Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

PROJETO E ANÁLISE DE ALGORITMOS

Quarto Trabalho. disponível em:

`http://www.dcc.ufmg.br/~horacio/PaaTP4`

Horácio Antonio Braga Fernandes de Oliveira

Professor - Nivio Ziviani

Belo Horizonte
27 de junho de 2004

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

1 Introdução

Diversos fatores têm apoiado o interesse crescente, em especial nas últimas décadas, no que diz respeito a ALGORITMOS e COMPUTAÇÃO PARALELA. Dentre estes fatores, pode-se citar, como os principais:

- *O advento e avanço de computadores de alta-performance e com paralelismo massivo:* Computadores dotados de uma grande quantidade de processadores, podendo variar de 2 até uma quantidade igual ou mesmo superior a 65.536 processadores [2].

- *Redução dos custos dos computadores pessoais e até mesmo de servidores de pequeno e médio porte:* possibilitando, assim, a criação de redes de computadores em grande escala e compostos de computadores com processadores descentes e com capacidade de processamento muitas vezes comparado a alguns servidores.

Um outro motivo para se utilizar computação paralela também pode-se encontrar na necessidade de se aproveitar recursos do computadores que já estão obsoletos. Ao formar uma rede com tais equipamentos, é possível de se obter resultados próximos a computadores atuais.

Mudando um pouco dos motivos de disponibilidade de equipamentos, tem-se que a necessidade por alto desempenho vem aumentado cada dia mais. Algumas áreas que necessitam de alto desempenho incluem a *previsão do tempo*, o *processamento de imagens*, *simulações* e etc.

Como pode-se ver, muitos são os motivos para se estudar COMPUTAÇÃO PARALELA. Em tais ambientes, tipos diferentes dos algoritmos seqüenciais tradicionais são utilizados: os ALGORITMOS PARALELOS. Algoritmos estes que serão estudados neste trabalho e que serão aplicados ao PROBLEMA DE ORDENAÇÃO, em especial na solução deste problema através da utilização do QUICKSORT.

1.1 Histórico do Problema

Levou-se mais de 20 anos para que computadores paralelos saíssem dos laboratórios para a área comercial. Daniel Slotnick, da Universidade de Illinois elaborou duas máquinas paralelas: Solomon, construído por Westinghouse Electric Company por volta de 1960, e ILLIAC IV, montada na Burroughs Corporation por volta de 1970. Na Universidade de Carnegie-Mellon, dois computadores paralelos foram construídos por volta de 1970: C.mmp e Cm*. No início de 1980, pesquisadores da Caltech construíram o Cosmic Cube, o ancestror dos multicomputadores construídos por Ametek, Intel e nCUBE [1].

Mas ainda não foi nos anos 80 que os computadores paralelos tornaram-se comercialmente disponíveis. Um estudo do crescimento de performance em diversas classes de computadores mostra porque microcomputadores baseados em processamento paralelo finalmente tornaram-se disponíveis. O aumento de performance dos minicomputadores, mainframes e dos tradicionais supercomputadores foi crescendo até 20% ao ano, enquanto que o aumento de performance do s microcomputadores era em média 35% ao ano.

Porque que a performance dos microcomputadores aumentou mais rapidamente do que dos outros tipos de processadores? A performance de um processados pode ser melhorada através de avanços na arquitetura ou na tecnologia. Melhorias na arquitetura podem aumentar a quantidade de trabalho executada por ciclo de instrução. Por volta do meio dos anos 70, houve avanços fundamentais de arquitetura: paralelismo de bits na memória, aritmética com paralelismo de bits, memória *cache*, canais, procura adiante de instruções, *pipeline* de instruções, unidades multifuncionais, e *pipeline* de dados. Tais avanços foram incorporados à arquitetura dos supercomputadores. Depois disso, melhorias na performance dos processadores individuais resultaram na redução do tempo de ciclo de instruções. Este último tem se tornado cada vez mais difícil, já que a velocidade de circuitos eletrônicos são limitados à velocidade da luz.

Em contraste, os microcomputadores tinham a possibilidade de um ganho muito maior de performance, já que eles eram lentos e tinham muito o que caminhar. Inicialmente os microcomputadores não tinham incorporado todos os avanços de arquitetura que já estavam presentes nos supercomputadores, e sua velocidade de clock era muito mais lenta. Avanços na arquitetura, em conjunto com uma redução no tempo de ciclo, acabaram por levar a um aumento impressionante no ganho de performance desta classe de computadores.

Esta convergência na performance relativa entre microcomputadores e os tradicionais supercomputadores têm tornado viável o desenvolvimento de computadores comercialmente disponíveis constituídos de dezenas, centenas ou mesmo milhares de microprocessadores. No que diz respeito a eficiência, computadores paralelos baseados em microprocessadores como o Paragon XP/S da Intel, MP-2 da MasPar e CM-5 da Thinking Machines têm excedido a velocidade os tradicionais supercomputadores com um processador, como o Y/MP da Cray e o SX-3 da NEC.

Aumentar a força latente dos computadores baseados em microprocessadores de paralelismo massivo, entretanto, requer o desenvolvimento de arquiteturas, sistemas operacionais, linguagens de programação e algoritmos. Escolhas pobres de desenvolvimento podem gerar performance baixa. Este trabalho, como será visto adiante, estuda o desenvolvimento eficiente de algoritmos paralelos com a implementação no problema de ordenação utilizando o Quicksort.

1.2 Objetivos do Trabalho

Conforme mencionado, o objetivo principal deste trabalho é o estudo, implementação e análise dos Algoritmos Paralelos aplicados ao Quicksort.

Além de toda a parte teórica, foi implementado o algoritmo *Parallel Sorting by Regular Sampling (PSRS)* conforme descrito na seção 10.5.3 do livro do Quinn [1], utilizando uma máquina paralela MIMD (ver Seção 2 e 4). A máquina paralela MIMD é do tipo rede de estações de trabalho (*NOW - Network of Workstations*)(Anderson, Culler e Patterson, 1995), utilizando mecanismos de comunicação e sincronização.

Os seguintes itens foram executados e descritos neste trabalho:

- *Listagem dos programas implementados*: foi feito também uma descrição sucinta das estruturas de dados utilizadas bem como dos mecanismos de comunicação, sincronização e de paralelismos utilizadas. Ver Seção 5 e Anexo A.

- *Listagem dos testes executados:* execução passo-a-passo da massa de dados da figura 10-29 do livro do Quinn [1]. Ver Seção 5, em especial a 5.3.5.

- *Speedup obtido pela implementação:* utilizando 2 e 4 máquinas. Foi gerado massas de teste usando um gerador de números aleatórios. Os resultados foram discutidos. Ver Seção 6.

- *Estudo comparativo do algoritmo implementado com a versão tradicional de Quicksort Paralelo da seção 10.5.1 [1]:* Tal estudo comparativo foi realizado e encontra-se na Seção 4.

- *Discussão da implementação do QuickSort Paralelo sob o ponto de vista dos modelos de paralelismo de dados e de controle:* Dicussão essa que faz parte da Seção 4.

2 Algoritmos Paralelos

2.1 Alguns Conceitos Básicos

Para que o assunto seja melhor entendido, e também para evitar má interpretação das palavras, algumas definições são necessárias. A seguir será definido alguns termos para que a sua definição considerada neste trabalho possa ser conhecida. Estes conceitos foram baseados em parte nos estudos realizados por Quinn [1].

- *Processamento Paralelo:* Processamento de informações que enfatiza a manipulação paralela de elementos de dados pertencentes a um ou mais processos para a solução de um único problema.

- *Computador Paralelo:* Um computador com múltiplos processadores capaz de fazer um processamento paralelo. Estamos considerando também, como um computador paralelo, uma *NOW - Network of Workstations*, uma máquina paralela MIMD, proposto por Anderson, Culler e Patterson em 1995, que utiliza uma rede de estações de trabalho para o processamento paralelo.

- *Supercomputador:* Computador de uso geral capaz de resolver problemas individuais em velocidades extremamente rápidas, comparadas com outros computadores desenvolvidos na mesma época. Claro que "extremamente rápida"é um termo relativo. Desta forma, supercomputadores sempre existiram. Todos os supercomputadores contemporâneos são computadores paralelos. Alguns apresentam pequenos números de processadores extremamente poderosos; outros são constituídos de uma grande quantidade de número de microprocessadores.

- *Concorrência:* Número de operações realizadas no mesmo tempo.

- *Throughput: Throughput* de um dispositivo é o número de resultados que este produz em unidade de tempo. Existem diversas formas de se aumentar o *throughput* de um dispositivo. A velocidade que este opera pode ser aumentada ou a concorrência pode ser aumentada.

- *Pipeline: Pipeline* é uma forma de se aumentar a concorrência em um computador. Uma computação com *pipeline* é dividida em um número de passos, chamados segmentos ou estágios. Cada segmento trabalha em velocidade máxima em uma determinada parte do computador. A saída de um segmento serve como entrada do próximo segmento. Se todos os segmentos trabalham na mesma velocidade, uma vez que o pipe está cheio, a taxa de trabalho do *pipeline* é igual à soma da taxa de trabalho dos segmentos.

- *Paralelismo de Dados:* É a utilização de unidades múltiplas funcionais para aplicar a mesma operação simultaneamente aos elementos de um conjunto de dados. Um aumento de k no número de unidades funcionais leva a um aumento de k no aumento do *throughput* do sistema, se não tiver nenhum overhead associado com o aumento de paralelismo.

- *Speedup:* O *speedup* conseguido por um algoritmo paralelo rodando em p processadores é a divisão entre o tempo necessário do algoritmo seqüencial mais eficiente para resolver o problema (rodando em um processador) e o tempo necessário para resolver o mesmo problema em uma máquina utilizando paralelismo com p processadores.

- *Eficiência:* A eficiência de um algoritmo paralelo rodando em p processadores é o *speedup* dividido por p.

2.2 Organização de Processadores

A Organização de Processadores define métodos de conectar os processadores em computadores paralelos. Uma Organização de Processadores pode ser representada por um grafo onde os nós representam os processadores e as arestas a comunicação entre estes processadores. Alguns fatores que influem na eficiência desta organização incluem:

1. *Diâmetro:* É a maior distância entre dois processadores. Um diâmetro menor é melhor, visto que este diâmetro impõe um limite inferior nos problemas resolvidos que necessitam de comunicação.

2. *Tamanho da Bisessão:* Número mínimo de arestas que precisam ser removidas para dividir o grafo de processadores em duas metades.

3. *Número de arestas por processadores:* É importante que o número de arestas por processador seja constante independente do tamanho da rede para que a organização dos processadores seja mais escalável.

4. *Tamanho máximo de uma aresta:* Por motivos de escalabilidade, é melhor que as arestas entre os processadores seja constante, independente do seu tamanho, o que leva a uma solução de espaço tridimensional.

Será discutido agora os tipos de organização de processadores.

- *Mesh Network:* Neste tipo de organização, os processadores são arranjados em um reticulado q-dimensional. A comunicação é permitida apenas com os vizinhos. Logo, processadores internos comunicam com apenas $2q$ outros processadores.

- *Binary Tree Network:* Nesta topologia, os $2^k - 1$ processadores são organizados em uma estrutura de árvore binária completa de altura $k - 1$. Um processador comunica-se com apenas três outros processadores, no máximo.

- *Hypertree Networks:* Da mesma forma que o anterior, este é organizado em forma de árvore. A diferença é que aqui estamos lidando com uma árvore n-ária completa, onde n é o grau da árvore e d a sua altura.

- *Pyramid Networks:* É uma tentativa de se combinar as vantagens de se utilizar a topologia de *Mesh Networks* e a de *Trre Networks*. Uma *Pyramid Tree* de tamanho k^2 é uma árvore 4 -ária de altura $\log_2 k$ com a adição de arestas entre processadores de forma que em cada nível é formada um reticulado na base 2D.

- *Hypercube Networks:* Também chamada de *n-cube Network*. Formalmente, esta rede consiste de 2^k nós formando um *hypercube*. Os nós são identificados $0, 1, ..., 2^k - 1$; dois nós são adjacentes se suas identificações diferem em exatamente uma posição de bit.

- *Shuffle-Exchenge Networks:* Consiste de $n = 2^k$ nós, numeradas $0, 1, ..., n-1$ e dois tipos de conexões: *shuffle* e *exchange*. Conexões *Exchange* ligam pares de nós de números que diferem no seu último bit significativo. Conexões *Suffle* conectam nós i com nós $2imod(n-1)$, com a exceção que o nó $n-1$ é ligado a ele mesmo.

- Outros tipos de organização existem, mas começam a fugir do escopo deste trabalho.

2.3 Esquemas de Classificação de Arquiteturas de Computadores

A taxonomia de Flynn é o melhor esquema de classificação conhecida para computadores seriais e paralelos. Da multiplicidade de instruções e de dados, quatro classes de computadores podem ser derivadas:

- SISD - Single Instruction, Single Data: A maioria dos computadores seriais caem nesta categoria. Mesmo que a execução de instruções possam ser feitas utilizando *pipeline*, computadores desta categoria podem decodificar apenas uma única instrução por tempo. Computadores SISD podem ter múltiplas unidades funcionais trabalhando juntas, mas estas estão sob a direção de apenas uma unidade de controle.

- *SIMD - Single Instruction, Multiple Data:* Vetores de processadores (*Processors Arrays*) caem nesta categoria. Um vetor de processadores executa um único bloco de instrução, mas contém um número de unidades processadoras aritméticas, cada uma capaz de pegar e manipular seus próprios dados. Desta forma, tem-se diversos processadores sincronizados.

- *MISD - Multiple Instruction, Single Data:* Neste caso, teríamos um único dado e múltiplas instruções. Não existem computadores que se encaixam nesta categoria.

- *MIMD - Multiple Instruction, Multiple Data:* Esta categoria contem a maioria dos sistemas multiprocessados. Os primeiros sistemas multiprocessados eram construídos para aumentar o *throughput* dos sistemas operacionais multiprogramados, e haviam pouca iteração entre os processadores. Neste trabalho estamos lidando com computadores multiprocessados construídos para o processamento paralelo que utilizam iterações eficientes entre seus processadores. Esta categoria se divide em dois tipos: *multiprocessadores*, onde uma memória compartilhada oferece a comunicação entre os processadores e *multicomputadores*, conde a comunicação e a sincronização ocorre via mensagens, pois cada processo tem sua própria memória.

2.4 PRAM - *Parallel Random Access Machine*

PRAM é um modelo teórico de Computação Paralela. Este modelo consiste de uma unidade de controle, uma memória global e um conjunto ilimitado de processadores, cada um com sua memória privada. Embora processadores ativos possam executar instruções idênticas, cada processador tem um identificador único, e o valor de seu índice pode ser utilizado para habilitar ou desabilitar este processador ou influenciar a área de memória que o mesmo acessa.

Uma computação PRAM começa com a entrada armazenada na memória global e apenas um único processador ativo. Durante cada passo da computação, um processador ativo e habilitado pode ler um valor de uma memória privada ou global, executar uma única operação, e escrever em uma memória local ou global. Alternativamente, durante um passo da computação, um processador pode ativar outros processadores. Todos os processadores ativos e habilitados devem executar a mesma instrução, porém em diferentes locais da memória. A computação termina quando o último processador pára.

Diversos modelos PRAM diferem em como eles manipulam os conflitos de leitura e de escrita:

- *EREW - Exclusive Read, Exclusive Write:* Conflitos de leitura ou escrita não são permitidos.
- *CREW - Concurrent Read, Exclusive Write:* Permite a leitura ao mesmo tempo. Mas conflitos de escrita não são permitidos. Este é o modelo PRAM padrão.
- *CRCW - Concurrent Read, Concurrent Write:* Permite a leitura e a escrita ao mesmo tempo por processos diferentes. Quando há um conflito de escrita, existem três formas de se lidar com isso: *Comum*, todos os processos escrevendo no mesmo local devem estar escrevendo o mesmo valor, *arbitrária*, a escolha de qual processo irá escrever na memória é aleatória e *prioritária*, onde o processo com menor índice vence.

2.5 Complexidade em Computação Paralela

Os principais pontos a ser observado no que diz respeito à complexidade de algoritmos seqüenciais são o tempo de execução e a utilização de espaço. Estas métricas também são importantes em Algoritmos Paralelos, mas nós precisamos também nos preocupar com outros recursos. Um importante recurso é o número de processadores. Existem problemas que são naturalmente seqüenciais, e eles não podem ser "paralelizados" mesmo diante de um número infinito de processadores disponíveis. A maioria dos outros problemas, entretanto, pode ser paralelizados em um determinado nível. Quanto mais processadores forem utilizados - até um certo limite - mais rapidamente o algoritmo se torna. É importante estudar os limites dos Algoritmos Paralelos e também ser capaz de caracterizar os problemas que possuem soluções paralelas bastante rápidas. Uma vez que o número de processadores é limitado, é também importante utilizar tais recursos eficazmente.

Outro importante ponto é a comunicação entre os processos. Geralmente tempo muito maior é necessário para trocar dados entre dois processadores do que para executar uma simples operação nos dados. Além disso, alguns processadores podem estar perto um do outro, e, por outro lado, outros processadores podem estar longe. Logo, é muito importante minimizar a comunicação e também utilizar este recurso de forma eficaz.

Outro problema importante é a sincronização, o que é um problema ainda maior para Algoritmos Paralelos que rodam em máquinas independentes conectadas por alguma rede de comunicação [3].

3 Introdução ao Quicksort

O QUICKSORT é um algoritmo do tipo *divisão e conquista* que ordena a seqüência dividindo-a recursivamente em menores subseqüências. Assuma que uma seqüência de n elementos que precisa ser ordenado esteja armazenada em um vetor $A[1...n]$. QUICKSORT consiste de dois passos: dividir e conquistar. Durante o passo de divisão, a seqüência $A[1...m]$ é particionada em duas subseqüências $A[q...s]$ e $A[s+1...r]$ de forma que cada elemento da primeira subseqüência seja menor ou igual a cada elemento da segunda subseqüência. Durante o passo de conquista, a subseqüência é ordenada recursivamente aplicando o QUICKSORT. Já que as subseqüências $A[s...s]$ e $A[s+1...r]$ estão ordenadas e a primeira subseqüência tem elementos menores que o segundo, então a seqüência completa está ordenada.

3.1 Particionamento

Como que a seqüência $A[q...r]$ é particionada em duas partes - uma com todos os elementos menores que os elementos da outra? Este passo é geralmente realizado através da seleção de um elemento x do vetor $A[q...r]$ e usando este elemento para particionar a seqüência $A[q...r]$ em duas partes - uma com elementos menores ou iguais a x e a outra com elementos maiores que x. Este elemento x é chamado de *pivot*.

O algoritmo do *Quicksort* é mostrado no Código 1.

Código 1 Algoritmo do QUICKSORT tradicional.

```
Procedure QUICKSORT(A, q, r)
Begin
  if q < r then
  begin
    x:=A[q];
    s:=q;
    for i:=q + 1 to r do
      if A[i] ≤ x then
      begin
        s:=s+1;
        swap(A[q], A[s]);
      end if
    swap(A[q], A[s]);
    QUICKSORT(A, q, s);
    QUICKSORT(A, s+1, r);
  end if
end QUICKSORT
```

3.2 Complexidade

A complexidade de particionar uma seqüência de tamanho k é $\Theta(k)$. A performance do QUICKSORT é muito influenciada pela forma como é feita a partição da seqüência. Considere o caso onde a seqüência de tamanho k é dividida de forma pobre em duas subseqüências de tamanho 1 e $k-1$. O tempo de execução neste caso é dada pela recorrência $T(n) = T(n-1) + \Theta(n)$, onde a solução é $T(n) = \Theta(n^2)$. Alternativamente, considere o caso em que a seqüência é dividida de forma boa, em duas seqüências de tamanho quase iguais a $\lfloor \frac{k}{2} \rfloor$ e $\lceil \frac{k}{2} \rceil$ elementos. Neste caso, o tempo de execução é dada pela recorrência $T(n) = T(n-1) + \Theta(n)$, cuja solução é $T(n) = \Theta(n \log n)$. O segundo particionamento mostrado leva a uma solução ótima. Embora o Quicksort pode ter complexidade de $O(n^2)$ no pior caso, a sua complexidade média é significantemente melhor; o número médio de comparações/trocas necessitado pelo Quicksort para ordenar uma entrada de seqüência randômica é $1.4n \log n$, o que é assintoticamente ótimo.

Existem diversas formas de se selecionar o *pivot*. Por exemplo, o *pivot* pode ser a mediana de um número pequeno de elementos da seqüência, ou este pode ser elementos selecionados randomicamente. Algumas estratégias de se selecionar o *pivot* tem vantagens sobre outras para certas seqüências de entrada.

4 Ordenação Utilizando Computação Paralela

Desenvolver Algoritmos Paralelos é fácil quando existe um algoritmo PRAM ótimo para o problema onde as iterações do processador casam com o da arquitetura utilizada. Infelizmente, nós não temos este luxo quando estamos tratando de Ordenação. Conforme será mostrado adiante, o custo ótimo de um algoritmo de ordenação utilizando máquinas PRAM é de $\Theta(\log n)$ com n elementos processadores, porém sua enorme constante de proporcionalidade o torna impraticável de ser utilizado (Leighton, 1984).

Nesta Seção serão estudados dois Algoritmos de Ordenação Paralela baseados no Quicksort - um dos algoritmos seqüenciais de ordenação mais utilizados na prática.

Serão estudados o Quicksort Paralelo Tradicional (ver Seção 4.3) e o Parallel Sorting by Regular Sampling - Ordenação Paralela Utilizando Amostras Regulares (ver Seção 5.3).

Antes, será feito um breve estudo sobre o limite inferior de ordenação para a resolução deste problema.

4.1 Limite Inferior para a Ordenação Paralela

Muller e Preparata (1975) demonstrou que o modelo PRAM pode executar uma ordenação em tempo logarítmico. Na verdade, o algoritmo requer $\Theta(\log n)$ tempo para acionar n^2 processadores e $\Theta(1)$ para realizar a ordenação.

Para modelos de computação paralela mais realistas, os parágrafos a seguir mostram o limite inferior [1].

Em uma arquitetura de vetor de processadores organizados como um *one - dimension mesh* (ver Seção 2.2), se nós precisamos distribuir os elementos justamente e um para cada processador, então a complexidade de qualquer algoritmo de ordenação é $\Theta(n)$.

Assumindo que n elementos precisam ser ordenados em um vetor de processadores organizados como *two-dimension mesh*, e que antes e após a ordenação os elementos precisam estar distribuídos igualmente, um elemento por processador. Então o limite inferior de complexidade de tempo de qualquer algoritmo de ordenação é $\Theta(\sqrt{n})$.

Assumindo que $n = 2^k$ elementos estão para ser ordenados em um vetor de processadores organizados como um *shuffle-exchenge network* (ver Seção 2.2). Neste caso, o limite inferior para qualquer algoritmo de ordenação é $\Theta \log n$.

4.2 *Parallel Sorting by Regular Sampling*

O algoritmo de Ordenação Paralela Utilizando Amostras Regulares - Parallel Sorting by Regular Sampling[1] foi desenvolvido por Li *et al.* em 1992 e tem se mostrado bastante eficiente para uma grande variedade de arquiteturas MIMD.

Este algoritmo apresenta quatro fases:

[1] Abrev.: PSRS

1. *Fase:* Uma lista de n elementos precisa ser ordenado. Nesta primeira fase, cada processador p utiliza o algoritmo Quicksort seqüencial tradicional para ordenar um conjunto contínuo de não mais que $\lceil \frac{n}{p} \rceil$ elementos. Os n elementos agora formam p listas independentes de tamanho não maior que $\lceil \frac{n}{p} \rceil$. Cada processador seleciona um conjunto de dados, chamados de amostras regulares (*regular samples*), de sua lista local já ordenada. Os índices dos elementos são: $1, (\frac{n}{p^2}) + 1, 2(\frac{n}{p^2}) + 1, ..., (p-1)(\frac{n}{p^2}) + 1$.

