_count, (void *)&thread_ids[1]);
    pthread_create(&threads[2], &attr, watch_count, (void *)&thread_ids[2]);

    /* Wait for all threads to complete */
    for (i = 0; i < NUM_THREADS; i++) {
      pthread_join(threads[i], NULL);
    }
    printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

    /* Clean up and exit */
    pthread_attr_destroy(&attr);
    pthread_mutex_destroy(&count_mutex);
    pthread_cond_destroy(&count_threshold_cv);
    pthread_exit(NULL);

}
```

Programa 2: Uso de Variáveis de Exclusão Mútua (mutex) e de Condição (cond) em *pthreads*.

O uso de variáveis de exclusão mútua e de condição são interessantes nesse trabalho, quando deseja-se que somente uma *thread* realize uma determinada tarefa por vez e que o processo aguarde até que todas as *thread* tenham realizado determinada tarefa.

Uma boa referência *on-line* para *POSIX pthread* pode ser encontrada em [1]. Demais detalhes sobre estruturas de dados e *sockets* são dados na seção seguinte, Seção 4.

4 Aspectos de Implementação

Nessa seção são descritos de forma suscinta os passos relacionados a implementação do algoritmo PSRS.

Um exemplo de como executar a implementação do algoritmo PSRS para 4 processadores pode ser visto na Figura 2. A princípio, os quatro processadores escravos (traira, carpa, lambari e niger) devem ser executados (eles podem ser executados em qualuqer ordem, pois ficam aguardando a máquina mestre). Quando o processo mestre é inicializado na máquina mestre (nilo), a execução do algoritmo é iniciada nas quatro estações escravas. O processador mestre envia informações para os escravos, como o identificador de cada processador, o número de processadores envolvidos em todo o processo, e a massa de dados que eles devem manipular (de fato, é enviado o índice de arquivo, que através deste consulta-se uma tabela (Programa 1 - $processorfiles$)), e dessa forma o processo de ordenação é desencadeado. Nos parágrafos seguintes detalhes serão dados sobre a execução de cada passo do algoritmo e as tarefas desenvolvidas por cada um: mestre e escravo.

```
traira:~paa/tp4/src->./slave

carpa:~paa/tp4/src->./slave

lambari:~paa/tp4/src->./slave

niger:~paa/tp4/src->./slave

nilo:~/paa/tp3/src->./master 4 50
```

Figura 2: Executando a implementação do algoritmo PSRS para 4 processadores.

Nessa pré-fase de execução do algoritmo os processos escravos estão aguardando (função da biblioteca *socket* - accept) o processo mestre se conectar a eles através da função da biblioteca *socket* - connect. Foi convencionado que todos os escravos estariam esperando/ouvindo o processo mestre por uma única porta, TCP_PORT (1024), definida no arquivo de estruturas de dados (Programa 5).

4.1 1ª Fase

Após o processo mestre desencadear o processo de execução do algoritmo nos processos escravos, a execução no mestre sessa e fica aguardando o envio das amostras regulares pelos processos escravos que será realizada no inicio da segunda fase. Nessa fase os processos escravos carregam os arquivos do disco rígido e ordenam essas massas de dados segundo o algoritmo sequencial *QuickSort*. p amostras regulares são escolhidas do arquivo ordenado segundo índices pré-definidos. A implementação dessa fase, por parte dos escravos, pode ser vista no Programa 7, função $firstphase$. Nenhuma restrição aos processos escravos é feita, visto que nenhuma *thread* foi inicializada. Já para o processo mestre foi inicializado p *threads* que estão aguardando o recebimento de amostras regulares dos processos escravos.

4.2 2ª Fase

Nessa fase as p amostras regulares escolhidas por cada processador escravo são enviadas ao processador mestre. Este por sua vez, estava aguardando que todas a p^2 amostras fossem recebidas (essa situação de espera é implementada usando variáveis de condição e de exclusão mútua), ordena as p^2 amostras regulares recebidas (cada processador enviou p amostras regulares de um total de p processadores, perfazendo as p^2 amostras regulares) através do método de ordenação *InsertSort* (Programa 9). O algoritmo *QuickSort* não foi utilizado neste passo, pois o número de elementos a serem ordenados é muito pequeno e segundo a literatura [7] para pequenos conjuntos de dados é interessantes usar algoritmos menos complexos. Após o conjunto de p^2 amostras regulares estar ordenado, $p-1$ pivôs são escolhidos para particionarem os dados nos processos escravos. Então, esses $p-1$ elementos são enviados a cada um dos processos escravos. Com o objetivo de paralelizar esse envio são inicializadas, novamente, p *threads* no processo master. Ao mesmo tempo, cada processo escravo ficou aguardando pelo recebimento desses $p-1$ pivôs, após ter enviado as suas p amostras regulares. Após enviar os $p-1$ pivôs para os processos escravos, o processo mestre fica aguardando a finalização do algoritmo, que será sinalizada pelos processos escravos na fase 4. Os processos escravos uitilizam esses $p-1$ pivôs para dividirem logicamente a massa de dados ordenada em p subconjuntos ordenados, onde o primeiro subconjunto contém dados menores ou iguais ao primeiro pivô, o segundo subconjunto contém dados menores ou iguais ao segundo pivô e ao mesmo tempo maiores que o primeiro pivô e assim por diante. Essa divisão da massa de dados em partições será utilizado no próximo passo para a troca de dados entre os escravos. Novamente, por parte dos escravos, nenhuma restrição aos processos é imposta, visto que *threads* não são inicializadas. A implementação dessa fase é apresentada no Programa 7, função *secondphase*.

4.3 3ª Fase

Essa fase é realizada somente entre os processadores escravos. O processador mestre fica aguardando a sinalização de fim que será dada na quarta fase. Essa é a fase mais crítica do programa, pois é necessário estabelecer a comunicação entre todos os escravos. Para isso são geradas *threads* para comunicação em ambos os sentidos, ou melhor, uma para cada sentido. Por exemplo, o processador 1 precisa comunicar-se com o processador 2. Mas ambos precisam enviar e receber dados uns para os outros. Então são criadas 2 *threads* e 2 *sockets* para fazer a comunicação entre esses dois processos. Poderia ser levantanda a pergunta: Através de um único *socket* não seria possível fazer a comunicação em ambos os sentidos entre dois processos? A resposta é sim, mas encontrou-se dificuldades em fazer essa implementação, e a criação de duas *threads* e dois *sockets* permitiu que o processo seja escalonado pelo sistema operacional. Portanto, cada processo escravo cria $2(p-1)$ *threads* e *sockets* para fazer a comunicação entre seus "companheiros". Não são criados $2p$ *threads* e *sopckets*, e sim $2(p-1)$, pois não é necessário fazer comunicação com o próprio processo. Esse uso de mais *threads* também permite que um processo escravo após receber todas as suas partições, possa continuar a execução, enquanto que as outras *threads* estão enviando dados para outros escravos.

Após as *threads* criarem os *sockets* através das funções *Listen* e *Talk* (Programa 7), as *threads Talk* enviam aos outros processos através das *threads Listen* o tamanho do subconjunto a ser enviado. Essa informação é armazenada em cada processador, e novos blocos de memória são alocados para receber esses novos subconjuntos.

Tendo os novos blocos de memória alocados, cada processo escravo inicializa mais $2p$ *threads*, p *threads* para enviar os subconjuntos e p *threads* para receber os subconjuntos. No entanto, uma das *threads* para receber os subconjuntos, a *thread* relacionada ao próprio processo, simplesmente copia os dados para a área de memória diretamente, enquanto que as outras $p - 1$ recebem os dados dos outros processadores. O mesmo acontece para as p *threads* para enviar os subconjuntos, porém a *thread* que está relacionada ao próprio processo não realiza nenhuma tarefa em especial. Um mero toque singular ao trabalho, fator de diferenciação.

O processo escravo aguarda até que todas as *threads* de recebimento tenham sido finalizadas para passar ao próximo passo, porém não aguarda que todas as *threads* para enviar os subconjuntos sejam finalizadas, pois não é necessário esperar, mas elas continuam executando, ou seja, enviando dados aos outros processos. Esse esquema de espera de finalização de *threads*, novamente, é implementado com o uso de variáveis de condição e de exclusão mútua e a implementação dessa fase é apresentada no Programa 7, função *thirdphase*.

```
void Merge(unsigned long** in1,unsigned long** in2,int N1,int N2)
{
  unsigned long* out;
  int i1,i2,o;

  out = (unsigned long*)malloc(sizeof(unsigned long)*(N1+N2));

  i1=0;i2=0;o=0;
  while( (i1 < N1) && (i2 < N2) )
  {
    if ( (*in1)[i1] < (*in2)[i2])
      out[o++] = (*in1)[i1++];
    else
      out[o++] = (*in2)[i2++];
  }
  fprintf(stdout,"%d: %d, %d: %d\n",N1,i1,N2,i2);
  if (i1 ==N1)
    memcpy(out+o,(*in2)+i2,sizeof(unsigned long)*(N2-i2));
  else if (i2 ==N2)
    memcpy(out+o,(*in1)+i1,sizeof(unsigned long)*(N1-i1));

  free(*in1);
  free(*in2);

  *in1 = out;
  *in2 = NULL;
}
```

Programa 3: Implementação do algoritmo de *Merge*

4.4 4ª Fase

No quarto e último passo os p subconjuntos ordenados são intercalados 2 a 2. O algoritmo utilizado para esse passo de ordenação é basicamente o passo de *merge* do algoritmo *MergeSort*. A implementação do passo de *merge* pode ser visto no Programa 3. Essa implementação tem ordem de complexidade de $O(m_1 + m_2 - 1)$ que é o número de comparações máximo realizado para intercalar dois subconjuntos ordenados, onde m_1 e m_2 são o tamanho dos conjuntos a serem intercalados. Porém esse algoritmo exige um espaço adicional de $O(m_1 + m_2)$. Para p subconjuntos a serem intercalados são realizadas $p - 1$ intercalações 2 a 2. A ordem de complexidade de todo o passo é $O(m \log_2 p)$, onde m é o tamanho da massa de dados envolvida. Sabe-se que $m \approx 2n/p$ no pior caso. Maiores detalhes sobre esse passo serão apresentados na seção de Experimentos (Seção 5).

Assim que os processos escravos finalizem a intercalação entre os p subconjuntos, um sinal de finalização é enviado ao processo mestre. Esse sinal contém o número de elementos que cada processador i contém de forma ordenada. O processador mestre, então, por sua vez, recebe esses valores. Ele envia um sinal de finalização para todos os escravos, a partir do instante que ele recebe o sinal de finalização do último escravo em operação. E finalmente, os processos escravos são finalizados com o recebimento do sinal do processo mestre. Após o envio do sinal de finalização para os processos escravos, o processo mestre, apresenta em qual processador escravo consta os elementos ordenados e quais elementos estão neles.

Uma ilustração da saída do algoritmo PSRS para 4 processadores com uma massa de dados de 40 valores é apresentado nas Figuras 3, 4, 5, 6, 7.