2. *Fase:* Nesta fase, um processador reúne e ordena todas as amostras regulares. Este então seleciona $p-1$ valores *pivot* da lista de amostras. Os índices dos *pivots* são: $p + \lfloor \frac{p}{2} \rfloor, 2p + \lfloor \frac{p}{2} \rfloor, ..., (p-1) + \lfloor \frac{p}{2} \rfloor$. Neste ponto, cada processador particiona sua lista ordenada em p pedaços disjuntos, utilizando os *pivots* como valores separadores entre os pedaços.

3. *Fase:* Na terceira fase do algoritmo, cada processador i mantém sua i-ésima partição e envia a j-ésima partição para o processador j. Em outras palavras, cada processador mantém uma partição e envia $p-1$ partições para os outros processadores.

4. *Fase:* Na última fase do algoritmo, cada processador faz um merge de sua p partição com as outras listas recebidas dos outros processadores. Fica claro que os valores na lista de um processador são disjuntos dos valores presentes nas listas dos outros processadores. A concatenação das listas individuais de cada processador forma uma lista ordenada.

Considere o seguinte exemplo onde existem 3 processadores:

- Inicialmente tem-se a lista desordenada na memória global:
⇒*Lista:* 15,46,48,93,39,6,72,91,14|36,69,40,89,61,97,12,21,54|53,97,84,58,32,27,33,72,20

- Cada processador ordena a sua parte utilizando o Quicksort Sequencial:
⇒*Lista:* 6,14,15,39,46,48,72,91,93|12,21,36,40,54,61,69,89,97|20,27,32,33,53,58,72,84,97

- Cada processador seleciona amostras regulares de sua sublista:
⇒*Amostras:* 6, 39, 72 | 12, 40, 69 | 20, 33, 72

- Um processador ordena a lista e gera uma lista de *pivots*:
⇒*Amostras:* 6, 12, 20, 33, 39, 40, 69, 72, 72
⇒*Pivots:* 33, 69

- Cada processador utiliza os *pivots* calculados para dividir cada sublista ordenada em três partes.
⇒*Lista:* 6,14,15;39,46,48;72,91,93|12,21;36,40,54,61,69;89,97|20,27,32,33;53,58;72,84,97

- O processador i mantém a sua i-ésima sublista de sua sublista e envia as outras listas j para o processador j que fica responsável por fazer um merge das sublistas que recebe com a sua sublista:
⇒*Lista:* 6,12,14,15,20,21,27,32,33|36,39,40,46,48,53,54,58,61,69|72,72,84,89,91,93,97,97

Em Quinn [1], a complexidade deste algoritmo é provada como sendo $O(\frac{n}{p}\log n + p^2\log p + \frac{n}{p}\log p)$. Se $n \leq p^3$, o primeiro termo domina, e a complexidade do algoritmo fica como sendo $O(\frac{n}{p}\log n)$.

Este algoritmo foi implementado como parte deste trabalho. A descrição sucinta da implementação encontra-se na Seção 5, e o código-fonte disponível no Anexo A e na página do trabalho (ver Seção 7).

4.3 Quicksort Paralelo Tradicional

Este algoritmo de ordenação paralela baseada no Quicksort considera a existência de uma memória compartilhada entre os processadores. Neste caso, um número idêntico de processos, um por processador, executa o mesmo algoritmo paralelo.

A algoritmo funciona no seguinte ambiente: os elementos a serem ordenados encontram-se em uma lista na memória global. Uma outra variável do tipo pilha também nesta memória global armazena os índices dos subconjuntos que ainda precisam ser ordenados.

Quando um processo está disponível, este tenta retirar da memória global uma dupla de índices de uma sublista não ordenada. Se este processo em questão conseguir uma dupla de indices, este particiona a sublista formado pelos índices da mesma forma como ocorre o particionamento do Quicksort sequencial, ou seja, baseado em um *pivot*, gera duas sublistas: uma onde os elementos são todos menores ou iguais ao *pivot* e a outra onde os elementos são maiores que o *pivot*.

Após o passo de particionamento, o processo adiciona os índices de uma sublista na variável de pilha global de sublistas ainda não ordenadas e repete o mesmo procedimento de particionamento na outra sublista.

Considere o seguinte exemplo onde existe apenas um processador:

- Inicialmente tem-se a lista desordenada na memória global e a pilha de índices contém apenas o primeiro e o último índice:

⇒*Lista:* 15,46,48,93,39,6,72,91,14,36,69,40,89, 61, 97,12,21,54,53,97,84,58,32,27,33,72,20
⇒*Pilha:* (0,26)

- Então o processador 1, inicialmente ocioso, retira a primeira dupla de índices da pilha e particiona a lista com base em um *pivot*, e.g. 61, fica com os índices da primeira sublista e coloca os indices da segunda sublista na pilha global:

⇒*Lista:* 15,46,48,39,6,14,36,69,40, 12, 21,54,53,58,32,27,33,20,61, 93,72,91,89,97,97,84,72
⇒*Pilha:* (19,26)

- Novamente o processo 1 particiona, utilizando o *pivot* 12, a lista que ele manteve e adiciona os índices da segunda sublista na pilha global:

⇒*Lista:* 6,12, 15,46,48,39,14,36,69,40 ,21, 54,53,58,32,27,33,20,61, 93,72,91,89,97,97,84,72
⇒*Pilha:* (19,26), (2,18)

- Novamente o processo 1 pariciona, utilizando o pivot 6, a lista que ele manteve. Ao ver que o número de elementos das duas sublistas é igual a um, o processo termina sua execução e fica ocioso. A lista fica igual à última, já que 6 é menor que

12.

- Então o processador 1, novamente ocioso, retira a primeira dupla de índices da pilha e particiona a lista com base em um *pivot*, e.g. 21, fica com os índices da primeira sublista e coloca os indices da segunda sublista na pilha global:
⇒*Lista:* 6,12, 15,14,20,21, 46,48,39,36,69,40,54,53,58,32,27,33,61, 93,72,91,89,97,97,84,72
⇒*Pilha:* (19,26), (6,18)

- Novamente o processo 1 particiona, utilizando o *pivot* 14, a lista que ele manteve e adiciona os índices da segunda sublista na pilha global:
⇒*Lista:* 6,12, 14, 15,20,21, 46,48,39,36,69,40,54,53,58,32,27,33,61, 93,72,91,89,97,97,84,72
⇒*Pilha:* (19,26), (6,18), (3,5)

- Ao ver que a lista que ele manteve apresenta apenas um elemento, o processador fica ocioso.

- Então o processador 1, novamente ocioso, retira a primeira dupla de índices da pilha e particiona a lista com base em um *pivot*, e.g. 20, fica com os índices da primeira sublista. Como o número de elementos da segunda sublista é um, o processador não inclui os índices na pilha:
⇒*Lista:* 6,12, 14, 15,20, 21, 46,48,39,36,69,40,54,53,58,32,27,33,61, 93,72,91,89,97,97,84,72
⇒*Pilha:* (19,26), (6,18)

- Novamente o processo 1 pariciona, utilizando o *pivot* 15, a lista que ele manteve. Ao ver que o número de elementos das duas sublistas é igual a um, o processo termina sua execução e fica ocioso. A lista fica igual à última, já que 15 é menor que 20.

A execução deste exemplo poderia ser continuada até o final, mas seria muito texto para tratar de um assunto que já deve ter sido entendido até o momento. É fato que a execução do exemplo continua do jeito que foi mostrado até o momento até que nenhuma dupla de índices seja encontrada na pilha, ou seja, até que esta fique vazia.

Por motivos de simplificação e de entendimento, apenas um processador foi utilizado, mas este algoritmo é naturalmente paralelo, permitindo que outros processadores ociosos peguem índices da pilha e comecem a trabalhar em conjunto na ordenação da lista global.

Em Quinn [1], a complexidade deste algoritmo é provada utilizando duas fases: a primeira quando tem-se mais processadores do que sublistas a serem ordenadas e a segunda quando a situação se inverte. Na primeira fase, a complexidade é: $T_1(n,p) = 2(n+1)(1-\frac{1}{p}-\log p$. Na segunda fase, a complexidade é: $T_2(n,p) = \frac{((n+1)\log(n+1)-2n)-((n+1)\log p-2(p-1))}{p}$. A complexidade todal é $T_1 + T_2$.

4.4 Comparação entre o PSRS e o QSP

Os dois algoritmos paralelos de ordenação estudados nesta Seção, o *Parallel Sorting by Regular Sampling*[2] e a versão tradicional de Quicksort Paralelo[3], apesar de ambos utilizarem máquinas MIMD, seus funcionamentos e características diferem muito.

No PSRS, há troca de mensagens entre os processadores, de modo que realizar a sua implementação utilizando, por exemplo, máquinas diferentes pertencentes a uma rede de

[2] Abrev.: PSRS
[3] Abrev.: QSP

estações de trabalho torna-se bastante possível. Já no caso do QSP, nenhuma mensagem é considerada entre os processadores. Estes últimos são responsáveis por verificar a pilha de índices para ver se existem sublistas para serem ordenadas sempre que estes processadores estiverem ociosos.

Quanto à memória compartilhada, tem-se que ambos a consideram. Porém PSRS pode facilmente ser convertido para um sistema de memória distribuída, já que praticamente todas as fases seguem através de troca de mensagens. Já no caso do QSP, tem-se que este necessita mesmo de uma memória compartilhada, visto que apenas índices são considerados e as modificações precisam ser realizadas na lista de números global.

O QSP tem como alvo principal os computadores multiprocessados do tipo UMA (*Uniform Memory Access*[4]), pois este algoritmo assume que todo processador tem acesso rápido a todos os elementos ainda não ordenados. Assume também o acesso rápido por todos os processadores a uma pilha contendo os índices das sublistas.

A analise da ordenação paralela do tipo divisão e conquista como utilizado pelo QSP pode obter um *speedup* não satisfatório se todos os processadores não começarem a trabalhar rapidamente o bastante. Já o algoritmo PSRS foi aplicada a uma variedade de máquinas MIMD (inclusive, neste trabalho, a redes de estações de trabalho) e têm conseguido um bom *speedup* em todos [1].

4.5 Quicksort paralelo com paralelismo de Dados e de Controle

Antes de partirmos para a discussão de como o sc Quicksort pode ser implementado utilizando os modelos de paralelismo de dados e de controle, torna-se importante uma definição melhorada destes modelos:

- *Paralelismo de Dados - Data Parallelism:* É a utilização de unidades múltiplas funcionais para aplicar a mesma operação simultaneamente aos elementos de um conjunto de dados. Um aumento de k no número de unidades funcionais leva a um aumento de k no aumento do *throughput* do sistema, se não tiver nenhum overhead associado com o aumento de paralelismo.

- *Paralelismo de Controle - Control Parallelism:* Em contraste com o paralelismo de dados, onde o paralelismo é conseguido aplicando uma mesma operação sobre um conjunto de dados, o paralelismo de controle é conseguido aplicando-se diferentes operações sobre diferentes elementos de dados simultaneamente.

Praticamente todos os algoritmos estudados encontram-se no modelo de paralelismo de dados, visto que os mesmos algoritmos são executados simultaneamente por processadores diferentes sobre dados diferentes.

Parece meio difícil, pelo menos não muito trivial, imaginar um algoritmo paralelo que utilize o modelo de paralelismo de controle par a ordenação de conjuntos. O que acontece é que na maioria dos casos estamos trabalhando com algoritmos naturalmente recursivos e esta recursividade, ao ser transformado em algoritmo paralelo, geralmente envolve a execução de mesmos passos para processadores diferentes.

Uma possibilidade de algoritmo simples que se pode imaginar que utilize paralelismo de controle é o seguinte: O primeiro processador recebe uma lista para ser ordenada, faz um particionamento desta lista e passa as duas sublistas para dois processadores diferentes

[4]UMA: Assume que todos os processadores trabalham através de um mecanismos de troca centralizado para alcançar uma memória compartilhada centralizada.

e fica esperando outra lista para ser ordenada. Desta forma, tem-se que a saída de um processador serve como entrada para outros dois processadores, que irão servir como entrada para outros quatro, e assim por diante.

O algoritmo QSP explicado na seção anterior pode ser visualizado, de certa forma, com um algoritmo que utiliza o modelo de paralelismo de controle. Isso se considerarmos que o fato de inserir uma dupla de índices na pilha global, que é a saída de um processados, pode servir como entrada de outro processador que irá pegar os índices e ordenar a sublista.

Por outro lado, o QSP também pode ser considerado um algoritmo que utiliza o modelo de paralelismo de dados, visto que todos os processadores executam o mesmo algoritmo sobre dados diferentes.

5 Implementação dos Algoritmos Paralelos

A implementação dos algoritmos explicados até o momento (*Parallel Sorting by Regular Sampling* e *Quicksort Paralelo Tradicional*) foi realizada na *linguagem C*, utilizando o compilador *GCC*. Todo o código-fonte encontra-se listado no Anexo A e disponível também em meio digital na página deste trabalho (ver Seção 7).

Para facilitar a implementação dos Algoritmos Paralelos, tanto as citadas no trabalho como também diversos outros, uma *Biblioteca Genérica de Programação Paralela* foi desenvolvida em conjunto com uma metodologia que pareceu ser bem viável e fácil de ser utilizada. Esta biblioteca é composta basicamente de funções de comunicação entre processos localizados em computadores diferentes utilizando *sockets*. Além disso, exemplos simples de desenvolvimento tanto do *master* quando do *slave* são mostradas. A implementação desta biblioteca é explicada na Seção 5.2.

Para facilitar tanto a implementação como também o entendimento do código, este foi separado em diversos arquivos, cada qual com seus conteúdos relacionados:

- *Module.h:* Contém as definições dos Tipos de Dados e as declarações de todas as Funções da Implementação.

- *Socket.h:* Contém as definições dos Tipos de Dados e as declarações de todas as Funções da *Biblioteca de Programação Paralela.*

- *Main.c:* Código inicial da Implementação. Executa as implementações bem como o *master* ou o *slave* das mesmas, dependendo dos parâmetros passados pela linha de comando (explicadas abaixo).

- *Utils.c:* Funções auxiliares que são utilizadas pelos diversos métodos de solução do problema, como por exemplo funções para imprimir listas de números, para subtrair tempos, uma função que imprime a ajuda do programa, funções de *threads* e para gerar números aleatórios.

- *PSRSMasterMain.c:* Executa o código principal do *master* do Algoritmo, mostrado por Quinn [1], *Parallel Sorting by Regular Sampling.* Como será explicado na Seção 5.3, o *master* é responsável por acordar os outros processadores, coordená-los em sua execução do algoritmo e, por último, reunir as listas ordenadas.

- *PSRSSlaveMain.c:* Executa a parte dos *slaves* (escravos) do Algoritmo *Parallel Sorting by Regular Sampling.* Como será mostrado na Seção 4.3, os *slaves* basicamente ficam esperando pela conexão do *master* e em seguida executa os comandos por ele enviados.

- *QSPMain:* Executa tanto a parte *master* quanto a *slave* do Algoritmo do *Quicksort Paralelo Tradicional.* Esta implementação é explicada na Seção 4.3.

- *Arquivo.c:* Contém funções que manipulam arquivos do sistema. Tais funções incluem abrir um arquivo, ler uma lista de números a partir de um arquivo binário, salvar uma lista de números que está na memória principal em um arquivo binário e uma função *thread* executada pelo *master* para solicitar aos *slaves* que leiam os seus arquivos de entrada.

- *ListaNumero.c:* Contém funções que manipulam uma lista de números do tipo `unsigned long int`. A lista foi implementada utilizando vetor alocado dinamicamente. Neste arquivo encontram-se também funções *threads* e funções que manipulam *sockets* que são executadas tanto pelo *master* quanto pelo *slave* que são relacionadas a Lista de Números.

- *ListaPivots.c:* Neste arquivo estão localizadas as funções que lidam com a Lista de *Pivots*, necessária na terceira fase do algoritmo PSRS. Novamente incluem funções *threads* e que manipulam *socket* tanto do *master* quanto do *slave* que são relacionadas aos *Pivots*.

- *ListaSamples.c:* Neste arquivo estão localizadas as funções que lidam com a Lista de *Samples*, necessária na segunda fase do algoritmo PSRS. Novamente incluem funções *Threads* e que manipulam *socket* tanto do *master* quanto do *slave* que são relacionadas aos *Samples*.

- *Ordenacao.c:* Todas as funções relacionadas de alguma forma com a ordenação estão localizadas neste arquivo. Dentre estas funções pode-se citar a função do Quicksort seqüencial, utilizada tanto pelo PSRS quanto pelo QSP, a função de partição utilizada pelo QSP, uma função que faz um merge de duas listas (baseada no *Mergesort*) e uma função que verifica se uma lista está ordenada. Esta última função foi utilizada, em conjunto com o número de elementos do resultado dos algoritmos, para verificar se as listas obtidas realmente estavam ordenadas.

- *Slaves.c:* Neste arquivo estão implementadas as funções que lidam com a Lista de *slaves*. Como será explicado, uma lista de *slaves* é mantida em cada processo, seja *master* ou *Slave*, e esta lista é lida inicialmente pelo *master* de um arquivo e depois enviada para os *slaves* via mensagens. Algumas funções incluem ler a lista de *slaves* de um arquivo, uma função que conecta em todos os *slaves* e outra que desconecta, e funções de *thread* e *socket* executadas pelo *master* e pelos *Slaves* relacionadas a esta lista.

- *Sockets.c:* A *Biblioteca de Programação Paralela* desenvolvida contém diversas funções que estão localizadas neste arquivo. Este será explicado mais detalhadamente na Seção 5.2 e contém funções relacionadas a *sockets*, como abrir uma porta, conectar, desconectar, receber e enviar mensagens, abrir *threads* a cada conexão e etc.

- *Timer.c:* Pequena biblioteca responsável por calcular e exibir o tempo de execução de partes da implementação e, no final, o tempo total.

- *Timer.h:* Definição das variáveis e funções da biblioteca de temporização de execução.

- *makefile:* Diretrizes para o comando make para compilar, limpar e executar a implementação. Compatível com o make do Unix.

Conforme mencionado, um único comando é responsável por iniciar tanto o *master* quanto o *slave* dos dois algoritmos implementados (PSRS e QSP). A sintaxe deste comando é a seguinte:

```
Utilização:
      Parallel Sorting by Regular Sampling:
          - Master: # ./PaaTP4 -s <NomeArquivoClientes>
          - Slate:  # ./PaaTP4 -c <NumPortaTCP>
      Parallel QuickSort:
          - Master: # ./PaaTP4 -t -1
          - Slave:  # ./PaaTP4 -t <NumPortaTCP>
      Quicksort Sequencial:\n");
          -         # ./PaaTP4 -q <NomeArqNumeros>\n");
      Gerar Massa de Dados:\n");
          -         # ./PaaTP4 -g <QuantidadeNumeros> <SaidaArquivo>\n");
```

5.1 Estruturas de Dados Utilizadas

Diversas estruturas de dados foram utilizadas na implementação dos dois ALGORITMOS PARALELOS. Os Códigos 2 e 3 mostram tais estruturas que serão explicadas nos parágrafos a seguir. Além disso, é mostrado também as bibliotecas utilizadas, as constantes e também a definição de todas as funções da implementação espalhadas pelos diversos arquivos (ver Seção 5).

As principais constantes definidas foram:

- `MAX_IP`: Definido na linha 34, esta constante representa o tamanho máximo de um endereço IP (e.g., 192.168.8.1). Definido por padrão em 15.
- `MAX_HOSTS`: Definido na linha 35, indica o numero máximo de *slaves*. Utilizado para definir o tamanho da lista de *hosts slaves*. Definido por padrão em 20, um número muito superior ao necessário pelos testes executados.
- `MAX_RAND`: Número máximo devolvido pelo gerador de números aleatórios. Definido como sendo o maior número long menos 1. Linha 36.
- `NUM_TAM`: Tamanho máximo do vetor de números. Definido como sendo um número bastante superior ao arquivo de 200MB utilizado nos experimentos. Linha 37.
- `SAM_TAM`: Tamanho máximo do vetor de *samples*. Como cada *host* gera um número de *samples* igual ao número de *hosts*, o número máximo de *samples* que o *master* vai receber é o número de processadores ao quadrado. Linha 38.
- `PIV_TAM`: Tamanho máximo do vetor de *pivots*. Um número igual ao número máximo de *hosts* já seria o suficiente, mas para evitar problemas de memória, este número foi aumentado. Definido na linha 39.

Duas estruturas foram definidas:

- `TLista`: Estrutura que representa uma lista de números do tipo `unsigned long int`. Está definida nas linhas 43 a 46 e apresenta os seguintes componentes:

  `tamLista`: Indica qual o tamanho da lista atualmente.

  `numeros`: Ponteiro para um vetor de números do tipo `unsigned long int`. Este vetor é alocado dinamicamente pela função `novaListaNum` que aloca um vetor com tamanho definido pela constante `NUM_TAM`.

- `HostSlave` Estrutura que representa as informações necessárias de um único *host slave*. Mais adiante um vetor desta estrutura representando a lista de *slaves* será definida. Além disso, todos os *threads* criados pelos algoritmos tem como argumento uma estrutura deste tipo. Está definida nas linhas 49 a 58 e apresenta os seguintes componentes:

  `id`: Identificação do *host slave*. Serve para fins de *debug*, para que os *threads* possam se identificar nas mensagens geradas e é necessário também pelo algoritmo PSRS na hora de trocas as listas com os outros *slaves*.

  `IP`: IP do *host slave*. Utilizado para efetuar a conexão a este *host*.

  `porta`: Porta de conexão do *host slave*. Utilizado para efetuar a conexão a este *host*.

  `socketFD`: O resultado da conexão a este *host* é salva neste descritor de arquivo.

`thread`: *Thread* do *host slave*. Para cada *thread* gerado, o apontador de retorno é colocado nesta variável para que futuramente possa ser feito um *join*.

`arqEntrada`: Arquivo de entrada do *host slave*. É lido do arquivo de *hosts slaves* pelo *master* e salvo nesta variável.

`numLista`: Lista de números para o *slave*. Uma vez que não foram implementadas medidas de sincronização e bloqueios, os *threads* não podem escrever diretamente na lista de números globais. Desta forma, eles escrevem em listas separadas e depois o processo que executou os *threads* é responsável de unificar as listas.

`samLista`: Lista de *samples* do *host slave*. Necessária pelo mesmo motivo da lista anterior.

Por último, as seguintes variáveis globais são definidas:

- `hostsSlaves:` Vetor de estruturas do tipo HostSlave (definida anteriormente). O *master* lê uma lista de *slaves* de uma lista, preenche este vetor e depois a envia pela rede para todos os *slaves* que também preenchem este vetor. Desta forma, todos os processos ficam com a mesma lista de *slaves*. Definida na linha 61.
- `numHostsSlaves:` Número de *hosts slaves*.
- `numLista:` Lista de números do processo atual. Utiliza a estrutura `TLista` definida anteriormente. Definida na linha 64.
- `samLista:` Lista de *samples* do processo atual. Utiliza a estrutura `TLista` definida anteriormente. Definida na linha 65.
- `pivLista:` Lista de *pivots* do processo atual. Utiliza a estrutura `TLista` definida anteriormente. Definida na linha 66.
- `slaveID:` Variável que define o ID do *slave* atual. Utilizada apenas pelos processos *slaves*. Definida nalinha 68.

5.2 Biblioteca de Programação Paralela

Conforme mencionado, para facilitar a implementação dos Algoritmos Paralelos, tanto as citadas no trabalho como também diversos outros, uma *Biblioteca Genérica de Programação Paralela* foi desenvolvida em conjunto com uma metodologia que pareceu ser bem viável e fácil de ser utilizada. Esta biblioteca é composta basicamente de funções de comunicação entre processos localizados em computadores diferentes utilizando *sockets*. Além disso, exemplos simples de desenvolvimento tanto do *master* quando do *slave* são mostradas.

As principais características desta biblioteca são:

- *Utilização de Sockets:* Utiliza *sockets* para a comunicação entre os processos. Toda a parte de conexão, envio de dados brutos, verificação de erros é encapsulada o máximo possível. Ficando para as implementações apenas o trabalho de executar as funções com os parâmetros corretos.
- *Múltiplas Requisições:* Através da utilização de *threads* é possível o atendimento a múltiplas requisições ao mesmo tempo, permitindo, por exemplo, que um *slave* mantenha sua conexão com o *master*, conecte-se em outros *slaves* e ainda continue recebendo novas conexões de outros *slaves*, isso tudo ao mesmo tempo e com um número praticamente ilimitado (limitado pela capacidade da máquina).

Código 2 Tipos de Dados e Constantes utilizadas pelas Implementações.

```c
/***| Bibliotecas |***/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>

#include <pthread.h>
#include "Sockets.h"

/***| Constantes |***/

#ifndef _MODULE_H_
#define _MODULE_H_ "@(#)Module.h $Revisao 0.1"

#define _IS_UNIX_
#define _IS_CLIENT_
#define _TIMEVAL_

#define DEBUG                        4

#define FALSE                        0
#define TRUE   (signed int)     ~FALSE
#define OK                       FALSE
#define ERROR                     TRUE

#define PROGRAM_NAME "QuickSort Paralelo"
#define PROGRAM_VERSION          "0.1"

#define MAX_IP                      15
#define MAX_HOSTS                   20
#define MAX_RAND          LONG_MAX-1UL
#define NUM_TAM               53000000
#define SAM_TAM    MAX_HOSTS*MAX_HOSTS+1
#define PIV_TAM    MAX_HOSTS*MAX_HOSTS+1

/***| Declaracao das Estruturas |***/
// Lista de Numeros
typedef struct {
  int tamLista;                      // Tamanho da Lista
  unsigned long int *numeros;        // Lista de Numeros
} TLista;                            // Lista de Numeros

// Host Slave
typedef struct {
  int  id;                           // Identificacao do Host Slave
  char IP[MAX_IP];                   // IP do Host Slave
  int  porta;                        // Porta de Conexao do Host Slave
  int  socketFD;                     // Socket do Host Slave
  pthread_t thread;                  // Thread do Host Slave
  char arqEntrada[100];              // Arquivo de entrada do Host Slave
  TLista numLista;                   // Lista de Numeros para o Slave
  TLista samLista;                   // Lista de Samples do Host Slave
} HostSlave;                         // Host Slave

/***| Variaveis Globais |***/
HostSlave hostsSlaves[MAX_HOSTS]; // Lista de Hosts Slaves
int       numHostsSlaves;            // Numero de Hosts Slaves

TLista numLista;                     // Lista de Numeros do Processo Atual
TLista samLista;                     // Lista de Samples do Processo Atual
TLista pivLista;                     // Lista de Pivots do Processo Atual

int slaveID;                         // Identificacao do Slave Atual
```

Código 3 Definição de todas as funções implementadas.

```
/***| Declaracao das Funcoes |***/
// --> PSRSSlaveMain.c
int  slaveMain(int portaTCP);
void *funcaoFilho(void *voidargs);
int  respSolicitacao(char *solicitacao, char *resposta, int socket);
// --> PSRSMasterMain.c
int  masterMain(char *arqClientes);
void geraExemplo();
// --> QSPMain.c
int  qspMain(int portaTCP);
void executarQSP();
void *qspThread(void *voidargs);
void *funcaoFilhoQsp(void *voidargs);
int  respSolicitacaoQsp(char *solicitacao, char *resposta, int socket);
void retornaLista();
// --> Slaves.c
int  conectaSlaves();
int  desconectaSlaves();
void *cleanUpThread(void *voidargs);
int  lerSlavesArq(char *nomeArqEntrada);
void *enviarSlavesThread(void *voidargs);
void *enviarIDsThread(void *voidargs);
int  lerSlavesConexao(int socket);
// --> Arquivo.c
FILE *abreArquivo(char *nomeArquivo, char *modo);
int  lerListaArquivo(TLista *lista, char *arqEntrada);
int  salvaListaArquivo(TLista *lista, char *arqSaida);
void *enviarListaArqThread(void *voidargs);
// --> ListaNumeros.c
void novaListaNum(TLista *lista);
void addNumero(unsigned long int numero, TLista *lista);
void addNumeroOrdenado(unsigned long int numero, TLista *lista);
void addNumerosAleatorios(int quantidade, TLista *lista);
void printLista(TLista *lista);
void *enviarListaThread(void *voidargs);
void *execOrdenaListaThread(void *voidargs);
void *execTrocaListaThread(void *voidargs);
void *execMergeListasThread(void *voidargs);
void *getListasTrocadasThread(void *voidargs);
void joinListas();
int  lerListaConexao(int socket);
void *trocaListaThread(void *voidargs);
int  receberTrocaLista(int socket, int procID);
int  enviarListaTrocada(int socket);
void gerarListaTrocada();
// -> ListaSamples.c
void novaListaSam(TLista *lista);
void *getSamplesThread(void *voidargs);
void joinSamples();
void gerarSamples();
int  enviarSamples(int socket);
// --> ListaPivots.c
void novaListaPiv(TLista *lista);
void gerarPivots();
void *enviarPivotsThread(void *voidargs);
int  receberPivots(int socket);
// --> Ordenacao.c
void quickSort(TLista *lista);
void ordena(int esq, int dir, unsigned long int *A);
void particao(int esq, int dir, int *i, int *j, unsigned long int *A);
void verificaOrdenacao(TLista *lista);
void mergeListas(TLista *lista1, TLista *lista2);
// --> Utils.c
int criaThreads(void *funcThread);
int joinThreads();
void printAjuda();
void chomp(char *string);
void iniciaMyRand();
int myRand(int Max);
int isNumero(char *string);
int  stricmp(char *str1, char *str2);
#endif  /* _MODULES_H_ */
```

- *Compartilhamento de Variáveis Globais:* Uma vez que a tecnologia utilizada na concorrência é de *threads* e não de *fork*, todas as variáveis globais são acessíveis por todos os *threads* de conexão criados. Desta forma, uma conexão criada por um *slave* em outro *slave* pode acessar informações que o *master* enviou para este último.

- *Independente de Aplicação:* Na medida do possível, foi feita uma biblioteca independente de aplicação. Desta forma, esta biblioteca pode ser retirada desta implementação e facilmente utilizada em outros trabalhos e implementações sen nenhuma modificação na mesma. O único requisito da biblioteca é, como será mostrado adiante, que uma função chamada `void *funcaoFilho(void *voidargs)` exista na aplicação que está sendo implementada. Esta função será executada cada vez que uma nova requisição de conexão é feita ao processo.

5.2.1 Estruturas de Dados

Basicamente a única estrutura utilizada na biblioteca é uma lista de *threads* e seus argumentos. Uma constante chamada `MAXMSG` foi definida como sendo o tamanho máximo de uma linha de dados enviada em um *socket*.

O Código 4 lista estes tipos de dados implementados na linguagem C bem como a definição das funções da biblioteca e as outras bibliotecas necessárias para ela compilar.

5.2.2 Funções de Comunicação

Uma biblioteca de programação é composta de diversas funções que procura auxiliar o desenvolvimento de novos sistemas que utilizam um determinado recurso. Trabalhar com *sockets* diretamente não é uma tarefa tão simples, mas pode ser bastante facilitada com o auxílio de funções criadas que procurem abstrair toda a complexidade deste recurso. É exatamente este o objetivo da Biblioteca criada. Com a utilização das funções disponibilizadas, o conhecimento necessário para a utilização de *sockets* em novos aplicativos fica limitado apenas às definições de nome de *host* e de porta de conexão.

A seguir serão mostradas cada uma das funções implementadas na biblioteca, bem como uma breve descrição da mesma, seus parâmetros de entrada e também o seu retorno.

Função abrePorta

int `abrePorta(`*unsigned short numPorta*`)`

Descrição	Função utilizada para abrir uma porta do tipo TCP. Não é necessário executar esta função diretamente, visto que ela é executada pela função `gerenciaConexoes`.
Parâmetros	`numPorta` Numero da porta a ser aberta.
Retorno	Descritor de Arquivo do Socket Aberto.

Função esperaConexao

int `esperaConexao(`*int socket*`)`

Código 4 Tipos de Dados e Funções da Biblioteca de Programação Paralela.

```
#ifndef _SOCKETS_H_
#define _SOCKETS_H_ "@(#)Sockets.h $Revisao 0.1"

#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "Module.h"

// Constantes
#define MAXMSG 1024

// Argumentos do Thread do Socket
struct strSchArgs {
  int conexao;                    // Ponteiro para a Conexao
  int socket;                     // Ponteiro para o Socket Principal
};

// Definicao das Funcoes
int  abrePorta(unsigned short numPorta);
int  esperaConexao(int Socket);
int  socketPuts(int Socket, char *String);
int  socketGets(int Socket, char *String);
int  socketGetLinha(int Socket, char *String);
void gerenciaConexoes(unsigned short numPorta);
int  socketClient(char *NomeHost, unsigned short numPorta);
int  socketPutsArq(int Socket, char *ArquivoBin);
int  socketPutBin( char* buf, unsigned int sz, int clientSock );
extern void *funcaoFilho(void *voidargs);
int  conecta(char *nomeHost, unsigned short numPorta);
int  desconecta(int socket);
void SendVector(int sd, int size, unsigned long int* vector);
void ReceiveVector(int sd, int* numelem, unsigned long int* vector);

#endif /* _SOCKETS_H_ */
```

Descrição Espera até que uma conexão seja requisitada em um determina do *socket*. Não é preciso executar esta função diretamente visto que ela é executada pela função `gerenciaConexoes`.

Parâmetros `socket` Descritor de Arquivo do *socket*.

Retorno Descritor de Arquivo para a Nova Conexão.

Função socketPuts

int `socketPuts(`*int socket, char *string*`)`

Descrição Função utilizada principalmente pelas aplicações. Responsável por enviar dados no *socket* para a outra ponta da comunicação.

Parâmetros `socket` Descritor de Arquivo do *socket*.
`string` Conteúdo do Dado a ser enviado.

Retorno Quantidade de bytes enviado.

Função socketPutsArq

int `socketPutsArq(`*int socket, char *arquivoBin*`)`

Descrição Função utilizada para escrever dados binários, a partir de um arquivo, em um *socket*.

Parâmetros `socket` Descritor de arquivo do *socket*.
`arquivoBin` Arquivo a ser enviado em modo binário.

Retorno Quantidade de bytes enviado.

Status Esta função está apresentando problemas e ainda está em testes.

Função socketPutsArq

int `socketPutsArq(`*int socket, char *arquivoBin*`)`

Descrição Função utilizada para escrever dados binários, a partir de um arquivo, em um *socket*.

Parâmetros `socket` Descritor de arquivo do *socket*.
`arquivoBin` Arquivo a ser enviado em modo binário.

Retorno Quantidade de bytes enviado.

Status Esta função está apresentando problemas e ainda está em testes.

Função socketGets

int `socketGets(`*int socket, char *string*`)`

Descrição	Função utilizada principalmente pelas aplicações. Responsável por receber dados do *socket* que foi enviado pela outra ponta da comunicação.
Parâmetros	`socket` Descritor de Arquivo do *socket*.
	`string` Onde será armazenado os dados recebidos.
Retorno	Quantidade de bytes recebidos.

Função socketGetLinha

int `socketGetLinha(`*int socket, char *string*`)`

Descrição	Recebe várias linhas no estilo do FTP. Pega o código da primeira linha e fica recebendo ate que o mesmo código seja recebido (indicando a ultima linha).
Parâmetros	`socket` Descritor de Arquivo do *socket*.
	`string` Onde será armazenado os dados recebidos.
Retorno	Sucesso ou não.

Função gerenciaConexoes

void `gerenciaConexoes(`*unsigned short numPorta*`)`

Descrição	Função principal da biblioteca. Cria o *socket*, abre a porta, fica escutando e gera um processo filho (*thread*) a cada conexão recebida. No processo filho (thread), a função funcaoFilho() é executada.
Parâmetros	`numPorta` Número da porta a ser aberta.
Retorno	Sem retorno.

Função socketClient

int `socketClient(`*char *nomeHost, unsigned short numPorta*`)`

Descrição	Cliente do socket. Conecta em um *host* e em uma porta especificada nos parâmetros.
Parâmetros	`nomeHost` Nome do *host* que será conectado. `numPorta` Número da porta que está aberta na outra ponta.
Retorno	Descritor de Arquivo do *socket* criado.

Função socketPutBin

int `socketPutBin(`*char* buf, unsigned int sz, int clientSock*`)`

Descrição	Envia um buffer binário para a outra ponta.
Parâmetros	`buf` Buffer binário a ser enviado. `sz` Tamanho em Bytes a ser enviado. `clientSock` Ponteiro do *socket*.
Retorno	Quantidade de bytes enviados.
Status	Esta função está apresentando problemas e ainda está em testes.

Função conecta

int `conecta(`*char *nomeHost, unsigned short numPorta*`)`

Descrição	Conecta em uma máquina utilizando o nome do *host* e a porta passados por parâmetro. Não precisa ser executada diretamente. Esta função é utilizada pela função socket-Client.
Parâmetros	`nomeHost` Nome do *host* que será conectado. `numPorta` Número da porta que está aberta na outra ponta.
Retorno	Descritor de Arquivo do *socket* criado.

Função desconecta

int `desconecta(`*int socket*`)`

Descrição	Desconecta de uma máquina.
Parâmetros	`socket` *socket* que será desconectado.
Retorno	Sucesso ou não.

Função SendVector

void `SendVector(`*int sd, int size, unsigned long int* vector*`)`

Descrição	Envia um vetor do tipo `unsigned long int`.
Parâmetros	`sd` *Socket* que será enviado os dados.
	`size` Numero de elementos do vetor.
	`vector` O vetor a ser enviado.
Retorno	Sucesso ou não.

Função ReceiveVector

void `ReceiveVector(`*int sd, int* numelem, unsigned long int* vector*`)`

Descrição	Recebe um vetor do tipo `unsigned long int`.
Parâmetros	`sd` *Socket* que será recebido os dados.
	`size` Numero de elementos do vetor.
	`vector` O vetor a ser salvo.
Retorno	Sucesso ou não.

5.2.3 Exemplo Prático

Conforme mencionado, esta biblioteca é composta por funções e também por uma metodologia de uso que pode ser aplicado aos diversos programas que executam em paralelo ou mesmo que precisam de uma comunicação entre servidor e clientes.

Esta seção procura descrever esta metodologia através de um exemplo simples de implementação de um *master* e de um *slave*.

Exemplo de Slaves

Um processo *slave* basicamente abre uma porta de conexão e fica esperando que um *master* se conecte nele e o mande executar tarefas.

O código a seguir utiliza a *Biblioteca de Programação Paralela* para abrir esssa porta e uma função *thread* denominada `funcaoFilho` é definida para responder às requisições de conexões feitas na porta do *slave*.

Como pode-se perceber, pelos Códigos 5 e 6, com poucas linhas de código já é possível fazer um processo *slave* que atenda as requisições do *master* e ainda envia uma resposta de volta.

Master

O processo *master* é responsável por se conectar no(s) slave(s) e enviar trabalhos para eles realizarem.

Para se conectar em um *slave*, basta utilizar a função `socketClient` descrita anteriormente ou mesmo a função `conecta`, também descrita na seção anterior.

Código 5 Função Principal de um processo Slave.

```
#include "Sockets.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao Main do Slave do QuickSort Paralelo.         *
 *   Abre uma porta e fica respondendo  a  requisicoes *
 *   do Master.                                        *
 *                                                     *
 *    @param portaTCP Porta que sera aberta.           *
 *    @return         Execucao com sucesso ou nao.     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int slaveMain(int portaTCP) {
  if (DEBUG>3) printf("[slaveMain] Iniciando Slave ..\n");
  if (DEBUG>3) printf("[slaveMain]   Gerenciando Conexoes na porta %d ..\n", portaTCP);
  // Seta o ID do Slave atual para -1
  slaveID = -1;
  // Espera por Conexoes. Quando vier, a FuncaoFilho e executada.
  gerenciaConexoes(portaTCP,0);
  return 1;
}
```

Código 6 Função Filho de um processo Slave.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao funcaoFilho                                  *
 *   Funcao que e executada sempre que uma conexao no- *
 *   va eh recebida.                                   *
 *                                                     *
 *    @param voidargs Ponteiro que contem o socket.    *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void *funcaoFilho(void *voidargs) {
  char solicitacao[MAXMSG];       // String para as Solicitacoes
  char resposta[MAXMSG];          // String para as Respostas
  int continua = 1;               // Se continua na Funcao ou Sai
  struct strSchArgs *schArgs;     // Estrutura de Arg. do Thread
  int socket;                     // Ponteiro para o Socket

  // Seta a Estrutura enviada pelo Thread
  schArgs = (struct strSchArgs *)voidargs;
  // Seta o Socket
  socket = schArgs->conexao;

  if (DEBUG>3) printf("[%d][funcaoFilho]     Conexão recebida !!\n", slaveID);

  // Conexao Recebida. Vamos dar as Boas Vindas.
  sprintf(resposta, "001 (QuickSort Paralelo - User Default Installation)\r\n");
  socketPuts(socket, resposta);