```
nilo:~/paa/tp4/src->./master 4 10
communicating with slaves
waiting for regular samples
regular samples received:
 3 11 28 31  2 13 20 24  4 10 33 36  1  8 18 25
pivots: 10 20 31
sending pivots to slaves
waiting for ending signal
finished!
Processor 1 has        10 elements [        0,         9]
Processor 2 has        10 elements [       10,        19]
Processor 3 has        11 elements [       20,        30]
Processor 4 has         9 elements [       31,        39]
full time - [  0:00:01.606025]
```

Figura 3: Resultado da executação do algoritmo PSRS para 4 processadores com uma massa de dados de 40 valores. Processador mestre nilo.

Considerações gerais sobre implementação são apresentadas na próxima subseção.

```
traira:~/paa/tp4/src->./slave
slave: ready to accept connection request
slave: ready to receive request
i'm processor 1
number of processors 4
idxFile 0

loading file of 10 elements and 40 bytes
file loaded - time [  0:00:00.001679]
 34  7 32  3 28 27 11 31 12 30
sorting set
set sorted - time [  0:00:00.000023]
  3  7 11 12 27 28 30 31 32 34
regular samples: 3 11 28 31
first phase - time [  0:00:00.001983]
pivots: 10 20 31
partitions: 2 2 4 2
second phase - time [  0:00:00.015727]
partitions size received - time: [  0:00:00.563212]
new partitions: 2 2 3 3
size of the new partition: 10
cumulative partitions 1: 0 2 4 8
cumulative partitions 2: 0 2 4 7
receiving and sending data among slave
data swaped - time [  0:00:01.012825]
  3  7134539352 2513453954413453956813453959213453961 6  0 25
third phase - time [  0:00:01.576380]
merging sortered partitions
2: 1, 2: 2
3: 1, 3: 3
4: 4, 6: 3
set sorted - time [  0:00:00.000164]
  1  2  3  4  5  6  7  8  9 10
fourth phase - time [  0:00:00.000245]
sendind signal end - size of subset
waiting for signal from master
processor end - time [  0:00:00.604647]
```

Figura 4: Resultado da executação do algoritmo PSRS para 4 processadores com uma massa de dados de 40 valores. Processador escravo traira.

```
carpa:~/paa/tp4/src->./slave
slave: ready to accept connection request
slave: ready to receive request
i'm processor 2
number of processors 4
idxFile 0

loading file of 10 elements and 40 bytes
file loaded - time [  0:00:00.002110]
 20 13 24 21 38  5 17 29 14  2
sorting set
set sorted - time [  0:00:00.000029]
  2  5 13 14 17 20 21 24 29 38
regular samples: 2 13 20 24
first phase - time [  0:00:00.002483]
pivots: 10 20 31
partitions: 2 4 3 1
second phase - time [  0:00:00.012295]
partitions size received - time: [  0:00:00.574527]
new partitions: 2 4 0 4
size of the new partition: 10
cumulative partitions 1: 0 2 6 9
cumulative partitions 2: 0 2 6 6
receiving and sending data among slave
data swaped - time [  0:00:01.003220]
 11 12134539352 25134539544134539568134539592134539616  0 25
third phase - time [  0:00:01.578136]
merging sortered partitions
2: 2, 4: 0
0: 0, 4: 0
6: 5, 4: 4
set sorted - time [  0:00:00.000219]
 11 12 13 14 15 16 17 18 19 20
fourth phase - time [  0:00:00.000323]
sendind signal end - size of subset
waiting for signal from master
processor end - time [  0:00:01.602401]
```

Figura 5: Resultado da executação do algoritmo PSRS para 4 processadores com uma massa de dados de 40 valores. Processador escravo carpa.

```
lambari:~/paa/tp4/src->./slave
slave: ready to accept connection request
slave: ready to receive request
i'm processor 3
number of processors 4
idxFile 0

loading file of 10 elements and 40 bytes
file loaded - time [  0:00:00.002983]
 36 23  9  4 33 35 10 37 40 22
sorting set
set sorted - time [  0:00:00.000056]
  4  9 10 22 23 33 35 36 37 40
regular samples: 4 10 33 36
first phase - time [  0:00:00.003787]
pivots: 10 20 31
partitions: 3 0 2 5
second phase - time [  0:00:00.007746]
partitions size received - time: [  0:00:00.574151]
new partitions: 4 3 2 2
size of the new partition: 11
cumulative partitions 1: 0 3 3 5
cumulative partitions 2: 0 4 7 9
receiving and sending data among slave
data swaped - time [  0:00:01.002771]
 27 28 30 31 24 49  4  9 10 22 23
third phase - time [  0:00:01.577259]
merging sortered partitions
4: 2, 3: 3
2: 2, 2: 0
7: 2, 4: 4
set sorted - time [  0:00:00.000200]
 21 22 23 24 25 26 27 28 29 30 31
fourth phase - time [  0:00:00.000285]
sendind signal end - size of subset
waiting for signal from master
processor end - time [  0:00:01.600106]
```

Figura 6: Resultado da executação do algoritmo PSRS para 4 processadores com uma massa de dados de 40 valores. Processador escravo lambari.

```
niger:~/paa/tp4/src->./slave
slave: ready to accept connection request
slave: ready to receive request
i'm processor 4
number of processors 4
idxFile 0