  // Fica escutando os Comandos
  while (continua) {
    fflush(NULL);
    // Recebe a Solicitacao
    if (socketGets(socket, solicitacao) < 0) {
      close(socket);
      continua = 0;
    }
    else {
      // Responde a Solicitaca
      continua = respSolicitacao(solicitacao, resposta, socket);
      // Envia a Resposta
      socketPuts(socket, resposta);
    }
  }
  // Fecha o Socket
  close(socket);
  // Termina a execucao do Thread
  pthread_exit (0);
}
```

5.2.4 Trabalhos Futuros

Esta biblioteca já se mostrou bastante útil, viável, fácil de se utilizar e robusta, através da implementação dos *Algoritmos Paralelos* que serão explicadas nas seções seguintes.

Porém, algumas melhorias ainda precisam ser feitas nela para melhorar a performance dos aplicativos e também adicionar novas funcionalidades necessárias para os sistemas paralelos.

Uma melhoria que precisa ser feita é nas funções que lidam com arquivos binários. Apesar destas terem sido implementadas, estão apresentando problemas e não puderam ser utilizadas nas implementações. Utilizar dados binários traz um ganho muito grande na comunicação. Principalmente quando estamos tratando de redes de computadores, onde as velocidades de transmissão giram em torno de 10 a 100 MBits em uma rede local ou muito menos, de alguns Kbits até poucos MBits quando estamos lidando com redes distantes ou mesmo a Internet.

Um exemplo do que foi falado no parágrafo anterior pode ser tirado das implementações realizadas com o Quicksort. Arquivos de 200MB contendo números `unsigned long int` foram utilizados, tendo um total de 52.428.790 números. Isso pode ser mostrado com a seguinte conta: um número do tipo `unsigned long int` ocupa 4 bytes na memória e ((52.428.790 * 4)/1024)/1024 = 200MB.

Porém, se a transmissão destes dados não forem do tipo binário e sim no modo texto, tem-se que cada número (que tem cerca de 9 dígitos) enviado em modo ASCII irá ocupar 9 bytes mais 2 bytes de retorno de linha. Se for feita a conta, ((52.428.790 * 11)/1024)/1024 = 550MB. O que resulta em uma transmissão de muito mais do dobro de bytes.

Algumas funções foram implementadas para lidar com dados binários, mas apenas considerando os vetores de `unsigned long int`.

Uma outra melhoria que pode ser realizada nesta biblioteca é a inclusão de meios de se facilitar a sincronização e bloqueio através de semáforos, barreiras e etc.

5.3 Implementação do *Parallel Sorting by Regular Sampling*

O funcionamento algoritmo PSRS foi explicado na Seção 5.3. Na explicação realizada na seção citada, baseada nos estudos feitos por Quinn [1], uma arquitetura MIMD é considerada. Apesar de na implementação também estarmos utilizando um arquitetura MIMD, esta é do tipo rede de estações de trabalho (*NOW - Network of Workstations*). Como os mecanismos de comunicação são bem diferentes neste tipo, o funcionamento da implementação nesta arquitetura será melhor explicada nesta seção.

5.3.1 Funcionamento da Implementação

Toda a execução da implementação do algoritmo é coordenada pelo processo denominado *Master*. Este processo é responsável por contactar todos os *slaves* disponíveis e orientá-los no processo de execução do algoritmo. Desta forma, o *master* comanda aos *slaves* que leiam o arquivo de entrada, que o ordene, que coletem amostragens, que leiam os *pivots*, que troquem as listas entre si e, por último, que retornem os resultados parciais para o *master* concatenar e verificar a ordenação. A Figura 1 ilustra a posição do *master* diante dos *slaves*.

Esta coordenação é feita através de mensagens-comando enviada pelo *master* para os *slaves* através de do *socket*. Estas mensagens são linhas de texto que são recebidas pelo *slave* e este responde também em uma linha se a execução foi bem sucedida ou não bem como responde a requisições de listas de número e etc.

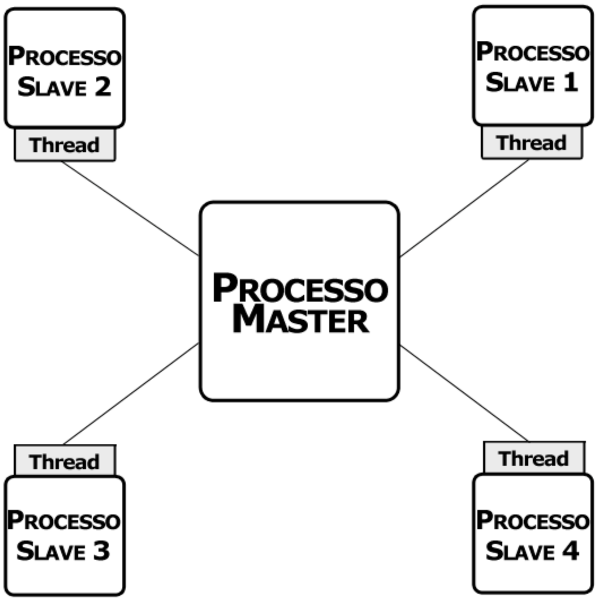

Figura 1: *Master* **coordenando a execução do algoritmo por parte dos** *slaves*.

As seguintes mensagens-comando utilizada pelo *master* para coordenar os *slaves* foram definidas:

- *YOU ARE <SlaveID>:* Esta mensagem é utilizada pelo *master* para dizer para o *slave* atual qual o *id* dele na lista de *slaves*. Conforme mencionado anteriormente, esta identificação do *slave* é necessária tanto por motivos de *debug* do sistema como também é necessário na hora de trocar as listas de números com os outros *slaves*. Se o comando for bem sucedido, o *slave* responde para o *master*: "*005 - Ok. Agora eu tenho o ID <SlaveID>*".

- *PUT SLAVES:* Para a execução do algoritmo é necessário que todos os *slaves* tenham conhecimento da lista de *slaves* presente no *master*. Para que isso seja possível, o *master* envia este comando para informar aos *slaves* que ele irá começar a enviar esta lista. Os *slaves* respondem "*022 - OK. Pronto para receber lista de Slaves*" e então a lista começa a ser enviada. Quando termina, o *master* envia o comando "*END SLAVES*" e então estes respondem com a mensagem "*022 - OK. Recebido os Slaves*".

- *PUT LIST:* Com este comando, o *master* está falando para o *slave* atual que ele irá começar a enviar a lista de números que deverá ser ordenado. O *slave* responde com a mensagem "*012 - OK. Pronto para receber Lista de Numeros*" então os números começam a ser enviados. Quando a lista termina, o *master* envia a mensagem de término "END LIST" e então se tudo correu conforme o esperado o *slave* responde que está com a lista pronta: "*014 - OK. Recebido a Lista*".

- *READ <Arquivo>:* Esta mensagem é utilizada pelo *master* para indicar qual o nome do arquivo que determinado slave deve ler para buscar os números de entrada a serem ordenados. Uma vez que um *slave* recebe este comando, ele lê o arquivo citado localmente. Quando este termina de ler o arquivo, informa ao *master* com a seguinte mensagem: "*005 - Ok. O Arquivo <Nome> foi lido e a lista criada com <NumElementos> numeros*".

- *ORDER LIST:* Utilizado pelo *master* para solicitar aos *slaves* que comecem a ordenar as suas listas de números. Uma vez que o *slave* terminou de ordenar a lista, este informa ao *master* com a mensagem "*005 - Ok. A Lista foi Ordenada*".

- *GET SAMPLES:* Ao receber este comando, os *slaves* calculam sua lista de *samples* e as enviam para o *master*. Primeiramente o *slave* envia a mensagem "*022 - OK. Lista de Samples:*" e então começa a enviar os números. Quando a lista termina, este informa o *master* com a mensagem "*022 - Ok. Fim dos Samples*".

- *PUT PIVOTS:* Uma vez de posse da lista de *samples* de todos os *slaves* o *master* calcula a lista de *pivots* e as envia para todos os *slaves* através deste comando. Os *slaves* respondem imediatamente com a mensagem "*032 - OK. Pronto para receber Lista de Pivots*" e então o *master* começa a enviar a lista de números. Quando a lista termina, este último envia a mensagem "*END PIVOTS*" e, se ocorreu tudo certo, os *slaves* respondem com a mensagem "*034 - OK. Recebido a Lista*".

- *EXCH LIST:* É através deste comando que o *master* instrui os *slaves* a começaram a trocar as suas listas entre si. Esta parte, por ser uma das mais complicadas do algoritmo, será explicada com mais detalhes na Seção 5.3.2. Uma vez que o *slave* atual enviou todas as partes de sua lista para os respectivos *slaves* este responde ao master com a mensagem "*044 - Ok. Listas Trocadas*".

- *PUT EXCH LIST <SlaveID>:* Este comando é o único executado em um *slave* por outro *slave*. O processo é melhor explicado na Seção 5.3.2, mas este comando é utilizado por um *slave*(SlaveID) para colocar parte de sua lista no *slave* atual como parte da troca das listas. Então o *slave* atual responde com a mensagem "*032 - OK. Pronto para receber Lista de Troca*" e o outro *slave* começa a enviar a lista de números. Quando termina, o comando "*END EXCH LIST*" é recebido pelo *slave* atual e este, se tudo ocorreu certo, com a mensagem "*034 - OK. Recebido a Lista*".

- *GET EXCH LIST:* Este comando é enviado pelo *master* para um determinado *slave* para solicitar que ele envie a lista de números que ele trocou com os outros *slaves*. Então o *slave* atual responde com a mensagem "*052 - OK. Lista Trocada:*" e começa a enviar a lista de números. Quando a lista termina, o *slave* envia a mensagem "*052 - Ok. Fim da Lista de Troca*".

- *CLEAN UP:* Como estamos lidando com listas muito grandes, este comando foi implementado para que o *master* enviasse para os *slaves* solicitando que estes limpem da memória todas as listas criadas. Após a limpeza, os *slaves* respondem: "*001 - Tudo Limpo*".

- *QUIT:* O último comando enviado por um *master* aos *slaves* é o comando de saída. Este comando faz o *slave* fechar a conexão com o *master*. O *slave* continua na memória aguardando novos *masters*.

- *KILL:* Este comando tem o mesmo objetivo que o anterior. A diferença é que após desconectar, o *slave* é terminado.

5.3.2 A Troca das Listas entre os *Slaves*

Na terceira parte do algoritmo PSRS, cada processador i mantém sua i-ésima partição e envia a j-ésima partição para o processador j. Em outras palavras, cada processador mantém uma partição e envia $p-1$ partições para os outros processadores.

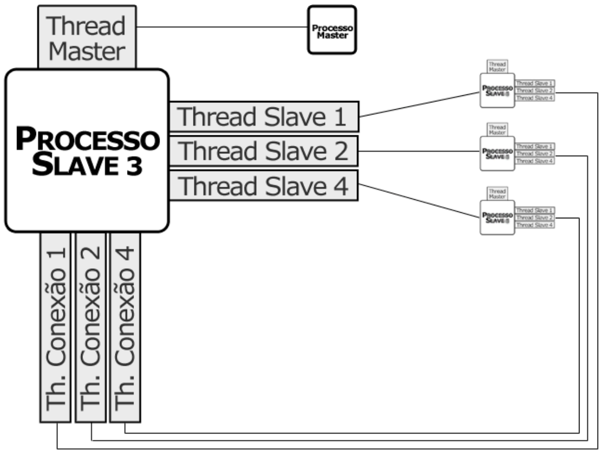

Figura 2: *Threads* **em execução em um determinado** *slave* **na fase de troca de listas.**

Esta parte foi a mais complicada de ser implementada, visto que um processo *slave* vai estar conectado com o *master*, se conectar em todos os outros *slaves* para enviar partes de sua lista e, tudo isso ao mesmo tempo, ainda deve receber as listas enviadas pelos outros *slaves*.

Para a implementação desta fase, foi feito um uso maciço de *threads*. Ao receber o comando *"EXCH LIST"*, cada processo *slave* abre um *thread* para cada *slave* que este irá enviar partes de sua lista (ou seja, $p-1$ *threads*). Além disso, cada processo *slave* irá receber $p-1$ conexões dos outros *slaves* neste mesmo tempo, abrindo mais $p-1$ *threads* no *slave*. A Figura 2 ilustra o que foi falado.

Neste momento, estima-se que o uso da rede seja levado ao extremo, visto que cada *thread* de cada processo *slave* está tentando utilizar toda a capacidade da rede.

5.3.3 Implementação do *Master*

O *master* foi o processo mais simples de ser implementado. Basicamente o que este faz é abrir *threads* para enviar os comandos simultaneamente para todos os *slaves* e esperar as respostas destes últimos.

Algumas funções que este processo executa além dos comandos é ler, se necessário, a lista de números localmente para ser enviado aos *slaves*, calcular o tempo de execução do programa, gerar a lista de *pivots*, concatenar as listas finais dos *slaves* e, por último, verificar se o resultado da lista obtida está realmente ordenado e possui o número certo de elementos.

Sendo assim, os seguintes passos são executados pelo *master*:

1. *Inicia cálculo do tempo;*

2. *Lê a lista de números:* Esta implementação tem a funcionalidade de fazer o *master* ler a lista de números a ser ordenado localmente e transferí-la, já particionada em p partes, para os p *slaves*. Esta função não foi utilizada nos experimentos realizados, pois a leitura das massas de dados foi feita localmente pelos *slaves*. A função executada é a `lerSlavesArq`.

3. *Lê a lista de slaves:* Neste passo o *master* lê o arquivo passado como parâmetro da linha de comando e cria a sua lista de *slaves*. Executa a função `lerSlavesArq`.

4. *Conecta aos slaves:* Executa a função `conectaSlaves` para conectar em cada um dos processos *slaves*. O resultado da conexão é salva na variável `socketFD` da estrutura `HostSlave` armazenada em cada elemento do vetor de *slaves*.

5. *Identifica os slaves:* Envia para cada um dos *slaves* a sua identificação. Este passo é executado ao mesmo tempo para todos os *slaves* com a utilização de *threads*. Os *threads* criados executam a função `enviarIDsThread`.

6. *Envia a lista de slaves:* Executa o *thread* `enviarSlavesThread` para enviar a lista de *slaves* para os *slaves*.

7. *Envia ou pede para ler a lista de números:* Neste passo o *master* ou envia a lista local de números já particionada para os *slaves* ou pede para que estes leiam tais números de um arquivo local. Executa o *thread* `enviarListaThread` ou o `enviarListaArqThread`.

8. *Solicitando a ordenacao das listas:* Executa o *thread* `execOrdenaListaThread` para solicitar aos *slaves* que estes comecem a ordenar a lista de números.

9. *Requisita os samples:* solicita dos *slaves* a lista de *samples* destes. Executa o *thread* `getSamplesThread`. Em seguida o *master* concatena as listas recebidas e ordena a lista concatenada.

10. *Geração dos pivots:* Executa a função `gerarPivots` para gerar a lista de *pivots*.

11. *Envia a lista de pivots para os slaves:* Executa o *thread* `enviarPivotsThread` para executar esta operação.

12. *Solicita que as listas sejam trocadas:* Neste passo o *master* solicita que todos os *slaves* comecem a trocar suas listas ao mesmo tempo. Executa o *thread* `execTrocaListaThread`.

13. *Solicita o merge das listas:* Neste passo o *master* solicita que todos os *slaves* façam o merge das listas recebidas na fase anterior. Executa o *thread* `execMergeListasThread`.

14. *Requisita as listas trocadas:* Uma vez que os *threads* do passo anterior tenham terminado, o *master* sabe que os *slaves* estão com as listas prontas. Ele as solicita neste passo executando o *thread* `getListasTrocadasThread`.

15. *Concatena as listas recebidas:* Executa a função `joinListas` para unificar as listas recebidas pelos *slaves* em uma só lista.

16. *Verifica a ordenação:* Uma função chamada `verificaOrdenacao` foi desenvolvida para fazer uma verificação na lista final e verificar se esta está realmente ordenada. Caso não esteja, um erro é mostrado na tela.

17. *Limpa a memória dos slaves:* Executa o *thread* `cleanUpThread` para solicitar aos *slaves* que limpem sua memória.

18. *Desconecta:* Desconecta dos *slaves* através da execução da função `desconectaSlaves`.

5.3.4 Implementação do *Slave*

A implementação do processo *slave*, apesar de não ter sido tão simples, não tem muito o que ser explicado. Basicamente este processo abre uma porta TCP e fica esperando pelo *master*. Uma vez que este último tenha se conectado, o *slave* fica executando cada um dos comandos recebidos.

Os comandos aceitos pelos *slaves* são citados e explicados rapidamente na Seção 5.3.1. E a parte mais complicada deste processo, que é a troca de listas entre os *slaves* foi explicada na Seção 5.3.2.

15	46	48	93	39	6	72	91	14	36	69	40	89	61	97	12	21	54	53	97	84	58	32	27	33	72	20

Figura 3: **Massa de dados para testes e exemplificação da implementação.**

6	14	15	**39**	46	48	**72**	91	93	**12**	21	36	**40**	54	61	**69**	89	97	**20**	27	32	**33**	53	58	**72**	84	97

Figura 4: **Ordenação das listas dos *slaves* e seleção das amostras.**

5.3.5 Execução Passo-a-Passo de um Exemplo Simples

Para ilustrar o funcionamento da implementação bem como realizar um teste bem simples nele, considere a massa de dados da Figura 3. Estaremos considerando também a presença de três processos *slaves*.

Considerando que estes números estão em um arquivo, o processo *master* inicialmente irá ler tal arquivo e colocar os números na memória. Este processo, então, irá dividir a lista em três partes iguais, se conectar nos processos *slaves* e enviar para cada um destes processos uma parte da lista. Cada processo irá receber $\frac{n}{p} = \frac{27}{3} = 9$ números. Em seguida, cada processo *slave* irá ordenar a lista recebida e selecionar p números como sendo de amostras (ver Figura 4). Assim termina a primeira fase do algoritmo.

Na segunda fase, o processo *master* recupera tais listas de amostras dos *slaves*, as concatena, ordena e seleciona $p-1 = 2$ números que formam a lista de *pivots* (ver Figura 5). Neste momento o *master* envia esta lista de *pivots* para os *slaves* que então dividem suas listas com base nestes *pivots*.

Na terceira fase, o *master* envia um comando aos *slaves* para que estes comecem a trocar suas listas entre si.

Quando a troca das listas termina, inicia a quarta fase, onde cada *slave* concatena as listas recebidas com a parte de sua lista que permaneceu no processo. Após esta fase o processo *master* solicita as listas dos *slaves* e então faz a concatenação das mesmas em uma lista totalmente ordenada (ver Figura 6) e centralizada.

A Figura 7 mostra a saída do processo *master* implementado utilizando o nível de *debug* 5. Neste nível, diversas mensagens de depuração são exibidas, de forma que apenas as mais relevantes foram mostradas na figura citada. Para facilitar a visualização, as partes mais importantes foram destacadas. Perceba que para melhor entender o funcionamento dos passos do algoritmo implementado, todos os passos foram temporizados, de modo a identificar quais são os mais custosos e que demandam mais tempo. Estas informações serão comentadas na Seção 6, onde os testes e resultados obtidos são estudados.

Já as Figuras 8, 9 e 10 mostram a saída dos processos *Slave1*, *Slave2* e *Slave3*, respectivamente. Novamente, muitas mensagens de depuração foram mostradas no nível de *debug* definido, de forma que apenas as mais relevantes foram mostradas. Dentre estas mensagens, as mais importantes estão em negrito. Note que, devido ao fato de se utilizar extensivamente a tecnologia de *threads*, as mensagens não são exibidas sempre na mesma ordem entre as execuções dos *slaves*. Note também que as mensagens dos *slaves* mostram os comandos explicados na Seção 5.3.1 enviados pelo *master*.

6	12	20	**33**	39	40	**69**	72	72

Figura 5: **Lista de amostras e a seleção dos *pivots*.**

6	12	14	15	20	21	27	32	33	36	39	40	46	48	54	54	58	61	69	72	72	84	89	91	93	97	97

Figura 6: **Listas ordenadas nos *slaves* e concatenada no *master*.**

```
[horacio@SixFeet Implementacao]$ ./PaaTP4.exe -s HostsClientes
[main] Iniciando PaaTP4: QuickSort Paralelo ..
[serverMain] Iniciando Master ..
[serverMain]     Criando Lista de Numeros ..
[print Time]        --> Tempo Ultimo: 0.000001 Segundos
[serverMain]     Lendo numeros ..
[print Time]        --> Tempo Ultimo: 0.000077 Segundos
[serverMain]     Lendo Slaves de HostsClientes ..
[abreArquivo] Abrindo Arquivo HostsClientes ..
[print Time]        --> Tempo Ultimo: 0.000110 Segundos
[serverMain]     Conectando a Slaves ..
[print Time]        --> Tempo Ultimo: 0.001067 Segundos
[serverMain]     Identificando Slaves ..
[print Time]        --> Tempo Ultimo: 0.000569 Segundos
[serverMain]     Enviando Lista de Slaves ..
[print Time]        --> Tempo Ultimo: 0.037008 Segundos
[serverMain]     Enviando Lista de Numeros ..
[print Time]        --> Tempo Ultimo: 0.039675 Segundos
[serverMain]     Solicitando a Ordenacao das Listas ..
[print Time]        --> Tempo Ultimo: 0.000403 Segundos
[serverMain]     Requisitando Samples ..
[enviaSamplesThread] Thread 0. Lista de Samples: 6 39 72
[enviaSamplesThread] Thread 1. Lista de Samples: 12 40 69
[enviaSamplesThread] Thread 2. Lista de Samples: 20 33 72
[joinSamples] Lista completa de Samples: 6 12 20 33 39 40 69 72 72
[print Time]        --> Tempo Ultimo: 0.039705 Segundos
[serverMain]     Gerando Pivots ..
[GeraPivots] Pivots Gerados: 33 69
[print Time]        --> Tempo Ultimo: 0.000013 Segundos
[serverMain]     Enviando Lista de Pivots ..
[print Time]        --> Tempo Ultimo: 0.040435 Segundos
[serverMain]     Solicitando Troca de Listas ..
[print Time]        --> Tempo Ultimo: 0.041823 Segundos
[serverMain]     Solicitando Merge das Listas ..
[print Time]        --> Tempo Ultimo: 0.000751 Segundos
[serverMain]     Requisitando Listas Trocadas ..
[getListasTrocadasThread] Thread 0. Lista Trocada: 6 12 14 15 20 21 27 32 33
[getListasTrocadasThread] Thread 1. Lista Trocada: 36 39 40 46 48 53 54 58 61 69
[getListasTrocadasThread] Thread 2. Lista Trocada: 72 72 84 89 91 93 97 97
[print Time]        --> Tempo Ultimo: 0.037224 Segundos
[serverMain]     Unindo as Listas Trocadas ..
[print Time]        --> Tempo Ultimo: 0.000043 Segundos
[serverMain]     Verificando Ordenacao da Lista ..
[verificaOrdenacao]      Numero de Elementos: 27
[verificaOrdenacao]      Ordenacao OK !! !!
[print Time]        --> Tempo Ultimo: 0.000004 Segundos
[serverMain]     LISTA FINAL:
[printLista] Lista: 6 12 14 15 20 21 27 32 33 36 39 40 46 48 53 54 58 61 69 72 72 84 89 91 93 97 97
[serverMain]     Limpando Memoria dos Slaves ..
[print Time]        --> Tempo Ultimo: 0.000527 Segundos
[serverMain]     Desconectando dos Slaves ..
[print Time]        --> Tempo Ultimo: 0.000277 Segundos
[print Time]        --> Tempo Final:  0.239721 Segundos
```

Figura 7: **Saída do processo *master* na execução do exemplo mostrado.**

```
[horacio@SixFeet Implementacao]$ ./PaaTP4.exe -c 5001
[main] Iniciando PaaTP4: QuickSort Paralelo ..
[slaveMain] Iniciando Slave ..
[slaveMain]  Gerenciando Conexoes na porta 5001 ..
[gerenciaConecoes ]Conexao Recebida .. Vou abrir tread ..
[gerenciaConexoes] Tread 0 Aberto .. Vou continuar ..
[-1][funcaoFilho]    Conexão recebida !!
[0][respSolicitacao] YOU ARE Recebido. ID setado para 0 !!
[0][respSolicitacao] PUT SLAVES Recebido.
[0][lerSlavesConexao] Adicionado Slave 0, 127.0.0.1, 5001 !!
[0][lerSlavesConexao] Adicionado Slave 1, 127.0.0.1, 5002 !!
[0][lerSlavesConexao] Adicionado Slave 2, 127.0.0.1, 5003 !!
[0][respSolicitacao] PUT LIST Recebido.
[0][lerListaConexao] Recebido a lista com 9 elementos !!
[0][respSolicitacao] ORDER LIST Recebido. Ordenando a Lista ..
[0][respSolicitacao] A Lista foi Ordenada.
[0][respSolicitacao] GET SAMPLES Recebido. Enviando Samples ..
[0][gerarSamples] Lista de Numeros: 6 14 15 39 46 48 72 91 93
[0][gerarSamples] Lista de Samples: 6 39 72
[0][respSolicitacao] PUT PIVOTS Recebido. Recebendo Pivots ..
[0][receberPivots] Lista de Pivots 33, 69 ..
[0][respSolicitacao] EXCH LIST Recebido. Trocando Listas ..
[0][Thd 0][trocaListaThread] Thread 0 Criado. Enviando Lista de Numeros entre 0 e 33 para 0.
[0][Thd 0][trocaListaThread] Recebido lista de 0: [printLista] Lista: 6 14 15
[gerenciaConecoes ]Conexao Recebida .. Vou abrir tread ..
[0][funcaoFilho]    Conexão recebida !!
[gerenciaConexoes] Tread 1 Aberto .. Vou continuar ..
[0][respSolicitacao] PUT EXCH LIST Recebido. Recebendo Lista de Troca de 1 ..
[0][Thd 1][trocaListaThread] Thread 1 Criado. Enviando Lista de Numeros entre 33 e 69 para 1.
[gerenciaConecoes ]Conexao Recebida .. Vou abrir tread ..
[0][funcaoFilho]    Conexão recebida !!
[0][respSolicitacao] PUT EXCH LIST Recebido. Recebendo Lista de Troca de 2 ..
[gerenciaConexoes] Tread 2 Aberto .. Vou continuar ..
[joinThreads] Esperando pelo processo 0 ..
[joinThreads] Esperando pelo processo 1 ..
[0][Thd 2][trocaListaThread] Thread 2 Criado. Enviando Numeros entre 69 e 2147483647 para 2.
[0][receberTrocaLista]Recebido a lista com 2 elementos de 1!!
[0][receberTrocaLista] Recebido lista de 1 COM 2 ELEMENTOS: 12 21
[0][respSolicitacao] QUIT Recebido. Desconectando !!
[0][Thd 1][trocaListaThread] Ternminei a troca ..
[joinThreads] Esperando pelo processo 2 ..
[0][receberTrocaLista] Recebido a lista com 4 elementos de 2!!
[0][receberTrocaLista] Recebido lista de 2 COM 4 ELEMENTOS: 20 27 32 33
[0][respSolicitacao] QUIT Recebido. Desconectando !!
[0][Thd 2][trocaListaThread] Ternminei a troca ..
[0][respSolicitacao] MERGE LISTS Recebido. Mergindo as Listas ..
[0][gerarListaTrocada] Lista Final Trocada: 6 12 14 15 20 21 27 32 33
[0][respSolicitacao] A Lista foi Ordenada.
[0][respSolicitacao] GET EXCH LIST Recebido. Enviando Lista Final ..
[0][enviarListaTrocada]  Enviando lista Trocada ..
[0][respSolicitacao] CLEAN UP Recebido. Limpando !!
```

Figura 8: **Saída do processo *slave 1* na execução do exemplo mostrado.**