loading file of 10 elements and 40 bytes
file loaded - time [  0:00:00.001907]
 18  1 19 15  6  8 25 39 26 16
sorting set
set sorted - time [  0:00:00.000023]
  1  6  8 15 16 18 19 25 26 39
regular samples: 1 8 18 25
first phase - time [  0:00:00.002192]
pivots: 10 20 31
partitions: 3 4 2 1
second phase - time [  0:00:00.006394]
partitions size received - time: [  0:00:00.574446]
new partitions: 2 1 5 1
size of the new partition: 9
cumulative partitions 1: 0 3 7 9
cumulative partitions 2: 0 2 3 8
receiving and sending data among slave
data swaped - time [  0:00:01.002993]
 32 34134539352 2513453954413453956813453959213453961б  0
third phase - time [  0:00:01.578379]
merging sortered partitions
2: 2, 1: 0
5: 4, 1: 1
3: 3, 6: 4
set sorted - time [  0:00:00.001251]
 32 33 34 35 36 37 38 39 40
fourth phase - time [  0:00:00.001847]
sendind signal end - size of subset
waiting for signal from master
processor end - time [  0:00:01.597160]
```

Figura 7: Resultado da executação do algoritmo PSRS para 4 processadores e uma massa de 40 valores. Processador escravo niger.

4.5 Geral

Para medida de tempo de cada etapa foi utilizada uma classe especial, classe $CStackTime$, desenvolvida em C++. Essa classe utiliza a função $gettimeofday$ para adquirir o tempo de relógio tomado por uma determinada tarefa. Como os experimentos foram realizados em um ambiente dedicado e exclusivo, ou seja, somente um usuário estava conectado às máquinas, considerou que o tempo de relógio era uma medida satisfatória para o trabalho. A implementação dessa classe pode ser vista no Programa 12 anexado ao trabalho.

Várias funções (*accept*, *connect*, *bind*, *socket*, *close*) relacionadas a *sockets* foram utilizadas, no entanto, considerou-se que um detalhamento dessas funções estava fora do contexto desse trabalho. Cita-se porém o seguinte site [4] como referência para *sockets*.

5 Experimentos

Para realizar os experimentos foram utilizados quatro computadores escravos com as mesmas especificações: AMD-K6 500 MHZ, 256 MB de memória RAM, disco rígido de 60GB, e 64 Kbytes de memória cache. Esses computadores estão listados na Tabela 1 e encontram-se na sala 2011 do DCC/ICEx/UFMG. Já para o computador mestre foi utilizada a máquina nilo dessa mesma sala, no entanto qualquer outra máquina poderia ser usada, pois a carga envolvida pelo computador mestre é baixa e quase desprezível. Mas para evitar efeitos de comunicação da rede, optou-se por utilizar essa máquina que também se encontra com as máquinas escravas.

Tabela 1: Máquinas utilizadas nos experimentos

PID	Nome da Máquina
1	traira
2	carpa
3	lambari
4	niger

Foram realizados experimentos sobre o conjunto de dados de 200MB na fase sequencial. Vale lembrar que todos esses experimentos realizados nas máquinas foram feitos de forma exclusiva, ou seja, somente um usuário estava ocupando o processador naquele momento, pois o tempo de relógio dos processos poderiam ser afetados caso esta condição não fosse respeitada.

A princípio o algoritmo *QuickSort* sequencial foi executado em cada uma das máquinas escravas, conforme mostra a Tabela 2 para o conjuntos de dados de 200MB.

Observando-se os tempos reportados em cada uma das colunas dessa tabela, observa-se que o tempo de abertura de arquivo, variou de forma não sistematica na máquina niger para a Tabela 2. Isso pode ser explicado pela condição das máquinas utilizadas. Esta massa de dados pode estar localizada em setores ruins do disco rígido, pois as outras máquinas comportaram-se de forma estável para esta massa de dados. Além, disso como será visto nos próximos experimentos, a máquina niger apresentou variação no tempo de leitura somente para este arquivo, o que permite concluir que realmente a massa de dados localizada na máquina niger está gravada em setores ruins do disco rígido.

Tabela 2: Resultado dos experimentos com o algoritmo *QuickSort* sequencial para o conjunto de dados de 200MB. Tempo em segundos.

Máquina	Abertura de Arquivo	Ordenação	Total
traira	10,980585	94,705097	105,762938
carpa	10,194159	93,409150	103,686565
lambari	09,905453	93,936243	103,905663
niger	21,639137	93,781301	115,491059

Este experimento com o algorimto *QuickSort* sequencial tem como objetivo estabelecer o tempo que um único processador leva para realizar a ordenação. Este tempo é utilizado para o cálculo do *speedup*. Para tanto, utilizou-se a média do tempo das três primeiras máquinas somente: traira, carpa e lambari. Então, O tempo médio calculado é o seguinte

$$\text{Tempo Sequencial} = 104,451772 \text{ segundos}$$

Para validar o algoritmo implementado, foi realizado duas baterias de testes. Primeiro com dois processadores, onde cada um deles manipulava uma massa de dados de 100MB (aproximadamente 25MB elementos). Esse teste com dois processadores foi realizado três vezes, com o intuito de observar alguma variação na arquitetura utilizada, ou seja nas cinco máquinas (4 escravas mais uma mestre). Os resultados desses três testes podem ser observados nas Tabelas 6, 7 e 8 anexadas ao trabalho. Essas tabelas apresentam o resultado de cada execução. Um resumo desses experimentos é apresentado na Tabela 3. A primeira coluna apresenta as fases do algoritmo, a segunda mostra o tempo médio das três execuções de cada fase, a terceira coluna apresenta o desvio padrão do tempo de cada fase e a última coluna mostra o percentual que cada fase do algoritmo representa no total.

Tabela 3: Resultado dos experimentos do algoritmo PSRS com o conjunto de dados de 200MB para 2 processadores (100MB cada um). Tempo em segundos.

Fase	Tempo Médio (s)	Desvio Padrão (s)	Percentual (%)
1ª	50,042788	0,068013	78,23
2ª	0,718270	0,079641	1,12
3ª	6,181990	0,287281	9,66
4ª	6,548324	0,425764	10,24
Total	63,971186	0,064595	100,00

Experimentos também foram realizados para 4 processadores, onde cada um dos processadores manipulava uma massa de dados de 50MB (aproximadamente 12,5MB elementos). Os experimentos foram realizados três vezes, da mesma forma que para 2 processadores. Os resultados desses experimentos encontram-se nas Tabelas 9, 10 e 11, anexadas ao trabalho. Um resumo desses experimentos é apresentado na Tabela 4 da mesma forma que foi feito para a Tabela 3 para 2 processadores.

Para melhor analisar os resultados das Tabelas 3 e 4 foi plotado o gráfico da Figura 8. Através desse gráfico pode-se observar que o tempo relativo da primeira fase do algoritmo PSRS (leitura e ordenação das massas de dados) diminiu. Isto é plausível, pois a quantidade de elementos que cada processador deve carregar e ordenar é $\lceil n/p \rceil$ e diminui linearmente com o número de processadores. No entanto, este tempo está sendo tomado relativamente e esse decréscimo não justifica a queda do tempo da fase 1 com o aumento do número de processadores. Então, o que justificaria esse decréscimo? A resposta é imediata, a ordem de complexidade de tempo deste passo é $O((n/p)\log(n/p))$, e isso sim, justifica esse decréscimo no tempo de ordem logarítmica com o aumento do número de processadores.

A segunda fase, descrita pela menor curva (reta) do gráfico da Figura 8, apresenta um aumento relativo no tempo total do algoritmo de aproximadamente 3 vezes e um

Tabela 4: Resultado dos experimentos do algoritmo PSRS com o conjunto de dados de 200MB para 4 processadores.

Fase	Tempo Médio (s)	Desvio Padrão (s)	Percentual
1ª	22,578353	0,448489	69,39
2ª	1,152748	0,445557	3,54
3ª	5,343781	0,095751	16,42
4ª	3,118056	0,120822	9,58
Total	32,537934	0,003220	100,00

aumento de duas vezes tomando-se o tempo da segunda fase para 2 processadores e 4 processadores. Essa segunda fase é composta pelos seguintes passos: a seleção de p amostras regulares de cada conjunto ordenado por cada processador, o envio dessas p amostras regulares à máquina mestre, perfazendo um total de p^2 amostras regulares, ordenação dessas amostras pela máquina central, seleção dos elementos pivôs, e redistribuição desses $p-1$ elementos pivôs para os p processadores escravos. A ordem de complexidade desse passo é $O(p^2 \log p^2)$, que é a ordenação dessas p^2 amostras regulares, no entanto, aparenta que, o custo de comunicação com os p processadores seja o motivo pelo qual o tempo absoluto da fase 2 para 2 processadores tenha dobrado para 4 processadores. Então, espera-se que essa fase aumente linearmente com o número de processadores.

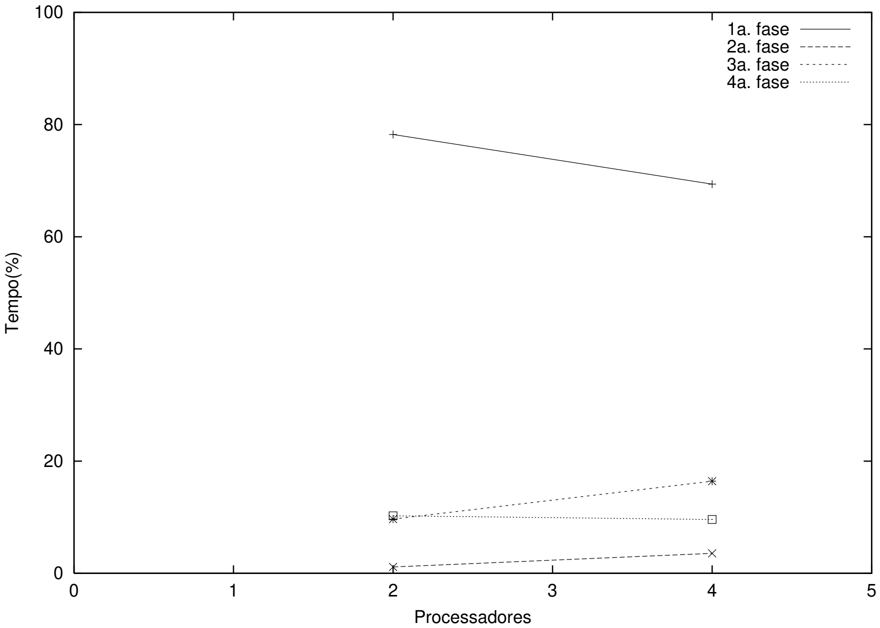

Figura 8: Percentual de cada fase do algoritmo PSRS para 2 e 4 processadores.

A terceira fase, é basicamente a troca de $\frac{(p-1)}{p}n$ elementos entre os processadores escravos, ou seja, cada processador precisa trocar com os outros processadores $p-1$ partições de n/p elementos. Portanto, o tempo exigido por essa fase é uma função do número de processadores. Essa comunicação com $p-1$ processadores é feita por várias *threads*, $p-1$ *threads*. Para 2 processadores, tem-se $n/p = 100MB$ e

$t = \frac{(p-1)}{p}n = 50MB$, para 4 processadores, tem-se $n/p = 50MB$ e $t = \frac{(p-1)}{p}n = 37.5MB$, para 8 processadores, tem-se $n/p = 25MB$ e $t = \frac{(p-1)}{p}n = 21,875$, onde t é aproximadamente a quantidade de informação a ser enviada e/ou recebida. Observa-se, então que, o tempo exigido para essa fase deve diminuir com o aumento do número de processadores.

Na quarta fase, é realizado o *merge* dos p subconjuntos ordenados. Sabe-se que um *merge* de dois subconjuntos, contendo i e j elementos, pode ser realizado em $i + j - 1$ comparações. A complexidade de tempo de um *merge* para p subconjuntos tomados dois a dois, onde cada um deles tem tamanho de n/p elementos, é $O(n \log_2 p)$. No entanto, os resultados de tempo apresentados para 2 e 4 processadores não obedece a complexidade de tempo citada. Verificando-se a decorrência dos experimentos, percebeu-se que para 2 processadores, onde cada um manipulava massas de 100MB, os processadores precisaram fazer *swap*, ou seja, usar memória virtual. Isto porque, eles tinham 200 MB de memória RAM livre inicialmente. 100 MB são usados para ordenar as massas de dados pelo algoritmo sequencial do *QuickSort*, e mais aproximadamente 100 MB são usados para fazer a troca de partições entre os processadores. Até esse instante, haviam sido alocados quase toda a memória RAM. O algoritmo de *merge* implementado precisa de um espaço extra da ordem do tamanho do conjunto a ser gerado. Justifica-se assim a não obediência do valor empírico em relação ao teórico. Esse fator de *swap* ocorrido poderia ser evitado e corrigido, mas visto as limitações físicas e de tempo para o trabalho, considera-se interessante somente reportar o ocorrido.

O objetivo dessa análise que foi realizada sobre cada fase do algoritmo PSRS é tentar prever o comportamento do algoritmo para mais processadores, visto que uma arquitetura com mais processadores não está disponível. Todavia, os experimentos reportados em [5] mostram que o *speedup* diminui a medida que o número de processadores é aumentado.

Para calcular o *speedup* alcançado pelo algoritmo é utilizado o tempo do algoritmo *QuickSort* sequencial reportado anteriormente (Tempo Sequencial $= 104,451772$) e o tempo médio levado pela máquina mestre para experimentos com 2 e 4 processadores. O tempo médio da máquina mestre é calculado sobre os experimentos reportados nas Tabelas 6, 7, 8, 9, 10, 11 em anexo. É utilizado o tempo médio da máquina mestre, pois esta espera que todos os escravos tenham findado suas tarefas para daí sim, finalizar sua execução. Ou seja, a máquina mestre contém o maior tempo de execução, pois é através dela que os processos são inicializados e só finda sua execução após as máquinas escravas terem acabado suas tarefas.

Tabela 5: Resultado dos experimentos do algoritmo PSRS para o cálculo do *speedup*. Tempo em segundos.

Processadores	Dados	Tempo Médio	Desvio Padrão
2	100MB	64,020890	0,840160
4	50MB	32,543012	0,167376

Segundo [6] (página 6), o *speedup* é definido como sendo a razão entre o tempo necessário para o algoritmo mais eficiente conhecido realizar a computação e o tempo necessário para realizar a mesma computação em uma máquina incorporando *pipelining* e/ou paralelismo.

$$S = \frac{\text{Tempo do Algoritmo Sequencial Usando 1 Processador}}{\text{Tempo do Algoritmo Paralelo usando } p \text{ processadores}}$$

Embora essa definição seja informal, mas, no entanto, intuitiva, ela será utilizada aqui. Então, pode-se calcular o *speedup* para 2 e 4 processadores, para os experimentos realizados, da seguinte forma

$$S(2) = \frac{104,451722}{64,020890} \approx 1,6315$$

$$S(4) = \frac{104,451722}{32,543012} \approx 3,2097,$$

respectivamente.

No gráfico da Figura 9 podem ser vistas curvas com o *speedup* ideal e o alcançado pela implementação aqui realizada. Uma comparação visual do gráfico apresentado na Figura 9 com o gráfico da Figura 10-31 de [6] permite concluir que a implementação aqui realizada é fiel ao reportado em [6].

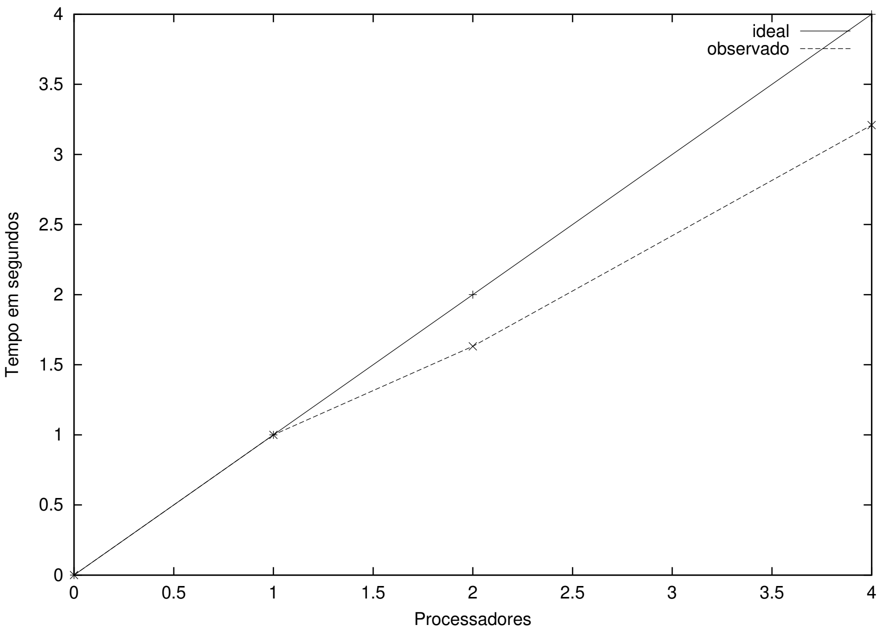

Figura 9: *Speedup* alcançado pela implementação do algoritmo PSRS.

A Eficiência obtida por um algoritmo paralelo executando com p processadores é, segundo [6] (página 80), o *speedup* divido por p, ou seja:

$$E = \frac{S}{p}$$

E, portanto, pode-se calcular a eficiência, E, para 2 e 4 processadores, para os experimentos realizados, da seguinte forma.

$$E(2) = \frac{1,6315}{2} \approx 0.8158$$

$$E(2) = \frac{3,2096}{4} \approx 0.8024$$

Resumindo, o algoritmo paralelo de ordenação PSRS implementado, alcançou uma eficiência levemente superior a 80%.

6 Conclusões

Após a confecção deste trabalho, chega-se as seguintes conclusões. O algoritmo paralelo implementado para ordenação apresentou eficiência levemente superior a 80% para 2 e 4 processadores. Os experimentos reportados aqui, relativos ao *speedup*, não são muito distintos dos apresentados em [5] e [6].

Em [5] é apresentado alguns experimentos. Foi medido o desvio relativo do tamanho entre a maior partição intercalada na fase quatro do algoritmo e o tamanho médio dessa partição. Esses experimentos mostram que se os elementos são selecionados de uma distribuição "aleatoriamente" uniforme, o maior tamanho de partição é, frequentemente, não mais do que um pequeno percentual maior que n/p elementos, o tamanho médio da partição.

Uma discussão do algoritmo *Quicksort* sobre o ponto de vista de paralelismo de dados e de paralelismo de controle pode ser feita. O algoritmo PSRS implementado neste trabalho é um algoritmo de paralelismo de dados, onde cada processador realiza o algoritmo sequential mais eficiente (*QuickSort*) nos dados compartilhados. Existe um algoritmo para ordenação de massas de dados baseado em paralelismo de controle e também sobre o algoritmo *QuickSort*. Este algoritmo é descrito na Seção 10.5.1 em [6].

O algoritmo PSRS alcança *speedp* melhor e é mais escalável que o algoritmo *QuickSort* paralelo da Seção 10.5.1. (QLQ) de [6]. O *speedup* alcançado pelo algoritmo QLQ é limitado pelo tempo tomado para realizar o passo inicial de particionamento, quando nem todos os processadores estão ativos, restrição imposta pelo paralelismo de controle. De uma forma geral, os algoritmos de paralelismo de dados são mais escaláveis que os algoritmos de controle de dados, isto porque o grau de paralelismo de controle é geralmente constante, independente do tamanho do problema, enquanto que o nível de paralelismo de dados é uma função crescente do tamanho do problema. A grande maioria dos problemas encontrados em boas referências de computação paralela ([6],[3]) apresentam soluções de paralelismo de dados.

7 Anexos

7.1 Tabelas dos Experimentos

Nessa subção estão anexadas as tabelas que resultaram dos experimentos sobre as as massas de dados disponíveis. Foram realizados 6 experimentos, sendo 3 deles com 4 processadores e mais 3 com 2 processadores. Tanto os experimentos com 4 ou 2 processadores foram realizados sobre a mesma massa de dados total, ou seja, nos experimentos de 4 processadores, cada processador manipulou 50 MB de dados, enquanto que nos experimentos de 2 processadores, cada processador manipulou 100MB de dados, perfazendo um total, em ambas as situações de 200MB de dados.

Tabela 6: Tabela com os resultados do primeiro experimento com dois processadores escravos (traira e carpa) manipulando cada um deles 100MB de dados, e um total de 200MB de dados. Tempo em segundos.