```
[horacio@SixFeet Implementacao]$ ./PaaTP4.exe -c 5002
[main] Iniciando PaaTP4: QuickSort Paralelo ..
[slaveMain] Iniciando Slave ..
[slaveMain]  Gerenciando Conexoes na porta 5002 ..
[gerenciaConecoes ]Conexao Recebida .. Vou abrir tread ..
[-1][funcaoFilho]     Conexão recebida !!
[gerenciaConexoes] Tread 0 Aberto .. Vou continuar ..
[1][respSolicitacao] YOU ARE Recebido. ID setado para 1 !!
[1][respSolicitacao] PUT SLAVES Recebido.
[1][lerSlavesConexao] Adicionado Slave 0, 127.0.0.1, 5001 !!
[1][lerSlavesConexao] Adicionado Slave 1, 127.0.0.1, 5002 !!
[1][lerSlavesConexao] Adicionado Slave 2, 127.0.0.1, 5003 !!
[1][respSolicitacao] PUT LIST Recebido.
[1][lerListaConexao] Recebido a lista com 9 elementos !!
[1][respSolicitacao] ORDER LIST Recebido. Ordenando a Lista ..
[1][respSolicitacao] A Lista foi Ordenada.
[1][respSolicitacao] GET SAMPLES Recebido. Enviando Samples ..
[1][gerarSamples] Lista de Numeros: 12 21 36 40 54 61 69 89 97
[1][gerarSamples] Lista de Samples: 12 40 69
[1][respSolicitacao] PUT PIVOTS Recebido. Recebendo Pivots ..
[1][receberPivots] Lista de Pivots 33, 69 ..
[1][respSolicitacao] EXCH LIST Recebido. Trocando Listas ..
[1][Thd 0][trocaListaThread] Thread 0 Criado. Enviando Lista de Numeros entre 0 e 33 para 0.
[gerenciaConecoes ]Conexao Recebida .. Vou abrir tread ..
[1][Thd 1][trocaListaThread] Thread 1 Criado. Enviando Lista de Numeros entre 33 e 69 para 1.
[1][Thd 1][trocaListaThread] Recebido lista de 1: 36 40 54 61 69
[1][Thd 1][trocaListaThread] Ternminei a troca ..
[1][funcaoFilho]     Conexão recebida !!
[gerenciaConexoes] Tread 1 Aberto .. Vou continuar ..
[gerenciaConecoes ]Conexao Recebida .. Vou abrir tread ..
[1][respSolicitacao] PUT EXCH LIST Recebido. Recebendo Lista de Troca de 0 ..
[1][Thd 2][trocaListaThread] Thread 2 Criado. Enviando Numeros entre 69 e 2147483647 para 2.
[joinThreads] Esperando pelo processo 0 ..
[gerenciaConexoes] Tread 2 Aberto .. Vou continuar ..
[1][funcaoFilho]     Conexão recebida !!
[1][respSolicitacao] PUT EXCH LIST Recebido. Recebendo Lista de Troca de 2 ..
[1][receberTrocaLista] Recebido a lista com 3 elementos de 0!!
[1][receberTrocaLista] Recebido lista de 0 COM 3 ELEMENTOS: 39 46 48
[1][Thd 0][trocaListaThread] Ternminei a troca ..
[joinThreads] Esperando pelo processo 1 ..
[joinThreads] Esperando pelo processo 2 ..
[1][respSolicitacao] QUIT Recebido. Desconectando !!
[1][Thd 2][trocaListaThread] Ternminei a troca ..
[1][receberTrocaLista] Recebido a lista com 2 elementos de 2!!
[1][receberTrocaLista] Recebido lista de 2 COM 2 ELEMENTOS: 53 58
[1][respSolicitacao] QUIT Recebido. Desconectando !!
[1][respSolicitacao] MERGE LISTS Recebido. Mergindo as Listas ..
[1][gerarListaTrocada] Lista Final Trocada: 36 39 40 46 48 53 54 58 61 69
[1][respSolicitacao] A Lista foi Ordenada.
[1][respSolicitacao] GET EXCH LIST Recebido. Enviando Lista Final ..
[1][enviarListaTrocada]  Enviando lista Trocada ..
[1][respSolicitacao] CLEAN UP Recebido. Limpando !!
```

Figura 9: **Saída do processo *slave 2* na execução do exemplo mostrado.**

```
[horacio@SixFeet Implementacao]$ ./PaaTP4.exe -c 5003
[main] Iniciando PaaTP4: QuickSort Paralelo ..
[slaveMain] Iniciando Slave ..
[slaveMain]  Gerenciando Conexoes na porta 5003 ..
[gerenciaConecoes ]Conexao Recebida .. Vou abrir tread ..
[-1][funcaoFilho]    Conexão recebida !!
[gerenciaConexoes] Tread 0 Aberto .. Vou continuar ..
[2][respSolicitacao] YOU ARE Recebido. ID setado para 2 !!
[2][respSolicitacao] PUT SLAVES Recebido.
[2][lerSlavesConexao] Adicionado Slave 0, 127.0.0.1, 5001 !!
[2][lerSlavesConexao] Adicionado Slave 1, 127.0.0.1, 5002 !!
[2][lerSlavesConexao] Adicionado Slave 2, 127.0.0.1, 5003 !!
[2][respSolicitacao] PUT LIST Recebido.
[2][lerListaConexao] Recebido a lista com 9 elementos !!
[2][lerListaConexao] Final da Lista: END LIST ..
[2][respSolicitacao] ORDER LIST Recebido. Ordenando a Lista ..
[2][respSolicitacao] A Lista foi Ordenada.
[2][respSolicitacao] GET SAMPLES Recebido. Enviando Samples ..
[2][gerarSamples] Lista de Numeros: 20 27 32 33 53 58 72 84 97
[2][gerarSamples] Lista de Samples: 20 33 72
[2][respSolicitacao] PUT PIVOTS Recebido. Recebendo Pivots ..
[2][receberPivots] Lista de Pivots 33, 69 ..
[2][respSolicitacao] EXCH LIST Recebido. Trocando Listas ..
[2][Thd 0][trocaListaThread] Thread 0 Criado. Enviando Lista de Numeros entre 0 e 33 para 0.
[2][Thd 1][trocaListaThread] Thread 1 Criado. Enviando Lista de Numeros entre 33 e 69 para 1.
[2][Thd 2][trocaListaThread] Thread 2 Criado. Enviando Numeros entre 69 e 2147483647 para 2.
[2][Thd 2][trocaListaThread] Recebido lista de 2: 72 84 97
[2][Thd 2][trocaListaThread] Ternminei a troca ..
[gerenciaConecoes ]Conexao Recebida .. Vou abrir tread ..
[joinThreads] Esperando pelo processo 0 ..
[gerenciaConexoes] Tread 1 Aberto .. Vou continuar ..
[2][funcaoFilho]    Conexão recebida !!
[2][respSolicitacao] PUT EXCH LIST Recebido. Recebendo Lista de Troca de 1 ..
[gerenciaConecoes ]Conexao Recebida .. Vou abrir tread ..
[gerenciaConexoes] Tread 2 Aberto .. Vou continuar ..
[2][funcaoFilho]    Conexão recebida !!
[2][respSolicitacao] PUT EXCH LIST Recebido. Recebendo Lista de Troca de 0 ..
[2][receberTrocaLista] Recebido a lista com 2 elementos de 1!!
[2][receberTrocaLista] Recebido lista de 1 COM 2 ELEMENTOS: 89 97
[2][Thd 0][trocaListaThread] Ternminei a troca ..
[joinThreads] Esperando pelo processo 1 ..
[2][respSolicitacao] QUIT Recebido. Desconectando !!
[2][receberTrocaLista] Recebido a lista com 3 elementos de 0!!
[2][receberTrocaLista] Recebido lista de 0 COM 3 ELEMENTOS: 72 91 93
[2][Thd 1][trocaListaThread] Ternminei a troca ..
[joinThreads] Esperando pelo processo 2 ..
[2][respSolicitacao] QUIT Recebido. Desconectando !!
[2][respSolicitacao] MERGE LISTS Recebido. Mergindo as Listas ..
[2][gerarListaTrocada] Lista Final Trocada: 72 72 84 89 91 93 97 97
[2][respSolicitacao] A Lista foi Ordenada.
[2][respSolicitacao] GET EXCH LIST Recebido. Enviando Lista Final ..
[2][enviarListaTrocada]  Enviando lista Trocada ..
[2][respSolicitacao] CLEAN UP Recebido. Limpando !!
```

Figura 10: **Saída do processo *slave 3* na execução do exemplo mostrado.**

6 Testes e Resultados

Alguns testes foram realizados na Implementação para procurar identificar o quão rápido um algoritmo deste tipo é capaz de executar em uma máquina paralela MIMD do tipo estações de trabalho (*NOW - Network of Workstations*).

Além disso, um outro objetivo dos testes realizados foi de tentar entender melhor o funcionamento prático do algoritmos, especialmente em relação às suas diversas fases.

Desta forma, tempos de execução foram calculados levando-se em consideração:

- *número de processadores;*
- *tamanho do arquivo de entrada;* e
- *fases do algoritmo.*

tais variações são explicadas na Seção 6.2.

6.1 Validação da Implementação

O funcionamento da implementação foi mostrado e, de certa forma, validado na Seção 5.3.5. Porém, o exemplo mostrado é muito pequeno e não pode-se dizer, com certeza, que se a implementação funcionou para essa massa de dados então a mesma irá funcionar para todas as instâncias (*e.g.*, massa de 200MB).

Desta forma, uma maneira mais automática e rápida de se verificar o real funcionamento da implementação foi implementada. Trata-se da função `verificaOrdenacao`, que tem como entrada uma lista de números e gera como saída a veracidade da ordenação da lista bem como o número de elementos que a mesma possui.

Esta função foi desenvolvida levando-se em consideração o conceito de uma lista ordenada: seja uma lista $A[0..n]$, diz-se que esta lista está ordenada se $\forall a \in A, a_i \leq a_{i+1}$ para $0 \leq i < n$.

A implementação desta verificação encontra-se no Código 7. Esta função imprime o número de elementos da lista para que esta possa ser comparada com o número de elementos que era esperado. Desta forma, pode-se garantir que tem-se uma lista ordenada e que possui o número de elementos esperado.

Esta função foi muito útil, visto que encontrou-se casos durante a implementação do algoritmo onde este funcionava para exemplos simples e pequenos mas, mesmo chegando ao final do algoritmo, apresentava problemas de ordenação diante de massas de dados muito grandes.

6.2 Experimentos Realizados

Uma vez que os resultados de todos os experimentos foram verificados, pode-se prosseguir para a realização dos experimentos. Como mencionado anteriormente, os testes foram realizados levando-se em consideração:

- *Número de Processadores:* ver Seção 6.2.1.
- *Tamanho do Arquivo de Entrada:* ver Seção 6.2.2.
- *Fases do Algoritmo:* ver Seção 6.2.3.

Código 7 Função de verificação de ordenação de uma lista.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao verificaOrdenacao                              *
 *    Verifica se uma lista esta ordenada.               *
 *                                                       *
 *    @param lista Ponteiro para a Lista.                *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void verificaOrdenacao(TLista *lista) {
  unsigned long ultimo = 0;       // Ultimo Numero Verificado
  int i, ok = 1;                  // Contador e Se estah Ok

  // Verifica os Numeros
  for (i=0; i<lista->tamLista; i++) {
    if (ultimo > lista->numeros[i]) {
      if (DEBUG>3) printf("[ver.Ord.] Numero na posicao %d (%ld) maior na posicao %d (%ld) !!\n",
                                          i, lista->numeros[i], i-1, lista->numeros[i-1]);
      ok = 0;
    }
    ultimo = lista->numeros[i];
  }
  if (DEBUG>3) printf("[verificaOrdenacao]     Numero de Elementos: %d\n", lista->tamLista);
  if (ok  && DEBUG>3) printf("[verificaOrdenacao]     Ordenacao OK !! !!\n");
  if (!ok && DEBUG>2) printf("[verificaOrdenacao]     ERRO na ordem !! !!\n");
}
```

6.2.1 Variação do Número de Processadores, *Speedup* e Eficiência

Para melhor entender o funcionamento do algoritmo, este foi executado variando-se o número de processadores utilizados. O objetivo deste experimento é tentar entender como é que ocorre a variação do tempo de ordenação de uma lista de números diante de modificações no número de processadores utilizados como *slaves*. Queremos responder a perguntas como:

- *É viável aumentar o número de processadores?*
- *Em quanto diminui o tempo de ordenação de uma lista ao se duplicar o número de processadores disponíveis?*
- *Até quando o aumento do número de processadores resultará na diminuição do tempo de ordenação?*
- *Quanto a ordenação em paralelo é mais rápida que a ordenação seqüencial utilizando x processadores (Speedup)?*

Para a realização deste tipo de experimento, o tempo de execução da implementação foi verificada utilizando-se:

- *2 processadores;*
- *4 processadores;* e
- *1 processador utilizando o Quicksort Seqüencial.*

Este último, utilizando o Quicksort Seqüencial, foi necessário para o cálculo do *speedup*.

Como o tipo de máquina MIMD utilizada é rede de estações de trabalho, cada processador citado corresponde a uma máquina rodando o *slave* da implementação. As máquinas utilizadas foram as localizadas no laboratório da Sala 2012 do DCC/ICEx/UFMG e listadas na Tabela 1

Num	Nome	Objetivo da Máquina
1	nilo	Máquina *Master*
2	traira	Máquina *Slave* 1
3	lambari	Máquina *Slave* 2
4	niger	Máquina *Slave* 3
5	carpa	Máquina *Slave* 4

Tabela 1: **Máquinas utilizadas nos experimentos.**

Pentium III 1GHz com 216MB de Memória principal mais 512MB de memória *Swap*. Sistema Operacional Linux Red Hat 8 Kernel 2.4. Devido a falta de permissões, o ambiente X permaneceu na memória durante a execução dos experimentos.

Figura 11: **Configuração das máquinas utilizadas nos experimentos.**

Para o melhor conduzimento dos testes e obtenção de resultados confiáveis, a configuração das máquinas foram padronizadas e são mostradas na Figura 11.

Infelizmente não foi disponibilizados mais máquinas, o que impossibilitou a realização de experimentos com 8 ou até mesmo 16 processadores.

6.2.2 Variação do Tamanho da Massa de Dados

Para a realização dos experimentos, um arquivo de 200MB foi particionado e disponibilizado nas diversas máquinas disponíveis. Para melhor compreender o funcionamento do algoritmo novas massas de dados de teste foram criadas utilizando um gerador de números aleatórios.

O objetivo principal deste tipo de experimento é tentar entender o comportamento do algoritmo, em especial o seu tempo de execução, diante de massas de dados de tamanhos diferentes. Procura-se responder, assim, questões do tipo:

- *Qual o comportamento do tempo de execução do algoritmo ao aumentarmos continuamente o tamanho da entrada?*
- *Quanto tempo a mais o algoritmo leva para ordenar o dobro desta massa de dados?*
- *A partir de qual tamanho de entrada o algoritmo tem Speedups melhores?*
- *Vale a pena ordenar em paralelo massas de dados pequenas? E grandes?*

Para a realização deste tipo de experimento, além da massa de 200MB disponibilizada, novos arquivos foram gerados utilizando a opção "`-g`" da implementação, que tem por função gerar uma massa de dados de números aleatórios de tamanho especificado e salvar em um arquivo de saída também especificado.

As massas de dados que foram geradas encontram-se listadas na Tabela 2.

Infelizmente não foi possível gerar massas de dados maiores do que 200MB, visto que as máquinas possuem uma quantidade de memória limitada a 256MB. Caso arquivos maiores fossem utilizados, possivelmente haveria um acesso muito grande à memória *swap*, o que resultaria em resultados inesperados e não muito robustas.

Num	**Arquivo**	**Característica**
1	dados200MB.bin	`Arquivo de 200MB`
2	dados100MB.bin	`Arquivo de 100MB`
3	dados50MB.bin	`Arquivo de 50MB`
4	dados25MB.bin	`Arquivo de 25MB`

Tabela 2: **Massas de Dados utilizadas nos experimentos.**

6.2.3 Tempo em relação às Fases do Algoritmo

O algoritmo PSRS é composto por quatro fases (ver 5.3). Já a implementação realizada, conforme mencionado na Seção 5.3.3, é coordenada completamente pelo *master* que, por si, é composta por uma série de passos. Diversos passos na implementação corresponde a uma determinada fase do algoritmo.

Uma vez que o *master* controla cada passo executado pelos *slaves*, fica fácil de calcular o tempo de execução de cada um dos passos da implementação e, conseqüentemente, cada uma das fases do algoritmo.

O objetivo principal deste tipo de experimento e tentar entender melhor o algoritmo implementado através da análise de tempo de execução de cada uma de suas fases. Estamos procurando responder perguntas do tipo:

- *Quais as fases mais custosas do algoritmo?* Uma vez identificada os passos/fases que mais levam tempo para executar, pode-se priorizar os esforços nestas para otimizar suas execuções.
- *Qual o comportamento destas fases ao variar o número de processadores disponíveis?*
- *Qual o comportamento destas fases ao variar o tamanho do arquivo de entrada?*

A Tabela 3 mostra cada um dos passos da implementação bem como a fase a que cada um destes passos pertence. Estas informações serão muito utilizadas ao mostrar os resultados. Além disso, três novas fases foram incluídas no algoritmo:

- *Fase 0:* Corresponde a uma fase anterior do algoritmo necessária para preparar o ambiente de estações de trabalho utilizada.
- *Fase 5:* Corresponde a uma fase posterior do algoritmo, onde o *master* coleta cada uma das listas ordenadas nos *slaves*, e as concatena, gerando uma lista apenas completamente ordenada e centralizada.
- *Verificação:* Mais um passo incluído na implementação para verificar se a lista recebida pelo *master* está realmente ordenada e que possui o número de elementos esperado.

6.2.4 Execuções da Implementação

A implementação do Algoritmo PSRS foi executada várias vezes variando-se o número de processadores e o tamanho do arquivo de entrada. Para melhor identificar cada uma das execuções realizadas, estas foram identificadas por um número. As execuções realizadas estão listadas na Tabela 14.

Fases	**Passos da Implementação**
Fase 0	1 - Inicia cálculo do tempo
	2 - Lê Lista de números
	3 - Lê lista de *slaves*
	4 - Conecta nos *slaves*
	5 - Identifica os *slaves*
	6 - Envia a lista de *slaves*
Fase 1	7 - Solicita a leitura da lista de números
	8 - Solicita a ordenação das listas
Fase 2	9 - Requisita os *samples*
	10 - Gera os *pivots*
	11 - Envia a lista de *pivots*
Fase 3	12 - Solicita a troca das listas
Fase 4	13 - Solicita o merge das listas
Fase 5	14 - Requisita as listas trocadas
	15 - Concatena as listas recebidas
Verific.	16 - Verifica a ordenação

Tabela 3: **Fases do algoritmo e passos da implementação correspondentes.**

Id	**Característica da Execução**
1	Ordenação de uma massa de dados de 200MB utilizando 2 processadores
2	Ordenação de uma massa de dados de 200MB utilizando 4 processadores
3	Ordenação de uma massa de dados de 200MB utilizando Quicksort Seqüencial
4	Ordenação de uma massa de dados de 100MB utilizando 2 processadores
5	Ordenação de uma massa de dados de 100MB utilizando 4 processadores
6	Ordenação de uma massa de dados de 100MB utilizando Quicksort Seqüencial
7	Ordenação de uma massa de dados de 50MB utilizando 2 processadores
8	Ordenação de uma massa de dados de 50MB utilizando 4 processadores
9	Ordenação de uma massa de dados de 50MB utilizando Quicksort Seqüencial
10	Ordenação de uma massa de dados de 25MB utilizando 2 processadores
11	Ordenação de uma massa de dados de 25MB utilizando 4 processadores
12	Ordenação de uma massa de dados de 25MB utilizando Quicksort Seqüencial

Tabela 4: **Execuções realizadas da implementação.**

Num. Proc.	Tam. Arq.			
	25MB	*50MB*	*100MB*	*200MB*
2	13.117598	27.963143	53.925526	121.369432
4	9.502362	18.447773	37.882227	81.092082
Seq	24.750607	46.200973	107.930634	195.692362

Tabela 5: **Dados de tempo de execução da implementação variando-se o número de processadores.**

6.3 Resultados Obtidos

A seção anterior definiu os experimentos realizados. Esta seção está dividida da mesma forma que esta última mostrando os resultados obtidos dos experimentos com o acréscimo de comentários a respeito da monitoração das máquinas enquanto estas executavam a implementação:

- *Variação do Número de Processadores e Speedup:* Ver Seção 6.3.1.
- *Variação do Tamanho da Massa de Dados:* Ver Seção 6.3.2.
- *Tempo em relação às Fases do Algoritmo:* Ver Seção 6.3.3.
- *Execuções da Implementação:* Ver Seção 6.3.4.
- *Resultado da Monitoração do Comportamento das Máquinas:* Ver Seção 6.3.5.

6.3.1 Variação do Número de Processadores, *Speedup* e Eficiência

Conforme mencionado, o objetivo deste experimento é tentar entender como é que ocorre a variação do tempo de ordenação de uma lista de números diante de modificações no número de processadores utilizados como *slaves*.

Infelizmente o número de máquinas disponíveis foi reduzido, possibilitando apenas o estudo comparativo de 2 e quatro processadores.

Desta forma, o gráfico ilustrado na Figura 12 mostra o tempo de execução da implementação[5] para cada tamanho de entrada variando-se o número de processadores utilizados na ordenação paralela.

Os dados de tempo ilustrados no gráfico encontram-se listados na Tabela 5 em conjunto com o tempo necessário para os arquivos serem ordenados utilizando o Quicksort Seqüencial.

Os conceitos de *speedup* e eficiência foram mostrados na Seção 2. Através dos dados mostrados na Tabela 5 é possível calcular o *speedup* e a eficiência do algoritmo nos diversos casos. Estes valores são mostrados na Tabela 6.

O que pode-se observar com os poucos dados obtidos com os experimentos e com os cálculos realizados é que quanto maior o arquivo a ser ordenado, maior proveito da paralelização é obtido. Pode-se observar isso observando que a linha mais inclinada no gráfico em estudo é a linha do maior arquivo utilizado nos experimentos (200MB).

[5] A não ser que seja mencionado o contrário, os tempos considerados tanto nos gráficos quanto nas tabelas correspondem apenas aos passos 1-4 do algoritmo excluindo -se, assim, os passos de unificação da lista no *master* e também de verificação da ordenação dos resultados. Obviamente isso não se aplica na execução do Quicksort Seqüencial.

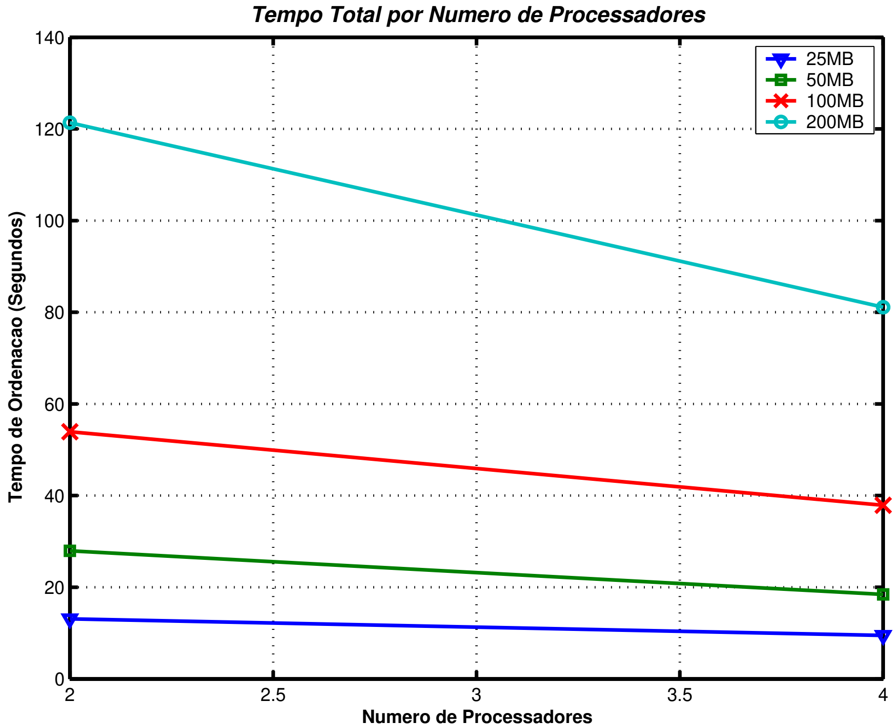

Figura 12: **Tempo de ordenação de cada tamanho de entrada variando-se o número de processadores.**

Num. Proc.	**Tam. Arq.**							
	25MB		*50MB*		*100MB*		*200MB*	
	Speedup	*Eficiência*	*Speedup*	*Eficiência*	*Speedup*	*Eficiência*	*Speedup*	*Eficiência*
2	1.886824	0.943412	1.652209	0.826104	2.001475	1.000737	1.612369	0.806184
4	2.604679	0.651169	2.504420	0.626105	2.849110	0.712277	2.413211	0.603302

Tabela 6: **Cálculo do *speedup* e da eficiência.**

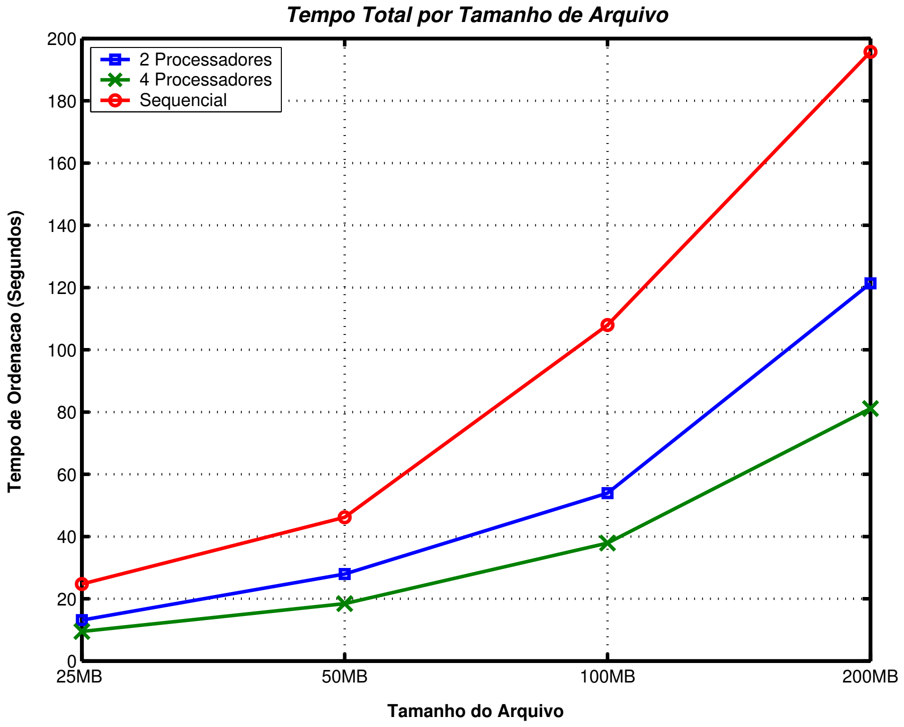

Figura 13: **Tempo de ordenação de cada número de processadores variando-se o tamanho da massa de dados utilizada como entrada da ordenação paralela.**

Pode-se observar também que em todos os casos foi-se viável aumentar o número de processadores, visto que todos ordenaram o arquivo em menos tempo ao pular de 2 para 4 processadores. Ao se duplicar o número de processadores, percebeu-se uma diminuição de 33.2% no tempo necessário para se ordenar a massa de dados de 200MB. Para a massa de dados de 100MB essa diminuição foi de 29.7%. Para a de 50MB houve uma diminuição de 34.0% e, por último, para a massa de dados de 25MB o tempo de ordenação caiu 27.5%.

Devido ao pequeno número de máquinas disponíveis para os experimentos, não foi possível realizar a execução da implementação com um número maior de processadores. Desta forma, não pode-se dizer em números o comportamento da implementação ao aumentarmos novamente o número de processadores. Não pode-se dizer também até quando o aumento do número de processadores resultará na diminuição do tempo de ordenação da massa de dados.

6.3.2 Variação do Tamanho da Massa de Dados

Conforme mencionado anteriormente, o objetivo deste experimento é tentar entender o comportamento do algoritmo, em especial o seu tempo de execução, diante de massas de dados de tamanhos diferentes.

Desta forma, o gráfico ilustrado na Figura 13 mostra o tempo de execução da implementação para cada número de processadores variando-se o tamanho da massa de dados utilizados como entrada da ordenação paralela.

Os dados de tempo ilustrados no gráfico encontram-se listados na Tabela 7 em conjunto com o tempo necessário para os arquivos serem ordenados utilizando o Quicksort Seqüencial.

Analisando-se o gráfico e os dados, percebe-se que quanto menor o número de processa-

Tam. Arq.	**Num. Processadores**		
	2	*4*	*Seqüencial*
25MB	13.117598	9.502362	24.750607
50MB	27.963143	18.447773	46.200973
100MB	53.925526	37.882227	107.930634
200MB	121.369432	81.092082	195.692362

Tabela 7: **Dados de tempo de execução da implementação variando-se o tamanho da massa de dados de entrada.**

dores, mais o algoritmo sente um aumento no tamanho da entrada. Percebe-se novamente também que em todos os tamanhos de entrada para o algoritmo paralelo houve-se ganho de tempo em relação ao algoritmo seqüencial.

6.3.3 Tempo em relação às Fases do Algoritmo

Nas seções anteriores, foram estudadas o tempo de execução do algoritmo como um todo. Nesta seção, será estudado o comportamento das diversas fases do algoritmo e analisar os seus comportamentos diante de mudanças no número de processadores e também no tamanho da entrada.

O objetivo principal deste tipo de experimento e tentar entender melhor o algoritmo implementado através da análise de tempo de execução de cada uma de suas fases.

Desta forma, os gráficos das Figuras 14 e 15 mostram o tempo de execução de cada uma das fases do algoritmo variando-se o tamanho da massa de dados de entrada para execuções utilizando-se 2 e 4 processadores, respectivamente. As Tabelas 8 e 9 mostram os dados para esses dois gráficos.

Pode-se observar nestes gráficos que a fase que mais consome tempo é a Fase 1, onde a leitura do arquivo de entrada é realizada, é feita a ordenação local e em seguida são selecionados os *samples*. A fase 5, onde as listas finais são enviadas para o *master* é a segunda que toma mais tempo, mas como este tempo não é considerado no tempo do algoritmo, este não será comentado. Logo, a segunda fase que toma mais tempo do algoritmo é a Fase 3, onde há uma troca de listas entre os processos *slaves*.

Os gráficos mostrados até o momento nesta Seção permitem uma análise e comparação entre os tamanhos diversos de entrada. Para permitir uma melhor comparação de tempo das fases do algoritmo levando-se em consideração o número de processadores utilizados, novos gráficos foram gerados, um para cada tamanho de entrada. Desta forma, os gráficos das Figuras 16, 17, 18 e 19 mostram estes gráficos para uma entrada de 25MB, 50MB, 100MB e 200MB respectivamente.

As Tabelas 10, 11, 12, e 13 mostram os dados utilizados na geração dos gráficos das figuras 16, 17, 18, e 19.

Percebe-se nestes gráficos que a fase que mais é afetada pelo número de processadores é a Fase 1. A Fase 1 fica mais rápida por dois motivos: primeiro que a leitura do arquivo está sendo realizada em paralelo. Desta forma, ao invés de se ler um arquivo de 200MB, estamos lendo um arquivo de 100MB por 2 processadores em paralelo ou lendo um arquivo de 50MB por 4 processadores em paralelo. O segundo motivo é a ordenação da lista de números que também é completamente favorecida pelo paralelismo.

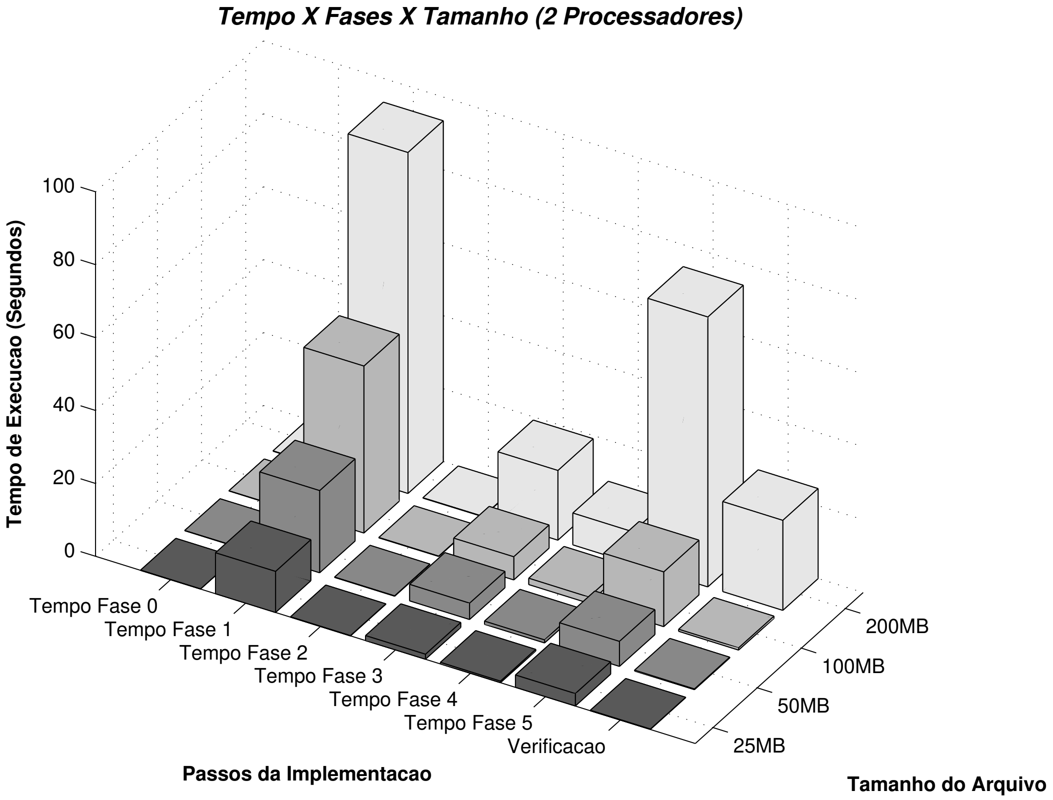

Figura 14: **Tempo de execução de cada uma das fases do algoritmo variando-se o tamanho da massa de dados de entrada utilizando 2 processadores.**

Fase	**Tam. Arquivo**			
	25	*50*	*100*	*200*
Tempo Fase 0	0,065771	0,064312	0,063719	0,115624
Tempo Fase 1	11,244844	22,535747	45,800661	93,474848
Tempo Fase 2	0,085866	0,084841	0,086368	0,085280
Tempo Fase 3	1,372957	4,508423	6,316336	19,096014
Tempo Fase 4	0,413931	0,834132	1,722161	8,713290
Tempo Fase 5	3,378792	6,816228	14,994674	74,008666
Tempo Verificacao	0,107507	0,206540	0,552540	24,677641

Tabela 8: **Dados de tempo de execução de cada uma das fases do algoritmo variando-se o tamanho da massa de dados de entrada utilizando 2 processadores.**

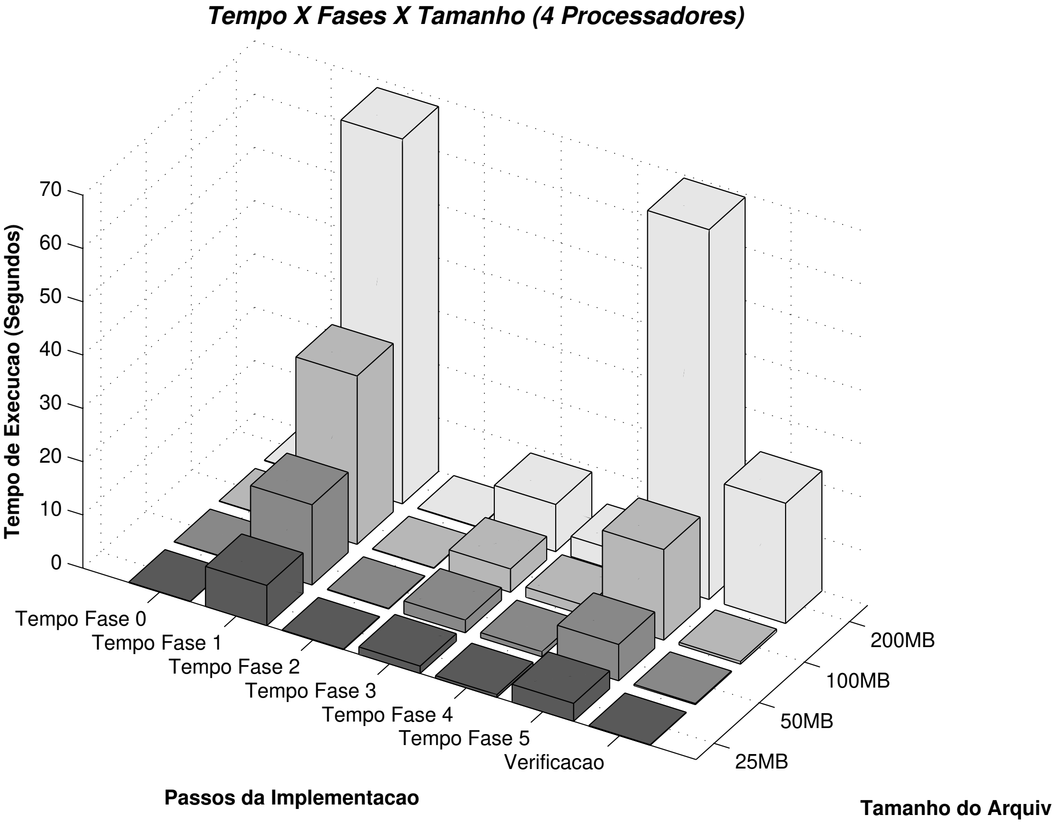

Figura 15: **Tempo de execução de cada uma das fases do algoritmo variando-se o tamanho da massa de dados de entrada utilizando 4 processadores.**

Fase	**Tam. Arquivo**			
	25	*50*	*100*	*200*
Tempo Fase 0	0,075929	0,076720	0,073414	0,10165
Tempo Fase 1	7,528201	15,045186	31,581842	68,434189
Tempo Fase 2	0,089953	0,089174	0,090190	0,09568
Tempo Fase 3	1,393482	2,377463	4,349307	8,824651
Tempo Fase 4	0,490726	0,935950	1,860888	3,737562
Tempo Fase 5	3,361111	6,800524	16,996741	69,390764
Tempo Verificacao	0,108772	0,206709	0,465835	22,538833

Tabela 9: **Dados de tempo de execução de cada uma das fases do algoritmo variando-se o tamanho da massa de dados de entrada utilizando 4 processadores.**

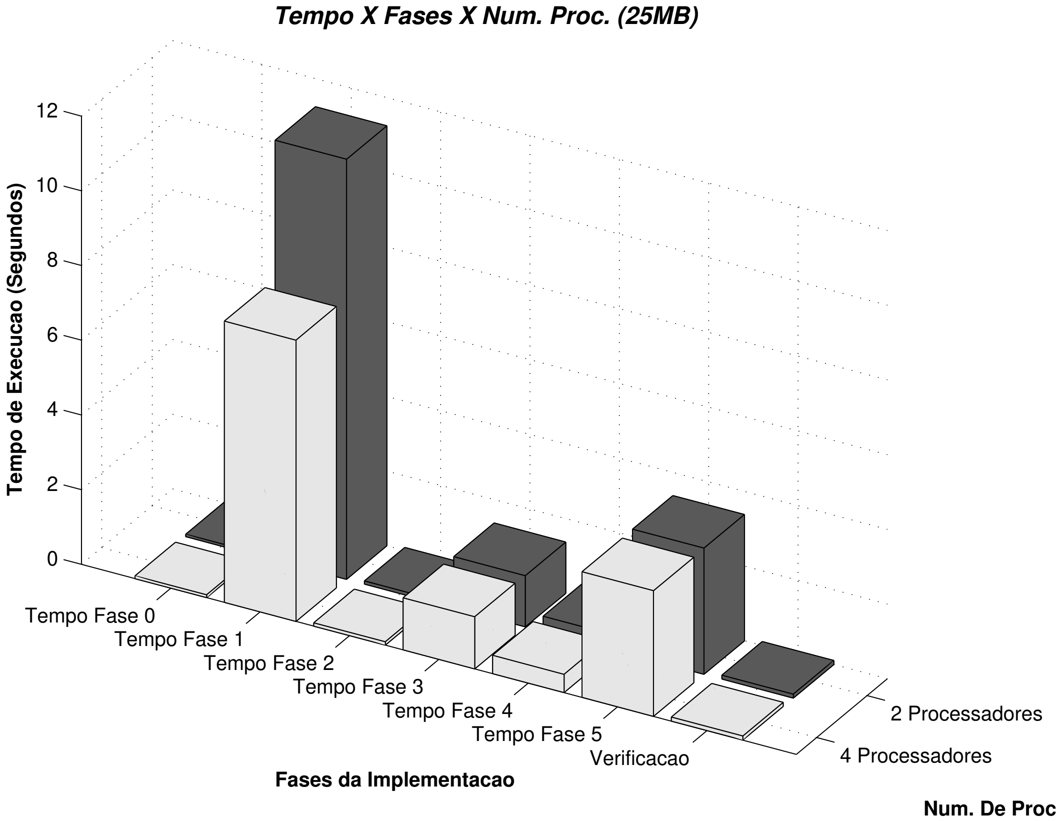

Figura 16: **Tempo de execução de cada uma das fases do algoritmo variando-se o número de processadores e utilizando uma massa de dado de entrada de 25MB.**

Fase	Num. Processadores	
	2	*4*
Tempo Fase 0	0,065771	0,075929
Tempo Fase 1	11,244844	7,528201
Tempo Fase 2	0,085866	0,089953
Tempo Fase 3	1,372957	1,393482
Tempo Fase 4	0,413931	0,490726
Tempo Fase 5	3,378792	3,361111
Tempo Verificação	0,107507	0,108772

Tabela 10: **Dados de tempo de execução de cada uma das fases do algoritmo variando-se o número de processadores e utilizando uma massa de dado de entrada de 25MB.**

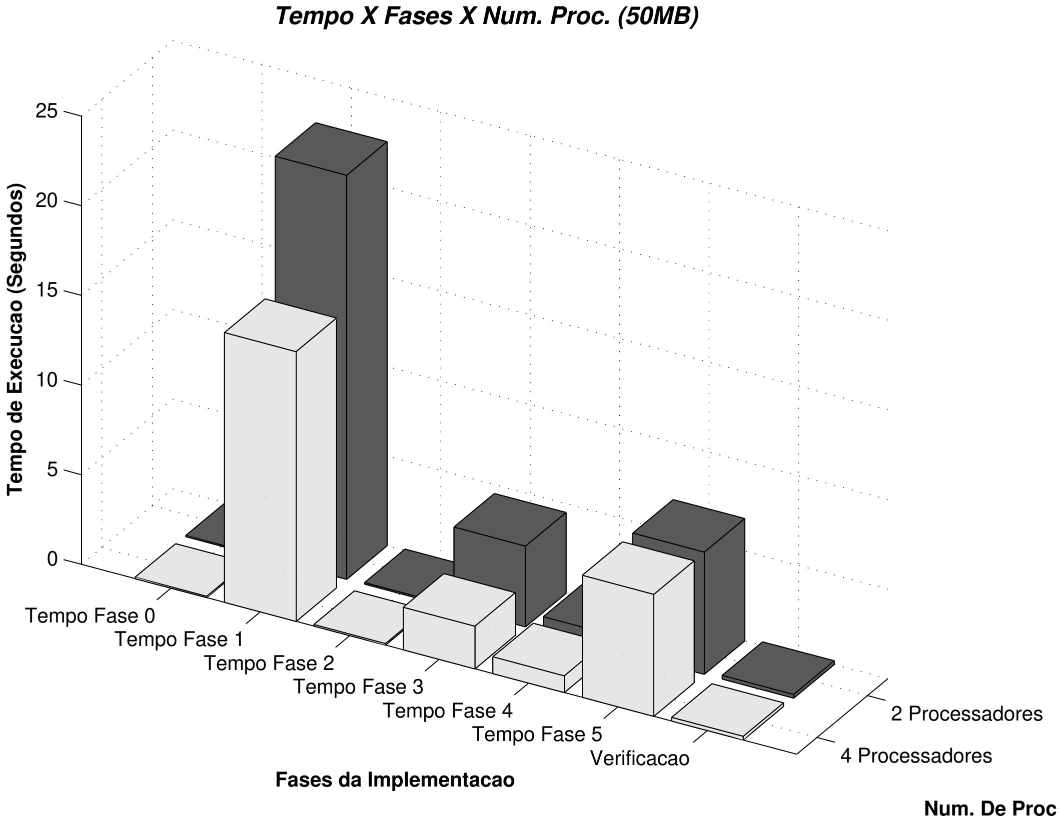

Figura 17: **Tempo de execução de cada uma das fases do algoritmo variando-se o número de processadores e utilizando uma massa de dado de entrada de 50MB.**

Fase	**Num. Processadores**	
	2	*4*
Tempo Fase 0	0,064312	0,076720
Tempo Fase 1	22,535747	15,045186
Tempo Fase 2	0,084841	0,089174
Tempo Fase 3	4,508423	2,377463
Tempo Fase 4	0,834132	0,935950
Tempo Fase 5	6,816228	6,800524
Tempo Verificacao	0,206540	0,206709

Tabela 11: **Dados de tempo de execução de cada uma das fases do algoritmo variando-se o número de processadores e utilizando uma massa de dado de entrada de 50MB.**

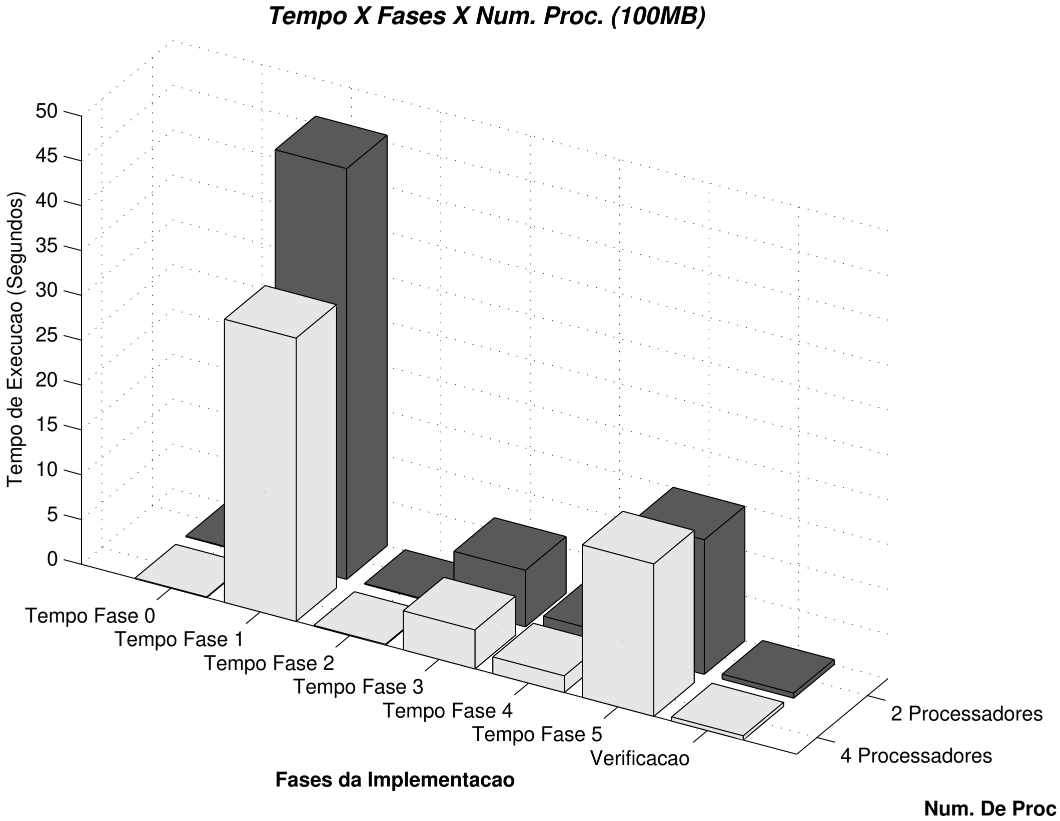

Figura 18: **Tempo de execução de cada uma das fases do algoritmo variando-se o número de processadores e utilizando uma massa de dado de entrada de 100MB.**

Fase	Num. Processadores	
	2	*4*
Tempo Fase 0	0,063719	0,073414
Tempo Fase 1	45,800661	31,581842
Tempo Fase 2	0,086368	0,090190
Tempo Fase 3	6,316336	4,349307
Tempo Fase 4	1,722161	1,860888
Tempo Fase 5	14,994674	16,996741
Tempo Verificacao	0,552540	0,465835