```
=== master:nilo
full time     - time [  0:01:03,400804]
=== slave:traira
i'm processor 1
file loaded   - time [  0:00:05,030865]
set sorted    - time [  0:00:45,000561]
first phase   - time [  0:00:50,033190]
second phase  - time [  0:00:00,668628]
third phase   - time [  0:00:05,996754]
fourth phase  - time [  0:00:06,293874]
sum of four   - time [  0:01:02,992446]
processor end - time [  0:01:03,400383]
=== slave:carpa
i'm processor 2
file loaded   - time [  0:00:04,659142]
set sorted    - time [  0:00:45,378317]
first phase   - time [  0:00:50,039221]
second phase  - time [  0:00:00,661329]
third phase   - time [  0:00:06,379633]
fourth phase  - time [  0:00:06,309131]
sum of four   - time [  0:01:03,389314]
processor end - time [  0:01:03,396948]
```

Tabela 7: Tabela com os resultados do segundo experimento com dois processadores escravos (traira e carpa) manipulando cada um deles 100MB de dados, e um total de 200MB de dados. Tempo em segundos.

```
=== master:nilo
full time     - time [  0:01:04,977067]
=== slave:traira
i'm processor 1
file loaded   - time [  0:00:04,876259]
set sorted    - time [  0:00:45,039752]
first phase   - time [  0:00:49,917707]
second phase  - time [  0:00:00,986500]
third phase   - time [  0:00:05,943045]
fourth phase  - time [  0:00:06,154042]
sum of four   - time [  0:01:03,001294]
processor end - time [  0:01:04,975764]
=== slave:carpa
i'm processor 2
file loaded   - time [  0:00:04,876055]
set sorted    - time [  0:00:45,362280]
first phase   - time [  0:00:50,240031]
second phase  - time [  0:00:00,660017]
third phase   - time [  0:00:06,215569]
fourth phase  - time [  0:00:07,849287]
sum of four   - time [  0:01:04,964904]
processor end - time [  0:01:04,972218]
```

Tabela 8: Tabela com os resultados do terceiro experimento com dois processadores escravos (traira e carpa) manipulando cada um deles 100MB de dados, e um total de 200MB de dados. Tempo em segundos.

```
=== master:nilo
full time     - time [  0:01:03,684799]
=== slave:traira
i'm processor 1
file loaded   - time [  0:00:04,571046]
set sorted    - time [  0:00:44,972667]
first phase   - time [  0:00:49,545421]
second phase  - time [  0:00:01,109664]
third phase   - time [  0:00:05,606637]
fourth phase  - time [  0:00:07,415851]
sum of four   - time [  0:01:03,677573]
processor end - time [  0:01:03,684633]
=== slave:carpa
i'm processor 2
file loaded   - time [  0:00:04,638365]
set sorted    - time [  0:00:45,353403]
first phase   - time [  0:00:49,993388]
second phase  - time [  0:00:00,664520]
third phase   - time [  0:00:06,560183]
fourth phase  - time [  0:00:06,389736]
sum of four   - time [  0:01:03,607827]
processor end - time [  0:01:03,681418]
```

Tabela 9: Tabela com os resultados do primeiro experimento com quatro processadores escravos (traira, carpa, lambari e niger) manipulando cada um deles 50MB de dados, e um total de 200MB de dados. Tempo em segundos.

```
=== master:nilo
full time     - time [  0:00:32,720438]
=== slave:traira
i'm processor 1
file loaded   - time [  0:00:01,019577]
set sorted    - time [  0:00:22,231636]
first phase   - time [  0:00:23,252994]
second phase  - time [  0:00:00,494114]
third phase   - time [  0:00:04,781144]
fourth phase  - time [  0:00:03,155611]
sum of four   - time [  0:00:31,683863]
processor end - time [  0:00:32,719086]
=== slave:carpa
i'm processor 2
file loaded   - time [  0:00:01,103626]
set sorted    - time [  0:00:21,170589]
first phase   - time [  0:00:22,275967]
second phase  - time [  0:00:01,471864]
third phase   - time [  0:00:05,559354]
fourth phase  - time [  0:00:03,400934]
sum of four   - time [  0:00:32,708119]
processor end - time [  0:00:32,716834]
=== slave:lambari
i'm processor 3
file loaded   - time [  0:00:01,074409]
set sorted    - time [  0:00:21,223234]
first phase   - time [  0:00:22,299430]
second phase  - time [  0:00:01,441478]
third phase   - time [  0:00:05,248916]
fourth phase  - time [  0:00:03,011732]
sum of four   - time [  0:00:32,001556]
processor end - time [  0:00:32,714281]
=== slave:niger
i'm processor 4
file loaded   - time [  0:00:01,015596]
set sorted    - time [  0:00:21,261532]
first phase   - time [  0:00:22,278829]
second phase  - time [  0:00:01,462030]
third phase   - time [  0:00:05,173398]
fourth phase  - time [  0:00:03,090147]
sum of four   - time [  0:00:32,004404]
processor end - time [  0:00:32,711838]
```

Tabela 10: Tabela com os resultados do segundo experimento com quatro processadores escravos (traira, carpa, lambari e niger) manipulando cada um deles 50MB de dados, e um total de 200MB de dados. Tempo em segundos.