Tabela 12: **Dados de tempo de execução de cada uma das fases do algoritmo variando-se o número de processadores e utilizando uma massa de dado de entrada de 100MB.**

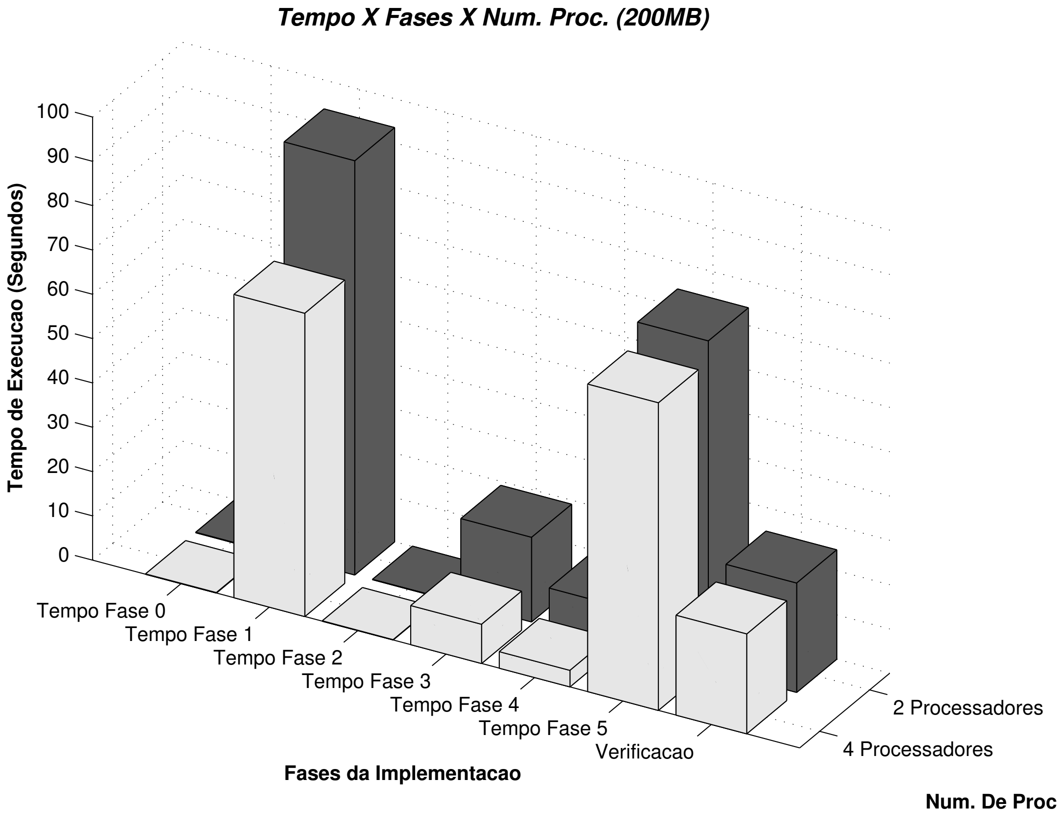

Figura 19: **Tempo de execução de cada uma das fases do algoritmo variando-se o número de processadores e utilizando uma massa de dado de entrada de 200MB.**

Fase	Num. Processadores	
	2	*4*
Tempo Fase 0	0,115624	0,101650
Tempo Fase 1	93,474848	68,434189
Tempo Fase 2	0,085280	0,095680
Tempo Fase 3	19,096014	8,824651
Tempo Fase 4	8,713290	3,737562
Tempo Fase 5	74,008666	69,390764
Tempo Verificacao	24,677641	22,538833

Tabela 13: **Dados de tempo de execução de cada uma das fases do algoritmo variando-se o número de processadores e utilizando uma massa de dado de entrada de 200MB.**

Execução / *Passo*	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	*0,0003*	*0,0003*	*146,22*	*0,0003*	*0,0003*	*84,811*	*0,0003*	*0,0003*	*33,674*	*0,0003*	*0,0003*	*19,521*
2	*0,0010*	*0,0010*	*48,643*	*0,0010*	*0,0010*	*22,703*	*0,0010*	*0,0010*	*12,278*	*0,0010*	*0,0010*	*5,1220*
3	*0,0022*	*0,0018*	*0,8260*	*0,0022*	*0,0023*	*0,4143*	*0,0022*	*0,0025*	*0,2467*	*0,0022*	*0,0028*	*0,1057*
4	*0,0654*	*0,0487*	*0,0015*	*0,0129*	*0,0202*	*0,0015*	*0,0141*	*0,0224*	*0,0015*	*0,0155*	*0,0221*	*0,0015*
5	*0,0031*	*0,0040*	—	*0,0032*	*0,0040*	—	*0,0031*	*0,0041*	—	*0,0031*	*0,0039*	—
6	*0,0435*	*0,0459*	—	*0,0442*	*0,0455*	—	*0,0436*	*0,0464*	—	*0,0436*	*0,0458*	—
7	*69,208*	*57,751*	—	*35,321*	*26,401*	—	*17,194*	*12,689*	—	*8,8953*	*6,4000*	—
8	*24,266*	*10,682*	—	*10,478*	*5,1800*	—	*5,3408*	*2,3557*	—	*2,3496*	*1,1282*	—
9	*0,0430*	*0,0451*	—	*0,0437*	*0,0452*	—	*0,0425*	*0,0443*	—	*0,0432*	*0,0450*	—
10	*0,0001*	*0,0001*	—	*0,0001*	*0,0001*	—	*0,0001*	*0,0001*	—	*0,0001*	*0,0004*	—
11	*0,0421*	*0,0505*	—	*0,0426*	*0,0449*	—	*0,0423*	*0,0448*	—	*0,0426*	*0,0446*	—
12	*19,096*	*8,8247*	—	*6,3163*	*4,3493*	—	*4,5084*	*2,3775*	—	*1,3730*	*1,3935*	—
13	*8,7133*	*3,7376*	—	*1,7222*	*1,8609*	—	*0,8341*	*0,9360*	—	*0,4139*	*0,4907*	—
14	*39,813*	*36,593*	—	*12,570*	*12,861*	—	*5,8968*	*5,8820*	—	*2,9082*	*2,9224*	—
15	*34,195*	*32,797*	—	*2,4240*	*4,1350*	—	*0,9194*	*0,9185*	—	*0,4706*	*0,4387*	—
16	*24,677*	*22,538*	—	*0,5525*	*0,4658*	—	*0,2065*	*0,2067*	—	*0,1075*	*0,1088*	—
Total(1-4)	*121,36*	*81,092*	—	*53,925*	*37,882*	—	*27,963*	*18,447*	—	*13,118*	*9,5023*	—
Total	*220,488*	*173,34*	*195,692*	*69,565*	*55,457*	*107,93*	*35,075*	*25,548*	*46,200*	*16,684*	*13,059*	*24,750*

Tabela 14: **Execuções realizadas da implementação.**

6.3.4 Execuções da Implementação

Conforme mostrado nos experimentos e também na Seção 6.2.4, diversas execuções da implementação foram realizadas. A Tabela 14 lista todos os dados de tempos coletados nestas execuções.

Note que as execuções 3, 6, 9 e 12 correspondem ao Quicksort Seqüencial e, portanto, não possuem as fases presentes no PSRS.

Foram incluídas também o tempo total de execução das Fases 1-4 do algoritmo PSRS e também o tempo todal de execução da implementação, que incluem os passos de enviar as listas para o *master* e verificar a corretude do resultado.

6.3.5 Observações sobre a Monitoração do Comportamento das Máquinas

Durante todas as execuções, todas as máquinas foram cuidadosamente monitoradas. Algumas observações foram retiradas desta monitoração:

- Praticamente toda a memória dos *slaves* foi utilizada, porém pouco acesso à memória virtual (*swap*) foi realizada por estes *hosts*. Em alguns casos nos testes com massa de dados de 200MB percebeu-se um pouco de acesso, mas nada muito substancial. Ao tentar realizar testes com massas maiores (400MB), percebeu-se a inviabilidade destas execuções devido à pouca memória dos *slaves* e, principalmente, do *master*, que precisa armazenar toda a lista de números para realizar a verificação da ordenação.
- No lado do *master*, este fez um uso estensivo do *swap* apenas ao receber todas as listas dos *slaves* na Fase 5. Porém não foi por muito tempo, mas este fato pode

ser observado visivelmente nos gráficos das Figuras 8, 9 e 13, onde as indicações de utilização do arquivo de 200MB mostram um aumento substancial em realção aos outros tamanhos.

- Percebeu-se que a máquina *carpa*, utilizada como *slave* 4 apresentava certos problemas de leitura do arquivo de massa de dados, visto que esta máquina era sempre a última a terminar com uma certa distância das outras. Procurou-se utilizar as outras máquinas disponíveis (*sena, eufrates, colorado e mississipi*) e todas apresentaram o mesmo problema, sendo que a carpa parecia ser a menos problemática. Este fato deve ter influenciado um pouco nos resultados dos testes realizados com 4 processadores de modo que se outra máquina estivesse disponível, possivelmente o tempo de execução dos testes utilizando 4 processadores poderiam ter sido um pouco menores.

7 Conclusão

Este trabalho tratou a respeito da ordenação utilizando algoritmos paralelos. Uma introdução aos algoritmos paralelos foi feita incluindo diversos conceitos relacionados ao trabalho.

Além disso, foi realizada o estudo e a implementação de dois algoritmos paralelos para o problema de ordenação, o *Parallel Sorting by Regular Sampling - Ordenação Paralela Utilizando Amostras Regulares* e o Quicksort Paralelo Tradicional. Sendo que apenas o primeiro foi realmente detalhado, experimentado e analisado, por estar mais diretamente relacionado ao assunto do trabalho.

Através deste trabalho foi possível não só entender melhor o assunto de algoritmo paralelos como também permitiu o aprendizado de implementação de um algoritmo deste tipo utilizando máquinas MIMD do tipo rede de estações de trabalho.

8 Sobre o Documento

Este documento foi feito completamente utilizando o sistema LaTeX[6][7]. Os gráficos foram gerados utilizando o MatLab e o GNUPlot. As implementações, com o GCC e *Bash Shell Script*.

Uma versão on-line da mesma, incluindo os códigos-fonte em C e em Bash, todos os dados gerados (inclusive os dados que não puderam ser colocado neste documento), e os gráficos, pode ser encontrada em:

`http://www.dcc.ufmg.br/~horacio/PaaTP4`

Referências

[1] QUINN, M.J. *Parallel Computing: Theory and Pratice*. McGraw-Hill, 1994.

[2] MANBER, Udi. *Introduction to Algorithms: A Creative Approach*. Editora Addison-Wesley, 1989.

[3] HOROWITZ, Ellis, SAHNI, Sartaj. *Fundamentals Of Computer Algorithms*. Editora Computer Science, 1978.

[4] ZIVIANI, Nivio. *Projeto de Algoritmos: Com implementações em Pascal e C*. Pioneira Thomson Learning, 2 edição, 2004.

[5] CROCHEMORE, Maxime. RYTTER, Wojciech *Text Algorithms*. Oxford, 1994.

[6] GOOSSENS, Michel. MITTELBACH, Frank. SAMARIN, Alexander. *The LaTeX Companion*. Editora Addison-Wesley, 1993.

[7] LAMPORT, Leslie. *The LaTeX: A Document Preparation System*. Editora Addison-Wesley, 1986.

[8] *CiteSeer: Scientific Literature Digital Library*. http://citeseer.nj.nec.com/cs, acessado em abril de 2004.

A Códigos Fonte

A.1 Implementação

Arquivo makefile.

```
# Sistema Unix, compilador gnu
CC = /usr/bin/gcc

LIBS = -lm -lpthread

# Diretorio raiz
T = .

#Diretorio dos arquivos .h
INCLUDE = $(T)/

# Opcoes de compilacao
OPTIONST = -c -g -O3 -Wall -I $(INCLUDE)

# Insira aqui o diretorio dos arquivos objetos sem a barra no final
obj_T = $(T)

# Nome do arquivo executavel
EXE=$(T)/PaaTP4.exe

# OBJs
OBJS=Main.o PSRSMasterMain.o PSRSSlaveMain.o QSPMain.o Sockets.o Utils.o Arquivo.o Slaves.o \
  ListaNumeros.o Ordenacao.o ListaSamples.o ListaPivots.o Timer.o

.c.o:
$(CC) $(OPTIONST) $<

all:  $(OBJS)
$(CC)  -o $(EXE)     $(obj_T)/*.o $(LIBS)
rm -rf *~

run:
$(EXE)

clean:
rm -rf $(obj_T)/*.o
rm -rf $(EXE)
rm -rf *~
```

Arquivo Module.h.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                  *
 * --------- --------                                  *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                            *
 *                                                     *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>          *
 *            http://www.dcc.ufmg.br/~horacio          *
 *                                                     *
 *    Principal Referência: Quinn (1994 - 10.5.3)      *
 *                                                     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

/***| Bibliotecas |***/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <sys/types.h>
```

```
#include <pthread.h>
#include "Sockets.h"
#include "Timer.h"

/***| Constantes |***/

#ifndef _QSP_MODULE_H_
#define _QSP_MODULE_H_ "@(#)Module.h $Revisao 0.1"

#define _IS_UNIX_
#define _IS_CLIENT_
#define _TIMEVAL_

#define DEBUG                          50

#define FALSE                          0
#define TRUE   (signed int)        ~FALSE

#define OK                         FALSE
#define ERROR                       TRUE

#define PROGRAM_NAME "QuickSort Paralelo"
#define PROGRAM_VERSION            "0.1"

#define MAX_IP                        15
#define MAX_HOSTS                     20
#define MAX_RAND          LONG_MAX-1000UL

//#define NUM_TAM                  530
#define NUM_TAM                53000000
//#define NUM_TAM                  23000000
#define SAM_TAM     MAX_HOSTS*MAX_HOSTS+1
#define PIV_TAM     MAX_HOSTS*MAX_HOSTS+1

#define ENVIO_ASCII                    0

/***| Declaracao das Estruturas |***/
// Lista de Numeros
typedef struct {
  int tamLista;                    // Tamanho da Lista
  unsigned long int *numeros;      // Lista de Numeros
} TLista;                          // Lista de Numeros

// Host Slave
typedef struct {
  int  id;                         // Identificacao do Host Slave
  char IP[MAX_IP];                 // IP do Host Slave
  int  porta;                      // Porta de Conexao do Host Slave
  int  socketFD;                   // Socket do Host Slave
  pthread_t thread;                // Thread do Host Slave
  char arqEntrada[100];            // Arquivo de entrada do Host Slave
  TLista numLista;                 // Lista de Numeros para o Slave
  TLista samLista;                 // Lista de Samples do Host Slave
} HostSlave;                       // Host Slave

/***| Variaveis Globais |***/
HostSlave hostsSlaves[MAX_HOSTS]; // Lista de Hosts Slaves
int       numHostsSlaves;         // Numero de Hosts Slaves

TLista numLista;                   // Lista de Numeros do Processo Atual
TLista samLista;                   // Lista de Samples do Processo Atual
TLista pivLista;                   // Lista de Pivots do Processo Atual

int slaveID;                       // Identificacao do Slave Atual

/***| Declaracao das Funcoes |***/

// --> PSRSSlaveMain.c
int  slaveMain(int portaTCP);
void *funcaoFilho(void *voidargs);
int  respSolicitacao(char *solicitacao, char *resposta, int socket);
```

```
// --> PSRSMasterMain.c
int  masterMain(char *arqClientes, char *arqEntrada);
void geraExemplo();
// --> QSPMain.c
int  qspMain(int portaTCP);
void executarQSP();
void *qspThread(void *voidargs);
void *funcaoFilhoQsp(void *voidargs);
int  respSolicitacaoQsp(char *solicitacao, char *resposta, int socket);
void retornaLista();
// --> QSSMain.c
int qssMain(char *nomeArq);
// --> Slaves.c
int  conectaSlaves();
int  desconectaSlaves();
void *cleanUpThread(void *voidargs);
int  lerSlavesArq(char *nomeArqEntrada);
void *enviarSlavesThread(void *voidargs);
void *enviarIDsThread(void *voidargs);
int  lerSlavesConexao(int socket);
// --> Arquivo.c
FILE *abreArquivo(char *nomeArquivo, char *modo);
int  lerListaArquivo(TLista *lista, char *arqEntrada);
int  salvaListaArquivo(TLista *lista, char *arqSaida);
void *enviarListaArqThread(void *voidargs);
// --> ListaNumeros.c
void novaListaNum(TLista *lista);
void addNumero(unsigned long int numero, TLista *lista);
void addNumeroOrdenado(unsigned long int numero, TLista *lista);
void addNumerosAleatorios(int quantidade, TLista *lista);
void printLista(TLista *lista);
void *enviarListaThread(void *voidargs);
void *execOrdenaListaThread(void *voidargs);
void *execTrocaListaThread(void *voidargs);
void *execMergeListasThread(void *voidargs);
void *getListasTrocadasThread(void *voidargs);
void joinListas();
int  lerListaConexao(int socket);
void *trocaListaThread(void *voidargs);
int  receberTrocaLista(int socket, int procID);
int  enviarListaTrocada(int socket);
void gerarListaTrocada();
// -> ListaSamples.c
void novaListaSam(TLista *lista);
void *getSamplesThread(void *voidargs);
void joinSamples();
void gerarSamples();
int  enviarSamples(int socket);
// --> ListaPivots.c
void novaListaPiv(TLista *lista);
void gerarPivots();
void *enviarPivotsThread(void *voidargs);
int  receberPivots(int socket);
// --> Ordenacao.c
void quickSort(TLista *lista);
void ordena(int esq, int dir, unsigned long int *A);
void particao(int esq, int dir, int *i, int *j, unsigned long int *A);
void verificaOrdenacao(TLista *lista);
void mergeListas(TLista *lista1, TLista *lista2);
// --> Utils.c
int criaThreads(void *funcThread);
int joinThreads();
void printAjuda();
void chomp(char *string);
void iniciaMyRand();
int myRand(int Max);
int isNumero(char *string);
int timevalSubtract(struct timeval *result, struct timeval *x,struct timeval *y);
void printTime();
#ifdef _IS_UNIX_
int  stricmp(char *str1, char *str2);
#endif
```

```
#endif
```

Arquivo Main.c.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                  *
 * --------- --------                                  *
 *    Ordenacao de Conjuntos  utilizando  Programacao  *
 *    Paralela e QuickSort.                            *
 *                                                     *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>          *
 *            http://www.dcc.ufmg.br/~horacio          *
 *                                                     *
 *    Principal Referência: Quinn (1994 - 10.5.3)      *
 *                                                     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                  *
 * --------- --------                                  *
 *    Ordenacao de Conjuntos  utilizando  Programacao  *
 *    Paralela e QuickSort.                            *
 *                                                     *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>          *
 *            http://www.dcc.ufmg.br/~horacio          *
 *                                                     *
 *    Principal Referência: Quinn (1994 - 10.5.3)      *
 *                                                     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao Main do QuickSortParalelo.                   *
 *   Verifica o parâmetro de entrada para ver qual mo- *
 *   do deve ser  executado.                           *
 *                                                     *
 *    @param argc Numero de parâmetros passados.       *
 *    @param argv Parâmetros passados pela execucao.   *
 *    @return Execucao com sucesso ou nao.             *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int main(int argc, char *argv[]) {

  if (DEBUG>2) printf("[main] Iniciando PaaTP4: QuickSort Paralelo .. \n");

  // Verifica os Parâmetros
  if (argc<3) {
    printAjuda();
    return 1;
  }

  // Inicia o Master ou o Slave
  if      (!stricmp(argv[1], "-s")) masterMain(argv[2], argv[3]);
  else if (!stricmp(argv[1], "-c")) {
    int porta = 65501;
    sscanf(argv[2], "%d", &porta);
    slaveMain(porta);
  }
  else if (!stricmp(argv[1], "-t")) {
    int porta = 65501;
    sscanf(argv[2], "%d", &porta);
    qspMain(porta);
  }
  else if (!stricmp(argv[1], "-q")) {
    qssMain(argv[2]);
  }
```

```
  else if (!stricmp(argv[1], "-g")) {
    unsigned long int tamanho = 0;
    sscanf(argv[2], "%ld", &tamanho);
    addNumerosAleatorios(tamanho, &numLista);
    salvaListaArquivo(&numLista, argv[3]);
  }
  // Parâmetros Errados
  else    printAjuda();

  return 0;
}
```

Arquivo PSRSMasterMain.c.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                *
 * --------- --------                                *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                          *
 *                                                   *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>        *
 *            http://www.dcc.ufmg.br/~horacio        *
 *                                                   *
 *    Principal Referência: Quinn (1994 - 10.5.3)    *
 *                                                   *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao Main do Master do QuickSort Paralelo.       *
 *   Executa cada um dos Passos do Algoritmo encontra- *
 *   do na referencia Quinn (1994 - 10.5.3)           *
 *                                                    *
 *    @param arqSlaves Nome do Arquivo de Slaves.     *
 *    @return Execucao com sucesso ou nao.            *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
int masterMain(char *arqSlaves, char *arqEntrada) {
  if (DEBUG>3) printf("[serverMain] Iniciando Master ..\n");

  slaveID=-1;

  // Calculo do Tempo de Execucao
  startTimer();

  // Cria as Listas Necessarias
  if (DEBUG>3) printf("[serverMain]     Criando Lista de Numeros .. \n");
  //novaListaNum(&numLista);
  //novaListaSam(&samLista);
  //novaListaPiv(&pivLista);
  printTimer();

  // Le ou Gera os Numeros Necessarios
  if (DEBUG>3) printf("[serverMain]     Lendo numeros .. \n");


  novaListaNum(&numLista);
  geraExemplo();
  //lerListaArquivo(&numLista, arqEntrada);
  //  lerListaArquivo(&numLista, "/var/tmp/paa/OutraSaidaTeste.bin");
  //  lerListaArquivo(&numLista, "/var/tmp/paa/t2.bin");
  //  lerListaArquivo(&numLista, "/var/tmp/paa/dados200MB.bin");
  //  addNumerosAleatorios(3276800, &numLista);
  //  salvaListaArquivo(&numLista, "/var/tmp/paa/OutraSaidaTeste.bin");
  //  exit(0);
  //  printf("Numeros: "); printLista(&numLista);
  //  quickSort(&numLista);
  //  printf("Numeros: "); printLista(&numLista);
  printTimer();
```