```
=== master:nilo
full time     - time [  0:00:32,520668]
=== slave:traira
i'm processor 1
file loaded   - time [  0:00:01,013880]
set sorted    - time [  0:00:22,236042]
first phase   - time [  0:00:23,251707]
second phase  - time [  0:00:00,496266]
third phase   - time [  0:00:05,388067]
fourth phase  - time [  0:00:03,093343]
sum of four   - time [  0:00:32,229383]
processor end - time [  0:00:32,519520]
=== slave:carpa
i'm processor 2
file loaded   - time [  0:00:01,112987]
set sorted    - time [  0:00:21,372865]
first phase   - time [  0:00:22,487656]
second phase  - time [  0:00:01,258403]
third phase   - time [  0:00:04,898822]
fourth phase  - time [  0:00:03,322957]
sum of four   - time [  0:00:31,967838]
processor end - time [  0:00:32,516407]
=== slave:lambari
i'm processor 3
file loaded   - time [  0:00:01,043623]
set sorted    - time [  0:00:21,222143]
first phase   - time [  0:00:22,267502]
second phase  - time [  0:00:01,473505]
third phase   - time [  0:00:05,612869]
fourth phase  - time [  0:00:03,000741]
sum of four   - time [  0:00:32,354617]
processor end - time [  0:00:32,513366]
=== slave:niger
i'm processor 4
file loaded   - time [  0:00:00,921405]
set sorted    - time [  0:00:21,258254]
first phase   - time [  0:00:22,181419]
second phase  - time [  0:00:01,557349]
third phase   - time [  0:00:05,812612]
fourth phase  - time [  0:00:02,951868]
sum of four   - time [  0:00:32,503248]
processor end - time [  0:00:32,511276]
```

Tabela 11: Tabela com os resultados do terceiro experimento com quatro processadores escravos (traira, carpa, lambari e niger) manipulando cada um deles 50MB de dados, e um total de 200MB de dados. Tempo em segundos.

```
=== master:nilo
full time     - time [  0:00:32,387930]
=== slave:traira
i'm processor 1
file loaded   - time [  0:00:00,918435]
set sorted    - time [  0:00:22,293787]
first phase   - time [  0:00:23,213926]
second phase  - time [  0:00:00,496469]
third phase   - time [  0:00:05,585412]
fourth phase  - time [  0:00:03,083844]
sum of four   - time [  0:00:32,379651]
processor end - time [  0:00:32,386625]
=== slave:carpa
i'm processor 2
file loaded   - time [  0:00:01,124283]
set sorted    - time [  0:00:21,250709]
first phase   - time [  0:00:22,376747]
second phase  - time [  0:00:01,331881]
third phase   - time [  0:00:05,356312]
fourth phase  - time [  0:00:03,136519]
sum of four   - time [  0:00:32,201459]
processor end - time [  0:00:32,384138]
=== slave:lambari
i'm processor 3
file loaded   - time [  0:00:01,035028]
set sorted    - time [  0:00:21,735403]
first phase   - time [  0:00:22,772140]
second phase  - time [  0:00:00,932079]
third phase   - time [  0:00:05,449023]
fourth phase  - time [  0:00:02,999613]
sum of four   - time [  0:00:32,152855]
processor end - time [  0:00:32,382114]
=== slave:niger
i'm processor 4
file loaded   - time [  0:00:01,012562]
set sorted    - time [  0:00:21,267603]
first phase   - time [  0:00:22,281919]
second phase  - time [  0:00:01,417534]
third phase   - time [  0:00:05,259438]
fourth phase  - time [  0:00:03,169367]
sum of four   - time [  0:00:32,128258]
processor end - time [  0:00:32,379723]
```

7.2 Códigos fontes

Nesta seção estão anexados os códigos fonte em C/C++ do programa utilizado para desenvolver o trabalho.

```
CC = g++
CFLAGS = -g -c -Wall
LFLAGS = -lm -lpthread -lnsl

ARQOBJS =

all : master slave

run : all

master: $(ARQOBJS) master.c util.c sort.c
	$(CC) $(CFLAGS) ./util.c       $(ARQOBJS) -o ./util.o
	$(CC) $(CFLAGS) ./sort.c       $(ARQOBJS) -o ./sort.o
	$(CC) $(CFLAGS) ./master.c     $(ARQOBJS) -o ./master.o
	$(CC) $(LFLAGS) ./sort.o ./util.o ./master.o -o ./master

slave: $(ARQOBJS) slave.c  util.c sort.c
	$(CC) $(CFLAGS) ./util.c       $(ARQOBJS) -o ./util.o
	$(CC) $(CFLAGS) ./sort.c       $(ARQOBJS) -o ./sort.o
	$(CC) $(CFLAGS) ./slave.c      $(ARQOBJS) -o ./slave.o
	$(CC) $(LFLAGS) ./sort.o ./util.o ./slave.o -o ./slave

mastersel: $(ARQOBJS) mastersel.c

clean:
	/bin/rm -f *.o

cleanall:
	/bin/rm -f *.o master slave
```

Programa 4: Arquivo makefile

```
#ifndef _TDA_H_
#define _TDA_H_

struct strArgs
{
  int sockfd;
  int slaveId;
};

#define TCP_PORT 1024

char ips[4][14]={"150.164.6.108" , /* traira  */
                 "150.164.6.102",  /* carpa   */
                 "150.164.6.103",  /* lambari */
                 "150.164.6.65" }; /* niger   */


int tcp_ports[4][4]={{  -1,2012,2013,2014},
                     {2021,  -1,2023,2024},
                     {2031,2032,  -1,2034},
                     {2041,2042,2042,  -1}};
```

```
int idx[] = {10,50,100};

char processorfiles[4][3][30] = {
{"../data/data-10-1.bin","/var/tmp/paa/dados50MB-1.bin",
"/var/tmp/paa/dados100MB-1.bin"},
{"../data/data-10-2.bin","/var/tmp/paa/dados50MB-2.bin",
"/var/tmp/paa/dados100MB-2.bin"},
{"../data/data-10-3.bin","/var/tmp/paa/dados50MB-3.bin",""},
{"../data/data-10-4.bin","/var/tmp/paa/dados50MB-4.bin",""}};

#endif
```

Programa 5: Arquivo tda.h

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <pthread.h>

#include "tda.h"
#include "util.h"
#include "sort.h"
#include "stacktime.h"

CStackTime oStackTime;

int P,N;
unsigned long *rs = NULL;
unsigned long *prs = NULL;
int *subset = NULL;
int count_rs = 0,count_end=0;

pthread_mutex_t mutex_rs,mutex_end;
pthread_cond_t  cond_rs ,cond_end ;

void* ending(void* voidargs)
{
  int slaveId,sockfd;
  struct strArgs *schArgs;

  schArgs = (struct strArgs*) voidargs;
  sockfd  = schArgs->sockfd;
  slaveId = schArgs->slaveId;

  /* receiving the size of each subset */
  RecvInteger(sockfd,&(subset[slaveId-1]));

  pthread_mutex_lock(&mutex_end);
  count_end++;
  if (count_end==P)
    pthread_cond_signal(&cond_end);
  pthread_mutex_unlock(&mutex_end);
```

```
  pthread_exit(NULL);
}

void *schedule(void *voidargs)
{
  int sockfd,slaveId,i;
  struct strArgs *pArgs;

  /* Unpack arguments. */
  pArgs = (struct strArgs *)voidargs;
  sockfd  = pArgs->sockfd;
  slaveId = pArgs->slaveId;

  /* Receive the response from the slave. */
  RecvString(sockfd,(void*)&(rs[(slaveId-1)*P]),P*sizeof(unsigned long));

  pthread_mutex_lock(&mutex_rs);
  count_rs++;
  if (count_rs==P)
  {
    pthread_cond_signal(&cond_rs);
    fprintf(stdout,"regular samples received: ");
    for(i=0;i<P*P;i++)
      fprintf(stdout,"%2d ",rs[i]);
    fprintf(stdout,"\n");
  }
  pthread_mutex_unlock(&mutex_rs);

  pthread_exit(NULL);
}

void sequential(int argc,char** argv)
{
    int size;
    FILE* pF;
    int i;
    char sTmp[BUFSIZ];
    unsigned long *data;
    void (*pSort)(unsigned long*,int);

  oStackTime.Start(); // Full Time

  oStackTime.Start(); // Loading file
  fprintf(stdout,"loading file of ");
  if ( (pF=fopen(argv[2],"rb")) == NULL)
  {
    fprintf(stderr,"error while opening file!\n");
    exit(2);
  }
  size = filesize(pF);
  N = size / sizeof(unsigned long);
  fprintf(stdout,"%d elements - %d bytes\n",N,size);
  data = (unsigned long*)malloc(sizeof(unsigned long)*N);
  fread(data,sizeof(unsigned long),N,pF);
  fclose(pF);

  oStackTime.Finish();
```

```
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"file loaded - time: %s\n",sTmp);

  if (argc < 4)
    pSort = QuickSort;
  else
  {
    switch(atoi(argv[3]))
    {
      case 0: pSort = QuickSort;  break;
      case 1: pSort = HeapSort;   break;
      case 2: pSort = ShellSort;  break;
      case 3: pSort = SelectSort; break;
      case 4: pSort = InsertSort; break;
    }
  }

  oStackTime.Start(); // Sorting
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"sorting set\n");

  (*pSort)(data,N);

  oStackTime.Finish();
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"set sorted - time %s\n",sTmp);

  if (N <= 100)
  {
    for(i=0;i<N;i++)
      fprintf(stdout,"%d ",data[i]);
    fprintf(stdout,"\n");
  }

  free(data);

  oStackTime.Finish();
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"full time - %s\n",sTmp);
}

void parallel(int argc, char** argv)
{
  pthread_t* proc;
  int procId,idxFile;
  struct sockaddr_in serv_addr;
  struct strArgs* schArgs;
  int i,iCt;
  int* sockfd;
  char sTmp[BUFSIZ];

  oStackTime.Start(); // Full Time

  rs     = (unsigned long*)malloc(sizeof(unsigned long)*P*P);
  prs    = (unsigned long*)malloc(sizeof(unsigned long)*(P-1));
  subset = (int*)          malloc(sizeof(int)          *(P));

  N = atoi(argv[2]);
```

```
for(i=0;i<5;i++)
  if (N== idx[i])
  {
      idxFile = i;
      break;
  }

if (i == 5)
{
  fprintf(stderr,"index not found!\n");
  exit(1);
}

proc = (pthread_t*)malloc(sizeof(pthread_t)*P);
sockfd = (int*)malloc(sizeof(int*)*P);
schArgs = (struct strArgs*)malloc(sizeof(struct strArgs)*P);

fprintf(stdout,"communicating with slaves\n");
/* Set host addresses */
/* Connect to slaves */
for (procId=1;procId<=P;procId++)
{
  /* Fill in the structure "serv_addr" with the address of the
     slave that we want to connect with */
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr(ips[procId-1]);
  serv_addr.sin_port = htons(TCP_PORT);
  /* Open a TCP socket (an Internet stream socket). */
  if ( (sockfd[procId-1] = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    fprintf(stderr,"master: can't open stream socket\n");
  }
  /* Connect to the slave. */
  while (connect(sockfd[procId-1],
      (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0);
  if ( sockfd[procId-1] < 0 )
  {
    fprintf(stderr,"master: can't connect to slave\n");
  }

  /* Send the processor identification */
  SendInteger(sockfd[procId-1],procId);

  /* Send the number of processors envolved */
  SendInteger(sockfd[procId-1],P);

  /* Send the index file in the table */
  SendInteger(sockfd[procId-1],idxFile);
}

fprintf(stdout,"waiting for regular samples\n");
pthread_mutex_init(&mutex_rs,NULL);
pthread_cond_init (&cond_rs  ,NULL);

pthread_mutex_lock(&mutex_rs);

/* Create threads. */
```

```
for (procId=1;procId<=P;procId++)
{
  /* Set arguments for the function applied by the thread. */
  schArgs[procId-1].sockfd = sockfd[procId-1];
  schArgs[procId-1].slaveId = procId;

  /* Create the thread. */
  pthread_create(&proc[procId-1], NULL,schedule,&schArgs[procId-1]);
}

pthread_cond_wait(&cond_rs,&mutex_rs);
pthread_mutex_unlock(&mutex_rs);

pthread_mutex_destroy(&mutex_rs);
pthread_cond_destroy (&cond_rs);

/* ordenar regular samples */
InsertSort(rs,P*P);

/* escolher p-1 pivos */
fprintf(stdout,"pivots: ");
for(i=1;i<P;i++)
{
  fprintf(stdout,"%2lu ",rs[i*P-1+(int)(P/2)]);
  prs[i-1] = rs[i*P-1+(int)(P/2)];
}
fprintf(stdout,"\n");

/* enviando p-1 pivos para os escravos */
fprintf(stdout,"sending pivots to slaves\n");
for(procId=1;procId<=P;procId++)
  SendString(sockfd[procId-1],prs,sizeof(unsigned long)*(P-1));

fprintf(stdout,"waiting for ending signal\n");
pthread_mutex_init(&mutex_end,NULL);
pthread_cond_init (&cond_end ,NULL);

pthread_mutex_lock(&mutex_end);

/* Create threads. */
for (procId=1;procId<=P;procId++)
{
  /* Set arguments for the function applied by the thread. */
  schArgs[procId-1].sockfd = sockfd[procId-1];
  schArgs[procId-1].slaveId = procId;

  /* Create the thread. */
  pthread_create(&proc[procId-1], NULL,ending,&(schArgs[procId-1]) );
}

pthread_cond_wait(&cond_end,&mutex_end);
pthread_mutex_unlock(&mutex_end);

fprintf(stdout,"finished!\n");

/* sending an ending signal to slaves */
for(procId=1;procId<=P;procId++)
  SendInteger(sockfd[procId-1],1);
```

```
  pthread_mutex_destroy(&mutex_end);
  pthread_cond_destroy (&cond_end);

  iCt = 0;
  for (procId=0;procId<P;procId++)
  {
    fprintf(stdout,"Processor %d has %10d elements [%10d,%10d]\n"
        ,procId+1,subset[procId],iCt,iCt+subset[procId]-1);
    iCt += subset[procId];
  }

  for (procId=0;procId<P;procId++)
    close(sockfd[procId]);

  free(rs);
  free(prs);
  free(subset);

  free(proc);
  free(sockfd);
  free(schArgs);

  oStackTime.Finish();
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"full time - %s\n",sTmp);

  return;
}

int main(int argc, char *argv[])
{
  if (argc  < 3)
  {
    fprintf(stderr,"use: %s num_processors data_size!\n",argv[0]);
    exit(1);
  }

  P = atoi(argv[1]);
  if ( (P!= 1) && (P != 2) && (P != 4) )
  {
    fprintf(stderr,"the number of processors must be 1, 2 or 4!\n");
    exit(1);
  }

  if (P == 1)
    sequential(argc,argv);
  else if ( (P == 2) || (P == 4) )
    parallel(argc,argv);

  return 0;
}
```

Programa 6: Arquivo master.c

```
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
```

```
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <math.h>

#include "tda.h"
#include "util.h"
#include "sort.h"
#include "stacktime.h"

char        sTmp[BUFSIZ];
CStackTime oStackTime;
int         idxFile,size;
int         mastersockfd;

pthread_t* proc;
unsigned long *data,*rs,*prs,**data2;
int *part,*part2,*cpart,*cpart2,**slavesock;
int myId;
int P,N,N2;

pthread_mutex_t mutex_part,mutex_recv;
pthread_cond_t  cond_part ,cond_recv;

int count_part=0,count_recv=0;

void PrintSet(unsigned long* set,int n)
{
    int i;
    for(i=0;i<n;i++)
        fprintf(stdout,"%3d",set[i]);
    fprintf(stdout,"\n");
}

void* Listen(void* voidargs)
{
  int sockfd,slaveId;
  socklen_t clilen;
  struct sockaddr_in cli_addr, serv_addr;
  struct strArgs *pArgs;

  /* Unpack arguments. */
  pArgs  = (struct strArgs *)voidargs;
  slaveId = pArgs->slaveId;

  /* Open a TCP socket (an Internet stream socket). */
  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
  {
    fprintf(stderr,"listen: can't open stream socket\n");
  }

  /* Bind our local address so that the master can send to us */
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
  serv_addr.sin_port = htons(tcp_ports[slaveId-1][myId-1]);
  while(bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0);
  listen(sockfd, 5);

  /* Wait for a connection from the other slave process. */
  clilen = sizeof(cli_addr);
  while ( (slavesock[slaveId-1][myId-1] = accept(sockfd,
     (struct sockaddr *) &cli_addr, &clilen)) <0);
  if (slavesock[slaveId-1][myId-1] < 0)
  {
    fprintf(stderr,"listen: accept error\n");
  }

  RecvInteger(slavesock[slaveId-1][myId-1],&(part2[slaveId-1]));

  pthread_mutex_lock(&mutex_part);
  count_part++;
  if (count_part == 2*(P-1))
    pthread_cond_signal(&cond_part); //,&mutex_part);
  pthread_mutex_unlock(&mutex_part);

  pthread_exit(0);
}

void* Talk(void* voidargs)
{
  int     slaveId;
  struct sockaddr_in serv_addr;
  struct strArgs *pArgs;

  /* Unpack arguments. */
  pArgs = (struct strArgs *)voidargs;
  slaveId = pArgs->slaveId;

  /* Set host addresses */
  /* Connect to other slaves */

  /* Fill in the structure "serv_addr" with the address of the
    slave that we want to connect with */
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr(ips[slaveId-1]);
  serv_addr.sin_port = htons(tcp_ports[myId-1][slaveId-1]);
  /* Open a TCP socket (an Internet stream socket). */
  if ( (slavesock[myId-1][slaveId-1] = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    fprintf(stderr,"talk: can't open stream socket\n");
  /* Connect to the other slave. */

  while (connect(slavesock[myId-1][slaveId-1],
     (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0);
  if (slavesock[myId-1][slaveId-1] < 0)
    fprintf(stderr,"talk: can't connect to slave (%d)\n",slaveId);

  SendInteger(slavesock[myId-1][slaveId-1], part [slaveId-1]);

  pthread_mutex_lock(&mutex_part);
  count_part++;
  if (count_part == 2*(P-1))
```

```
    pthread_cond_signal(&cond_part); //,&mutex_part);
  pthread_mutex_unlock(&mutex_part);

  pthread_exit(0);
}

void* SendData(void* voidargs)
{
  int slaveId;
  struct strArgs *schArgs;

  // Unpack arguments.
  schArgs = (struct strArgs *)voidargs;
  slaveId = schArgs->slaveId;

  if ( (slaveId != myId) && (part [slaveId-1] > 0) )
  {
    SendString(slavesock[myId-1][slaveId-1]
        ,&(data [cpart [slaveId-1]]),sizeof(unsigned long)*part [slaveId-1]);
  }

  pthread_exit(0);
}

void* RecvData(void* voidargs)
{
  int slaveId;
  struct strArgs *schArgs;

  // Unpack arguments.
  schArgs = (struct strArgs *)voidargs;
  slaveId = schArgs->slaveId;

  if ( (slaveId != myId) && (part2[slaveId-1] > 0) )
    RecvString(slavesock[slaveId-1][myId-1]
        ,data2[slaveId-1],sizeof(unsigned long)*part2[slaveId-1]);
  else
    // data are in myself
    memcpy(data2[slaveId-1]
        ,&(data[cpart [slaveId-1]]),sizeof(unsigned long)*part[slaveId-1]);

  pthread_mutex_lock(&mutex_recv);
  count_recv++;
  if (count_recv == P)
    pthread_cond_signal(&cond_recv); //,&mutex_recv);
  pthread_mutex_unlock(&mutex_recv);

  pthread_exit(0);
}

void opensocket(int& newsockfd)
{
  int sockfd;
  socklen_t clilen;
  struct sockaddr_in cli_addr, serv_addr;

  /* Open a TCP socket (an Internet stream socket). */
  if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
```

```
  {
    fprintf(stderr,"slave: can't open stream socket\n");
  }

  /* Bind our local address so that the master can send to us */
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  serv_addr.sin_port = htons(TCP_PORT);
  while(bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0);
  listen(sockfd, 5);

  /* Wait for a connection from the master process. */
  clilen = sizeof(cli_addr);
  printf("slave: ready to accept connection request\n");
  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
  if (newsockfd < 0)
  {
    fprintf(stderr,"slave: accept error\n");
  }
  printf("slave: ready to receive request\n");
  close(sockfd);
}

void firstphase(int argc,char** argv)
{
    int procId;
  FILE* pF;
  void (*pSort)(unsigned long*,int);

  if (argc == 1)
    pSort = QuickSort;
  else
  {
    switch(atoi(argv[1]))
    {
      case 0: pSort = QuickSort;  break; // QuickSort
      case 1: pSort = HeapSort;   break; // HeapSort
      case 2: pSort = ShellSort;  break; // ShellSort
      case 3: pSort = SelectSort; break; // SelectSort
      case 4: pSort = InsertSort; break; // InsertSort
        break;
      default:
        fprintf(stdout,"use: %s 0, for QuickSort\n" ,argv[0]);
        fprintf(stdout,"use: %s 1, for HeapSort\n"  ,argv[0]);
        fprintf(stdout,"use: %s 2, for ShellSort\n" ,argv[0]);
        fprintf(stdout,"use: %s 3, for SelectSort\n",argv[0]);
        fprintf(stdout,"use: %s 4, for InsertSort\n",argv[0]);
        exit(2);
    }
  }

  oStackTime.Start(); /* first phase */

  oStackTime.Start(); /* loading data */
  fprintf(stdout,"loading file of ");
  if ( (pF = fopen(processorfiles[myId-1][idxFile],"rb")) == NULL)
  {
```

```
    fprintf(stderr,"error while opening file!\n");
    exit(1);
  }
  size = filesize(pF);
  N = size/sizeof(unsigned long);
  data = (unsigned long*)malloc(sizeof(unsigned long)*N);
  fprintf(stdout,"%d elements and %d bytes\n",N,size);
  fread((void*)data,sizeof(unsigned long),N,pF);
  fclose(pF);

  oStackTime.Finish();
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"file loaded - time %s\n",sTmp);

  if (N <= 20) PrintSet(data,N);

  oStackTime.Start();
  fprintf(stdout,"sorting set\n");
  (*pSort)(data,N);

  oStackTime.Finish();
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"set sorted - time %s\n",sTmp);

  if (N <= 20) PrintSet(data,N);

  /* choosing regular samples */
  rs = (unsigned long*)malloc(sizeof(unsigned long)*P);
  fprintf(stdout,"regular samples: ");
  for(procId=0;procId<P;procId++)
  {
    rs[procId] = data[procId*N/P];
    fprintf(stdout,"%lu ",rs[procId]);
  }
  fprintf(stdout,"\n");

  oStackTime.Finish();
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"first phase - time %s\n",sTmp);
}

void secondphase()
{
  int i,procId,iCt,iCt2;

  oStackTime.Start(); /* second phase */

  /* sending regular samples */
  SendString(mastersockfd,(void*)rs,sizeof(unsigned long)*P);

  /* receiving p-1 pivot */
  prs = (unsigned long*)malloc(sizeof(unsigned long)*(P-1));
  RecvString(mastersockfd,(void*)prs,sizeof(unsigned long)*(P-1));
  fprintf(stdout,"pivots: ");
  for(i=0;i<P-1;i++)
    fprintf(stdout,"%lu ",prs[i]);
  fprintf(stdout,"\n");
```

```
/* counting the number of elements by partition */
part   = (int*)malloc(sizeof(int)*P);
part2  = (int*)malloc(sizeof(int)*P);
cpart  = (int*)malloc(sizeof(int)*P);
cpart2 = (int*)malloc(sizeof(int)*P);
iCt=0;iCt2=0;procId=0;
for(i=0;i<N && procId<P-1;i++)
{
  if (data[i] <= prs[procId])
    iCt++;
  else
  {
    part[procId] = iCt;
    i--;
    iCt2 += iCt;
    iCt = 0;
    procId++;
  }
}
part[procId] = N-iCt2;

fprintf(stdout,"partitions: ");
for(procId=0;procId<P;procId++)
  fprintf(stdout,"%d ",part[procId]);
fprintf(stdout,"\n");

oStackTime.Finish();
oStackTime.GetStringTime(sTmp);
fprintf(stdout,"second phase - time %s\n",sTmp);
}

void thirdphase()
{
  struct strArgs* pArgs;
  int procId;

  oStackTime.Start(); /* third phase */

  oStackTime.Start(); // receiving partition size

  pthread_mutex_init(&mutex_part,NULL);
  pthread_cond_init (&cond_part ,NULL);

  slavesock = (int**)malloc(sizeof(int*)*P);
  for(procId=0;procId<P;procId++)
    slavesock[procId] = (int*)malloc(sizeof(int)*P);
  proc = (pthread_t*)malloc(sizeof(pthread_t)*P);
  pArgs = (struct strArgs*)malloc(sizeof(struct strArgs)*P);

  pthread_mutex_lock(&mutex_part);
  for(procId=1;procId<=P;procId++)
  {
    /* Set arguments for the function applied by the thread. */
    pArgs[procId-1].slaveId = procId;

    if (procId == myId)
    {
      part2[myId-1] = part[myId-1];
```

```
    continue;
  }
  /* Create the thread. */
  /* myId listen from procId - accept*/
  pthread_create(&(proc[procId-1]), NULL, Listen,&(pArgs[procId-1]));
  /* myid talk to procId */
  pthread_create(&(proc[procId-1]), NULL, Talk   ,&(pArgs[procId-1]));
}

pthread_cond_wait(&cond_part,&mutex_part);
pthread_mutex_unlock(&mutex_part);

oStackTime.Finish();
oStackTime.GetStringTime(sTmp);
fprintf(stdout,"partitions size received - time: %s\n",sTmp);

fprintf(stdout,"new partitions: ");
N2=0;
for(procId=0;procId<P;procId++)
{
  fprintf(stdout,"%d ",part2[procId]);
  N2 += part2[procId];
}
fprintf(stdout,"\n");
fprintf(stdout,"size of the new partition: %d\n",N2);

/* allocating memory for the new partition */
data2 = (unsigned long**)malloc(sizeof(unsigned long*)*P);
for(procId=0;procId<P;procId++)
  data2[procId] = (unsigned long*)malloc(sizeof(unsigned long)*part2[procId]);

cpart [0] = 0;
cpart2[0] = 0;
for(procId=1;procId<P;procId++)
{
  cpart [procId] = cpart [procId-1]+part [procId-1];
  cpart2[procId] = cpart2[procId-1]+part2[procId-1];
}

fprintf(stdout,"cumulative partitions 1: ");
for(procId=0;procId<P;procId++)
  fprintf(stdout,"%d ",cpart[procId]);
fprintf(stdout,"\n");

fprintf(stdout,"cumulative partitions 2: ");
for(procId=0;procId<P;procId++)
  fprintf(stdout,"%d ",cpart2[procId]);
fprintf(stdout,"\n");

/* receiving and sending data among slaves */
oStackTime.Start();

fprintf(stdout,"receiving and sending data among slave\n");

pthread_mutex_init(&mutex_recv,NULL);
pthread_cond_init (&cond_recv  ,NULL);

pthread_mutex_lock(&mutex_recv);
```

```
  for(procId=1;procId<=P;procId++)
  {
     /* Set arguments for the function applied by the thread. */
    pArgs[procId-1].slaveId = procId;

     /* Create the thread. */
    pthread_create(&(proc[procId-1]), NULL,SendData,&(pArgs[procId-1]));
    pthread_create(&(proc[procId-1]), NULL,RecvData,&(pArgs[procId-1]));
  }

  pthread_cond_wait(&cond_recv,&mutex_recv);
  pthread_mutex_unlock(&mutex_recv);

  oStackTime.Finish();
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"data swaped - time %s\n",sTmp);

  if (N <= 20) PrintSet(data2[0],N2);

  oStackTime.Finish();
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"third phase - time %s\n",sTmp);

  free(pArgs);
}

void fourthphase()
{
  int procId,procId2;

  oStackTime.Start(); /* fourth phase */

  oStackTime.Start();
  fprintf(stdout,"merging sortered partitions\n");

  for(procId=0;procId<(int)(log(P)/log(2));procId++)
    for(procId2=0;procId2<(int)(P/pow(2,procId));procId2+=(int)(pow(2.,procId+1)) )
    {
      Merge(&(data2[procId2]),&(data2[procId2+(int)(pow(2.,procId))])
          ,part2[procId2],part2[procId2+(int)(pow(2.,procId))]);
      part2[procId2] += part2[procId2+(int)(pow(2.,procId))];
      part2[procId2+(int)(pow(2.,procId))] = 0;
    }

  oStackTime.Finish();
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"set sorted - time %s\n",sTmp);

  if (N <= 20) PrintSet(data2[0],N2);

  oStackTime.Finish();
  oStackTime.GetStringTime(sTmp);
  fprintf(stdout,"fourth phase - time %s\n",sTmp);
}

int main(int argc, char *argv[])
{
  int procId,procId2;
```

```
opensocket(mastersockfd);

oStackTime.Start(); // full time

/* Receive the processor identification */
RecvInteger(mastersockfd,&myId);
fprintf(stdout,"i'm processor %d\n",myId);

/* Receibe the number of processors envolved */
RecvInteger(mastersockfd,&P);
fprintf(stdout,"number of processors %d\n",P);

/* Receive the index file in the table */
RecvInteger(mastersockfd,&idxFile);
fprintf(stdout,"idxFile %d\n\n",idxFile);

/* first phase */
firstphase(argc,argv);

/* second phase */
secondphase();

/* third phase */
thirdphase();

/* fourth phase */
fourthphase();

fprintf(stdout,"sendind signal end - size of subset\n");
SendInteger(mastersockfd,N2);
fprintf(stdout,"waiting for signal from master\n");
RecvInteger(mastersockfd,&procId);

oStackTime.Finish();
oStackTime.GetStringTime(sTmp);
fprintf(stdout,"processor end - time %s\n",sTmp);

for(procId=0;procId<P;procId++)
  for(procId2=0;procId2<P;procId2++)
    if (procId != procId2) close(slavesock[procId][procId2]);

close(mastersockfd);

free(proc);

for(procId=0;procId<P;procId++)
  free(slavesock[procId]);
free(slavesock);

free(part);
free(part2);
free(cpart);
free(cpart2);
free(prs);
free(rs);
free(data2[0]);
free(data2);
```

```
  free(data);

  fflush(NULL);

  exit(0);
}
```

Programa 7: Arquivo slave.c

```
#ifndef _QUICKSORT_H_
#define _QUICKSORT_H_

void SelectSort(unsigned long* v,int );
void InsertSort(unsigned long* v,int );
void ShellSort (unsigned long* v,int );
void QuickSort (unsigned long* v,int );
void HeapSort  (unsigned long* v,int );
void Merge     (unsigned long**,unsigned long**, int,int);

#endif
```

Programa 8: Arquivo sort.h

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "sort.h"

void SelectSort(unsigned long *A, int n)
{
  int i, j, Min;
  unsigned long x;
  for (i = 0; i < n - 1; i++)
  {
    Min = i;
    for (j = i + 1; j < n; j++)
      if (A[j] < A[Min])
          Min = j;

          x = A[Min];
    A[Min] = A[i];
    A[i] = x;
  }
}

void InsertSort(unsigned long *A, int n)
{
  int i, j;
  unsigned long x;
  for (i = 1; i < n; i++)
  {
    x = A[i];
    j = i - 1;
    while (x < A[j] && j >=0)
    {
      A[j+1] = A[j];
```

```
      j--;
    }
    A[j+1] = x;
  }
}

void ShellSort(unsigned long *A, int n)
{
  int i, j;
  int h = 1;
  unsigned long x;
  do h = h * 3 + 1; while (h < n);

  do
  {
    h /= 3;
    for (i = h ; i < n; i++)
    {
      x = A[i];
      j = i;
      while (A[j - h] > x)
      {
        A[j] = A[j - h];
        j -= h;
        if (j <= h) goto L999;
      }
L999:
      A[j] = x;
    }
  } while (h != 1);
}

void Particao(int Esq, int Dir, int *i, int *j, unsigned long *A)
{
  unsigned long x, w;
  *i = Esq; *j = Dir;
  x = A[(*i + *j) / 2];    /* obtem o pivo x */
  do
  {
    while (x > A[*i] && *i < Dir) (*i)++;
    while (x < A[*j] && *j > Esq) (*j)--;
    if (*i <= *j)
    {
      w = A[*i]; A[*i] = A[*j]; A[*j] = w;
      (*i)++; (*j)--;
    }
  } while (*i <= *j);
}

void Ordena(int Esq, int Dir, unsigned long *A)
{
  int i, j;

  Particao(Esq, Dir, &i, &j, A);
  if (Esq < j) Ordena(Esq, j, A);
  if (i < Dir) Ordena(i, Dir, A);
}
```

```
void QuickSort(unsigned long *A,int n)
{
  Ordena(0,n-1,A);
}

void Refaz(int Esq, int Dir, unsigned long *A)
{
  int i = Esq;
  int j;
  unsigned long x;
  j = i * 2;
  x = A[i];
  while (j <= Dir)
  {
    if (j < Dir)
    {
      if (A[j] < A[j+1])
        j++;
          }
    if (x >= A[j])
      goto L999;

    A[i] = A[j];
    i = j; j = i * 2;
  }
L999:
  A[i] = x;
}

void Constroi(unsigned long *A, int n)
{
  int Esq;
  Esq = n / 2 + 1;
  while (Esq > 0)
  {
    Esq--;
    Refaz(Esq, n, A);
  }
}

void HeapSort(unsigned long *A, int n)
{
  int Esq, Dir;
  unsigned long x;

  Constroi(A, n);    /* constroi o heap */
  Esq = 0; Dir = n-1;
  while (Dir > 0)
  { /* ordena o vetor */
    x = A[0];
    A[0] = A[Dir];
    A[Dir] = x;
    Dir--;
    Refaz(Esq, Dir, A);
  }
}

void Merge(unsigned long** in1,unsigned long** in2,int N1,int N2)
```

```
{
  unsigned long* out;
  int i1,i2,o;

  out = (unsigned long*)malloc(sizeof(unsigned long)*(N1+N2));

  i1=0;i2=0;o=0;
  while( (i1 < N1) && (i2 < N2) )
  {
    if ( (*in1)[i1] < (*in2)[i2])
      out[o++] = (*in1)[i1++];
    else
      out[o++] = (*in2)[i2++];
  }
  fprintf(stdout,"%d: %d, %d: %d\n",N1,i1,N2,i2);
  if (i1 == N1)
    memcpy(out+o,(*in2)+i2,sizeof(unsigned long)*(N2-i2));
  else if (i2 == N2)
    memcpy(out+o,(*in1)+i1,sizeof(unsigned long)*(N1-i1));

  free(*in1);
  free(*in2);

  *in1 = out;
  *in2 = NULL;
}
```

Programa 9: Arquivo sort.c

```
#ifndef _UTIL_H_
#define _UTIL_H_

#include <stdio.h>

size_t filesize(FILE *stream);
void SendInteger(int sock,int  inteiro);
void RecvInteger(int sock,int* inteiro);
void SendString (int sock,void*,int tamanho);
void RecvString (int sock,void*,int tamanho);

#endif
```

Programa 10: Arquivo util.h

```
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>

#include "util.h"

size_t filesize(FILE *stream)
{
  long curpos;
  size_t length;
```

```
  curpos = ftell(stream);
  fseek(stream,0L, SEEK_END);
  length = ftell(stream);
  fseek(stream,curpos,SEEK_SET);
  return length;
}

void SendInteger(int sock,int inteiro)
{
    char msg[BUFSIZ];
    sprintf(msg,"%d",inteiro);
    if (send(sock, (void *)msg, BUFSIZ, MSG_NOSIGNAL) < 0)
    {
      fprintf(stderr,": can't send msg\n");
      close(sock);
      exit(1);
    }
}

void RecvInteger(int sock,int* inteiro)
{
    char msg[BUFSIZ];
    if (recv(sock, (void *)msg, BUFSIZ, MSG_WAITALL|MSG_NOSIGNAL) < 0)
    {
        fprintf(stderr,": can't receive\n");
        close(sock);
        exit(1);
    }
    *inteiro = atoi(msg);
}

void SendString(int sock,void* v,int tamanho)
{
    if (send(sock, (void *)v, tamanho, MSG_NOSIGNAL) < 0)
    {
      fprintf(stderr,"master: can't send msg\n");
      close(sock);
      exit(1);
    }
}

void RecvString(int sock,void* v,int tamanho)
{
  if (recv(sock, (void *)v, tamanho, MSG_WAITALL|MSG_NOSIGNAL) < 0)
  {
    fprintf(stderr,"slave: can't receive\n");
    close(sock);
    exit(1);
  }
}
```

Programa 11: Arquivo util.c

```
#ifndef _STACKTIME_H_
#define _STACKTIME_H_

#include <time.h>
```

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

class CTime
{
public:
  double tempo;
  CTime* pNext;

  CTime()
  {
    struct timeval val;
    struct timezone zone;
    tempo = gettimeofday(&val,&zone);
    tempo = val.tv_sec+(double)(val.tv_usec)/1000000.;
    pNext = NULL;
  };
  CTime(double _tempo)
  {
    tempo = _tempo;
    pNext = NULL;
  };
};

class CStackTime
{
private:
  CTime* pFirst;
  double dStart,dFinish;
  double dElapTime;

  int      iCt;
public:
  CStackTime()
  {
    pFirst = NULL;
    iCt = 0;
  };

  int GetSize()
  {
    return iCt;
  };

  double GetElapTime()
  {
    return dElapTime;
  };

  bool Push(CTime pt)
  {
    CTime* ptAux;
    if ( (ptAux = new CTime(pt.tempo)) == NULL)
      return false;

    iCt++;
```

```
  if (pFirst != NULL)
    ptAux->pNext = pFirst;
  else
    ptAux->pNext = NULL;
  pFirst = ptAux;

  return true;
};

bool Pop (CTime& pt)
{
  CTime *ptAux;

  if (pFirst ==NULL)
    return false;

  pt.tempo = pFirst->tempo;

  ptAux = pFirst;
  pFirst = pFirst->pNext;
  delete ptAux;

  iCt--;

  return true;
};

void RemoveAll()
{
  CTime *ptAux;
  for(ptAux = pFirst;ptAux != NULL;ptAux = ptAux->pNext,iCt--)
    delete ptAux;
};

void Start()
{
  CTime oTempo;
  Push(oTempo);
}

void Finish()
{
  CTime oTimeFinish,oTimeStart;

  Pop(oTimeStart);

  dFinish = oTimeFinish.tempo;
  dStart  = oTimeStart .tempo;

  dElapTime = (double)(dFinish - dStart);
}

char* GetStringTime(char* sTempo)
{
  sprintf(sTempo,"[%3d:%02d:%02d.%06d]"
    ,(int)((dFinish-dStart)/3600)
    ,(int)((dFinish-dStart)/60  )%60
    ,(int)((dFinish-dStart)/1   )%60
```

```
        ,(int)((dFinish-dStart)*1000000)%1000000);
      return sTempo;
   }
};

#endif
```

Programa 12: Arquivo stacktime.h

Referências

[1] Posix threads programming - http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/main.html#joining. *site* de internet, Junho 2004. consultado em 26/06/04 às 17h.

[2] T. Anderson, D. Culler, and D. Patterson. A case for now (network of workstations). *IEEE Micro*, 15(1):54–64, 1995.

[3] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, Pearson Education Limited, Essex CM20 2JE, England, 2003.

[4] Brian "Beej"Hall. Beej's guide to network programming - http://www.ecst.csuchico.edu/ beej/guide/net/. *site* de internet, Junho 2004. consultado em 26/06/04 às 17h.

[5] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versability of parallel sorting by regular sampling. Technical Report 91-06, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, March 1992.

[6] Michael Jay Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, New York, USA, 1994.

[7] Nivio Ziviani. *Projeto de Algoritmos com implementações em Pascal e C*. Pioneira Thomson Learnig, São Paulo - SP - Brasil, 2a. edition, 2004. revisada e ampliada.