```
// Le o arquivo de Slaves
if (DEBUG>3) printf("[serverMain]     Lendo Slaves de %s .. \n", arqSlaves);
if ( !lerSlavesArq(arqSlaves) ) exit(1);
printTimer();
// Conecta aos Slaves
if (DEBUG>3) printf("[serverMain]     Conectando a Slaves .. \n");
if ( !conectaSlaves() ) exit(1);
printTimer();
// Diz pros Slaves quem eles sao
if (DEBUG>3) printf("[serverMain]     Identificando Slaves .. \n");
criaThreads(enviarIDsThread);
joinThreads();
printTimer();
// Envia a lista de Slaves para cada um dos Slaves
if (DEBUG>3) printf("[serverMain]     Enviando Lista de Slaves .. \n");
criaThreads(enviarSlavesThread);
joinThreads();
printTimer();
// Envia a Lista de Numeros ou manda ler de um Arquivo. Os Slaves vao ordenar.
if (DEBUG>3) printf("[serverMain]     Enviando Lista de Numeros .. \n");
criaThreads(enviarListaThread);
//criaThreads(enviarListaArqThread);
joinThreads();
free(numLista.numeros);
printTimer();
// Solicita que os Slaves facam a Ordenacao de suas Listas
if (DEBUG>3) printf("[serverMain]     Solicitando a Ordenacao das Listas .. \n");
criaThreads(execOrdenaListaThread);
joinThreads();
printTimer();
// Requisita os Samples
if (DEBUG>3) printf("[serverMain]     Requisitando Samples .. \n");
criaThreads(getSamplesThread);
joinThreads();
joinSamples();
quickSort(&samLista);
printTimer();
// Gera os Pivots
if (DEBUG>3) printf("[serverMain]     Gerando Pivots .. \n");
gerarPivots();
printTimer();
// Envia os Pivots para os Slaves
if (DEBUG>3) printf("[serverMain]     Enviando Lista de Pivots .. \n");
criaThreads(enviarPivotsThread);
joinThreads();
printTimer();
// Solicita que os Slaves facam a Troca das Listas entre Si
if (DEBUG>3) printf("[serverMain]     Solicitando Troca de Listas .. \n");
criaThreads(execTrocaListaThread);
joinThreads();
printTimer();
// Solicita a Execucao do Merge das Listas
if (DEBUG>3) printf("[serverMain]     Solicitando Merge das Listas .. \n");
criaThreads(execMergeListasThread);
joinThreads();
printTimer();
// Requisita as Listas dos Slaves
if (DEBUG>3) printf("[serverMain]     Requisitando Listas Trocadas .. \n");
criaThreads(getListasTrocadasThread);
joinThreads();
printTimer();
// Unifica as Listas em uma so
if (DEBUG>3) printf("[serverMain]     Unindo as Listas Trocadas .. \n");
joinListas();
printTimer();
// Verifica se a ordenacao final esta correta
if (DEBUG>3) printf("[serverMain]     Verificando Ordenacao da Lista .. \n");
verificaOrdenacao(&numLista);
printTimer();
// Mostra a Lista Final
if (DEBUG>3) {
```

```
    printf("[serverMain]     LISTA FINAL:\n\n");
    printLista(&numLista);
    printf("\n");
  }
  // Desconecta dos Slaves
  if (DEBUG>3) printf("[serverMain]     Limpando Memoria dos Slaves .. \n");
  criaThreads(cleanUpThread);
  joinThreads();
  printTimer();
  // Desconecta dos Slaves
  if (DEBUG>3) printf("[serverMain]     Desconectando dos Slaves .. \n");
  desconectaSlaves();
  printTimer();

  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao geraExemplo                                 *
 *    Gera uma Lista de Numeros de Exemplo.           *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void geraExemplo() {

  addNumero(15, &numLista);
  addNumero(46, &numLista);
  addNumero(48, &numLista);
  addNumero(93, &numLista);
  addNumero(39, &numLista);
  addNumero(6,  &numLista);
  addNumero(72, &numLista);
  addNumero(91, &numLista);
  addNumero(14, &numLista);
  addNumero(36, &numLista);
  addNumero(69, &numLista);
  addNumero(40, &numLista);
  addNumero(89, &numLista);
  addNumero(61, &numLista);
  addNumero(97, &numLista);
  addNumero(12, &numLista);
  addNumero(21, &numLista);
  addNumero(54, &numLista);
  addNumero(53, &numLista);
  addNumero(97, &numLista);
  addNumero(84, &numLista);
  addNumero(58, &numLista);
  addNumero(32, &numLista);
  addNumero(27, &numLista);
  addNumero(33, &numLista);
  addNumero(72, &numLista);
  addNumero(20, &numLista);
}
```

Arquivo PSRSSlaveMain.c.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                 *
 * --------- --------                                 *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                           *
 *                                                    *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>         *
 *            http://www.dcc.ufmg.br/~horacio         *
 *                                                    *
 *    Principal Referência: Quinn (1994 - 10.5.3)     *
 *                                                    *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"
```

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao Main do Slave do QuickSort Paralelo.         *
 *   Abre uma porta e fica respondendo  a  requisicoes *
 *   do Master.                                        *
 *                                                     *
 *    @param portaTCP Porta que sera aberta.           *
 *    @return         Execucao com sucesso ou nao.     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int slaveMain(int portaTCP) {
  if (DEBUG>3) printf("[slaveMain] Iniciando Slave ..\n");
  if (DEBUG>3) printf("[slaveMain]   Gerenciando Conexoes na porta %d ..\n", portaTCP);
  // Seta o ID do Slave atual para -1
  slaveID = -1;
  // Espera por Conexoes. Quando vier, a FuncaoFilho e executada.
  gerenciaConexoes(portaTCP,0);
  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao funcaoFilho                                  *
 *   Funcao que e executada sempre que uma conexao no- *
 *   va eh recebida.                                   *
 *                                                     *
 *    @param voidargs Ponteiro que contem o socket.    *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void *funcaoFilho(void *voidargs) {
  char solicitacao[MAXMSG];       // String para as Solicitacoes
  char resposta[MAXMSG];          // String para as Respostas
  int continua = 1;               // Se continua na Funcao ou Sai
  struct strSchArgs *schArgs;     // Estrutura de Arg. do Thread
  int socket;                     // Ponteiro para o Socket

  // Seta a Estrutura enviada pelo Thread
  schArgs = (struct strSchArgs *)voidargs;
  // Seta o Socket
  socket = schArgs->conexao;

  if (DEBUG>3) printf("[%d][funcaoFilho]     Conexão recebida !!\n", slaveID);

  fflush(NULL);

  // Conexao Recebida. Vamos dar as Boas Vindas.
  sprintf(resposta, "001 %s %s (QuickSort Paralelo - User Default Installation)\r\n", \
      PROGRAM_NAME, PROGRAM_VERSION);
  socketPuts(socket, resposta);

  // Fica escutando os Comandos
  while (continua) {
    fflush(NULL);
    // Recebe a Solicitacao
    if (socketGets(socket, solicitacao) < 0) {
      close(socket);
      continua = 0;
    }
    else {
      // Responde a Solicitaca
      continua = respSolicitacao(solicitacao, resposta, socket);
      // Envia a Resposta
      socketPuts(socket, resposta);
    }
  }
  // Fecha o Socket
  close(socket);
  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao respSolicitacao                              *
 *   Recebe uma Solicitacao, executa as operacoes  ne- *
 *   cessarias e envia uma Resposta.                   *
 *                                                     *
```

```
 *    @param solicitacao String com a Solicitacao.     *
 *    @param resposta    String com a Resposta.        *
 *    @param socket      Ponteiro para o Socket.       *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int respSolicitacao(char *solicitacao, char *resposta, int socket) {
  strcpy(resposta, "");

  // Processa Comandos Remotos
  if (!strcmp(solicitacao, "")) return 1;

  // O Master esta falando quem eu, Slave, sou. Enviando o ID
  else if (!strncmp(solicitacao, "YOU ARE ", 8)) {
    sscanf(solicitacao, "YOU ARE %d", &slaveID);
    sprintf(resposta, "005 - Ok. Agora eu tenha o ID %d.\r\n", slaveID);
    if (DEBUG>4) printf("[%d][respSolicitacao] YOU ARE Recebido. ID setado para %d !!\n", \
      slaveID, slaveID);
    return 1;
  }
  // O Master quer colocar a lista de Slaves
  else if (!strncasecmp(solicitacao, "PUT SLAVES", 10)) {
    if (DEBUG>4) printf("[%d][respSolicitacao] PUT SLAVES Recebido.\n", slaveID);
    lerSlavesConexao(socket);
    return 1;
  }
  // O Master que enviar a Lista de Numeros pela Rede
  else if (!strncasecmp(solicitacao, "PUT LIST", 7)) {
    if (DEBUG>4) printf("[%d][respSolicitacao] PUT LIST Recebido.\n", slaveID);
    lerListaConexao(socket);
    return 1;
  }
  // O Master estah pedindo para ler um arquivo com a Lista de Numeros
  else if (!strncasecmp(solicitacao, "READ ", 5)) {
    char arqEntrada[100];
    sscanf(solicitacao, "READ %s", arqEntrada);
    if (DEBUG>4) printf("[%d][respSolicitacao] READ Recebido. Lendo Arquivo %s ..\n", slaveID, arqEntrada);
    novaListaNum(&numLista);
    lerListaArquivo(&numLista, arqEntrada);
    // quickSort(&numLista);
    sprintf(resposta, "005 - Ok. O Arquivo %s foi lido e a lista criada com %d numeros.\r\n", \
      arqEntrada, numLista.tamLista);
    if (DEBUG>4) printf("[%d][respSolicitacao] O Arquivo %s foi lido e a lista criada com %d numeros.\r\n",\
     slaveID, arqEntrada, numLista.tamLista);
    return 1;
  }
  // O Master estah pedindo para ordenar a lista
  else if (!strncasecmp(solicitacao, "ORDER LIST", 5)) {
    if (DEBUG>4) printf("[%d][respSolicitacao] ORDER LIST Recebido. Ordenando a Lista ..\n", slaveID);
    quickSort(&numLista);
    strcpy(resposta, "005 - Ok. A Lista foi Ordenada.\r\n");
    if (DEBUG>4) printf("[%d][respSolicitacao] A Lista foi Ordenada.\r\n", slaveID);
    return 1;
  }
  // O Master estah Solicitando a Lista de Samples
  else if (!strncasecmp(solicitacao, "GET SAMPLES", 11)) {
    if (DEBUG>4) printf("[%d][respSolicitacao] GET SAMPLES Recebido. Enviando Samples ..\n", slaveID);
    enviarSamples(socket);
    return 1;
  }
  // O Master esta enviando a Lista de Pivots
  else if (!strncasecmp(solicitacao, "PUT PIVOTS", 10)) {
    if (DEBUG>4) printf("[%d][respSolicitacao] PUT PIVOTS Recebido. Recebendo Pivots ..\n", slaveID);
    receberPivots(socket);
    return 1;
  }
  // O Master esta pedindo que eu troque minha lista com os outros Slaves
  else if (!strncasecmp(solicitacao, "EXCH LIST", 9)) {
    if (DEBUG>4) printf("[%d][respSolicitacao] EXCH LIST Recebido. Trocando Listas ..\n", slaveID);
    criaThreads(trocaListaThread);
    joinThreads();
    strcpy(resposta, "044 - Ok. Listas Trocadas !!\r\n");
    return 1;
  }
```

```
  // Um Slave esta querendo enviar sua Lista de Numeros para mim
  else if (!strncasecmp(solicitacao, "PUT EXCH LIST ", 14)) {
    int procID=-1;
    sscanf(solicitacao, "PUT EXCH LIST %d", &procID);

    if (DEBUG>4) printf("[%d][respSolicitacao] PUT EXCH LIST Recebido. Recebendo Lista de Troca de %d ..\n",\
     slaveID, procID);

    receberTrocaLista(socket, procID);

    if (DEBUG>15) printf("[%d][receberTrocaLista] Cheguei aqui 2 ..\n", slaveID);

    return 1;
  }
  // O Master estah pedindo para ordenar a lista
  else if (!strncasecmp(solicitacao, "MERGE LISTS", 5)) {
    if (DEBUG>4) printf("[%d][respSolicitacao] MERGE LISTS Recebido. Mergindo as Listas ..\n", slaveID);
    gerarListaTrocada();
    strcpy(resposta, "005 - Ok. As Listas foram Mergida !!.\r\n");
    if (DEBUG>4) printf("[%d][respSolicitacao] A Lista foi Ordenada.\r\n", slaveID);
    return 1;
  }
  // O Master esta solicitando a Lista Trocada Final
  else if (!strncasecmp(solicitacao, "GET EXCH LIST", 10)) {
    if (DEBUG>4) printf("[%d][respSolicitacao] GET EXCH LIST Recebido. Enviando Lista Final ..\n", slaveID);
    enviarListaTrocada(socket);
    return 1;
  }
  // Responde ao Comando de Ajuda (HELP)
  if (!strcasecmp(solicitacao, "HELP")) {
    strcpy(resposta, "001 - Os seguintes comandos são reconhecidos:\r\n\tHELP\tQSP\tQUIT\r\n001 - \
      Comentários para horacio@dcc.ufmg.br\r\n");
  }
  // Responde ao Comando de Saida (QUIT). Desconecta
  else if (!strcasecmp(solicitacao, "QUIT")) {
    if (DEBUG>4) printf("[%d][respSolicitacao] QUIT Recebido. Desconectando !!\n", slaveID);
    strcpy(resposta, "001 - T+ !!\r\n");
    return 0;
  }
  // Responde ao Comando de Saida (QUIT). Desconecta
  else if (!strcasecmp(solicitacao, "CLEAN UP")) {
    //int id;
    if (DEBUG>22) printf("[%d][respSolicitacao] CLEAN UP Recebido. Limpando !!\n", slaveID);
    fflush(NULL);
    // Limpando Memoria
    free(numLista.numeros);
    free(samLista.numeros);
    free(pivLista.numeros);
    /*
    for (id=1; id<numHostsSlaves; id++) {
      free(hostsSlaves[id].numLista.numeros);
      free(hostsSlaves[id].samLista.numeros);
    }
    */
    strcpy(resposta, "001 - Tudo Limpo !!\r\n");
    return 1;
  }


  // Fecha Cliente
  else if (!strcasecmp(solicitacao, "KILL")) {
    strcpy(resposta, "001 - Matando Cliente !! T+ !!\r\n");
    return 0;
  }
  // Comando nao Reconhecido
  else strcpy(resposta, "003 - Comando não Reconhecido !!\r\n");
  return 1;
}
```

Arquivo QSPMain.c.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                  *
 * --------- --------                                  *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort. O Quicksort Paralelo Tra- *
 *    dicional.                                        *
 *                                                     *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>          *
 *            http://www.dcc.ufmg.br/~horacio          *
 *                                                     *
 *    Principal Referência: Quinn (1994 - 10.5.1)      *
 *                                                     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao Main do QuickSort Paralelo.                  *
 *   Verifica se eh o Master ou algum Slave. Se for  o *
 *   Master, comeca a executar. Se for o Slave, abre a *
 *   porta e fica esperando.                           *
 *                                                     *
 *    @param portaTCP Porta que sera aberta.           *
 *    @return         Execucao com sucesso ou nao.     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int qspMain(int portaTCP) {
  int i;                          // Contador Geral

  if (DEBUG>3) printf("[qspMain] Iniciando QuickSortParalelo ..\n");

  // A porta identifica se eh o Master ou algum Slave. Se a porta for
  // menor que zero, é o Master. Slave caso contrário.

  if (portaTCP < 0) {
    // EH o MASTER .. Vamos ler os numeros e os slaves de arquivos e continuar ..
    if (DEBUG>3) printf("[qspMain:Master]   Iniciando MASTER ..\n");
    // Calculo do Tempo de Execucao
#ifdef _TIMEVAL_
    struct timeval inicio, fim, tempoDec; // Tempos de Execucao
    gettimeofday(&inicio, NULL);
#endif
    // Seta o SlaveIS para negativo, para identificar que e o Master
    slaveID = -1;
    // Cria a Lista de Numeros
    if (DEBUG>3) printf("[qspMain:Master]     Criando Lista de Numeros .. \n");
    // Le os Numeros do Arquivo
    if (DEBUG>3) printf("[qspMain:Master]     Lendo numeros .. \n");
    novaListaNum(&numLista);
    lerListaArquivo(&numLista, "/var/tmp/paa/OutraSaidaTeste.bin");
    //geraExemplo();
    // Le o arquivo de Slaves
    if (DEBUG>3) printf("[qspMain:Master]     Lendo Slaves de HostsClientes .. \n");
    if ( !lerSlavesArq("HostsClientes") ) exit(1);
    // Executa o Codigo Igual a Todos os Processos
    executarQSP();
    // Unifica as Listas
    free(numLista.numeros);
    novaListaNum(&numLista);
    for (i=0; i<hostsSlaves[0].numLista.tamLista; i++)
      addNumero(hostsSlaves[0].numLista.numeros[i], &numLista);
    for (i=0; i<hostsSlaves[1].numLista.tamLista; i++)
      addNumero(hostsSlaves[1].numLista.numeros[i], &numLista);
    // Verifica se a ordenacao final esta correta
    if (DEBUG>3) printf("[qspMain:Master]     Verificando Ordenacao da Lista .. \n");
    verificaOrdenacao(&numLista);
    // Mostra a Lista Final
    if (DEBUG>3) {
      printf("[qspMain:Master]     LISTA FINAL:\n\n");
      printLista(&numLista);
      printf("\n");
```

```
    }
#ifdef _TIMEVAL_
    gettimeofday(&fim, NULL);
    timevalSubtract(&tempoDec, &fim, &inicio);
    if (DEBUG>=0) printf("[serverMain] Tempo Final: %d.%06d\n", (int) tempoDec.tv_sec, \
      (int) tempoDec.tv_usec);
#endif
  }
  else {
    // EH o SLAVE. Vamos abrir uma Conexao e ficar esperando ..
    if (DEBUG>3) printf("[qspMainSlave]   Iniciando SLAVE ..\n");
    if (DEBUG>3) printf("[qspMainSlave]   Gerenciando Conexoes na porta %d ..\n", portaTCP);
    // Seta o ID do Slave atual para 1
    slaveID = 1;
    // Espera por Conexoes. Quando vier, a FuncaoFilhoQsp e executada.
    gerenciaConexoes(portaTCP, 1);
  }

  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao executarQSP                                  *
 *   Funcao que executa o algoritmo do QuickSort para- *
 *   lelo Tradicional.  Executada  tanto  pelo  master *
 *   quanto pelos Slaves.                              *
 *                                                     *
 *    @param portaTCP Porta que sera aberta.           *
 *    @return         Execucao com sucesso ou nao.     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
void executarQSP() {
  int i, j, l;

  // Verifica se tem mais Slaves.
  if (numHostsSlaves==1) {
    // Nao tem Slaves. Ordena e Retorna !!
    quickSort(&numLista);
    return;
  }
  // Tem Slaves !! Particiona e envia para os responsaveis.
  if (DEBUG>3) printf("[%d][executarQSP]       Particionando a Lista ..\n", slaveID);
  particao(0, numLista.tamLista-1, &i, &j, numLista.numeros);
  // Se for Slave, seta o Slave[0]
  if (slaveID>0) {
    if (DEBUG>3) printf("[%d][executarQSP]       Adicionando hostsSlaves[0] ..\n", slaveID);
    hostsSlaves[0].id    =  0;
    strcpy(hostsSlaves[0].IP,"-1");
    hostsSlaves[0].porta = -1;
  }
  // Coloca a primeira parte em slavesHost[0]
  if (DEBUG>3) printf("[%d][executarQSP]       Preenchendo a primeira Lista - hostsSlaves[0] ..\n", slaveID);
  novaListaNum(&hostsSlaves[0].numLista);
  if (0<=j) for (l=0; l<=j; l++) addNumero(numLista.numeros[l], &hostsSlaves[0].numLista);
  if (j+2==i) addNumero(numLista.numeros[j+1], &hostsSlaves[0].numLista);
  if (DEBUG>3) printf("[%d][executarQSP]         Lista: ", slaveID);
  if (DEBUG>3) printLista(&hostsSlaves[0].numLista);
  // Coloca a segunda parte em slavesHost[1]
  if (DEBUG>3) printf("[%d][executarQSP] Preenchendo a segunda Lista - hostsSlaves[1] ..\n", slaveID);
  novaListaNum(&hostsSlaves[1].numLista);
  if (i<numLista.tamLista) for (l=i; l<numLista.tamLista; l++) addNumero(numLista.numeros[l], \
    &hostsSlaves[1].numLista);
  if (DEBUG>3) printf("[%d][executarQSP]         Lista: ", slaveID);
  if (DEBUG>3) printLista(&hostsSlaves[1].numLista);
  // Inicia os Threads !!
  if (DEBUG>3) printf("[%d][executarQSP]       Executando Threads ..\n", slaveID);
  pthread_create(&hostsSlaves[0].thread, NULL,qspThread,&hostsSlaves[0]);
  pthread_create(&hostsSlaves[1].thread, NULL,qspThread,&hostsSlaves[1]);
  // Faz um Join nos Threads ..
  if (DEBUG>3) printf("[%d][executarQSP]       Esperando pelo Thread 0 ..\n", slaveID);
  pthread_join(hostsSlaves[0].thread,NULL);
  if (DEBUG>3) printf("[%d][executarQSP]       Esperando pelo Thread 1 ..\n", slaveID);
  pthread_join(hostsSlaves[1].thread,NULL);
```

```
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao qspThread                                    *
 *   Thread que ou ordena localmente a lista de  nume- *
 *   os ou os manda para outro processo ordenar e fica *
 *   esperando a resposta.                             *
 *                                                     *
 *    @param portaTCP Porta que sera aberta.           *
 *    @return         Execucao com sucesso ou nao.     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void *qspThread(void *voidargs) {
  char resposta[MAXMSG];                          // String com a Resposta
  char solicitacao[MAXMSG];                       // String com a Solicitacao
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual
  int i=-1, j=-1;                                 // Contadores

  if (DEBUG>3) printf("[%d][%d][qspThread] Thread %d Criado.\n", slaveID, hostSlave->id, hostSlave->id);

  // Verifica se vai ordenar Localmente
  if (slaveID>=0 && hostSlave->id == 0) {
    // Sim ..
    if (DEBUG>3) printf("[%d][%d][qspThread] Ordenando Localmente !!.\n", slaveID, hostSlave->id);
    quickSort(&hostSlave->numLista);
    // Termina a execucao do Thread
    pthread_exit (0);
  }

  // Nao vao ordenar localmente. Se conecta em outro e manda as listas
  // --> CONECTA <--
  if (DEBUG>3) printf("[%d][%d][qspThread] Conectando .. \n", slaveID, hostSlave->id);
  if ((hostsSlaves[hostSlave->id].socketFD = conecta(hostsSlaves[hostSlave->id].IP, \
    hostsSlaves[hostSlave->id].porta)) <= 0)
    printf("[%d][qspMasterThread] Nao foi possivel Conectar a %s porta %d !!\n", slaveID, \
      hostsSlaves[hostSlave->id].IP, hostsSlaves[hostSlave->id].porta);

  // --> DIZ QUEM EH <--
  if (DEBUG>3) printf("[%d][%d][qspThread] Enviando ID.\n", slaveID, hostSlave->id);
  int temp = slaveID;
  if (temp<0) temp=8;
  // Diz quem o Slave eh
  sprintf(solicitacao, "YOU ARE %d%d\r\n", temp, hostSlave->id);
  socketPuts(hostSlave->socketFD, solicitacao);
  // Espera pela Resposta "005 - Ok."
  socketGets(hostSlave->socketFD, solicitacao);

  // --> ENVIA LISTA DE SLAVES <--
  if (DEBUG>3) printf("[%d][%d][qspMasterThread] Enviando Lista de Slaves .. \n", slaveID, hostSlave->id);
  // Diz que vai comecar a enviar os Slaves
  socketPuts(hostSlave->socketFD, "PUT SLAVES\r\n");
  // Espera pela Resposta "022 - OK."
  if (socketGets(hostSlave->socketFD, solicitacao) >= 0) {
    // Comeca a enviar a Lista de Slaves
    if (slaveID<0) {
      if (hostSlave->id == 0) { i=2; j=((int)((numHostsSlaves-2)/2))+1; }
      else if (hostSlave->id == 1) { i=((int)((numHostsSlaves-2)/2))+2 ; j=numHostsSlaves-1; }
      else printf("\nEita !! !! !!\n");
    }
    else {i=2; j=numHostsSlaves-1;}
    if (DEBUG>3) printf("[%d][%d][qspThread] Enviando Slaves de %d ate %d\n", slaveID, i, j, hostSlave->id);
    for (; i<=j; i++) {
      sprintf(solicitacao, "SLAVE %d %s %d\r\n", hostsSlaves[i].id, hostsSlaves[i].IP, hostsSlaves[i].porta);
      socketPuts(hostSlave->socketFD, solicitacao);
    }
    // Diz que a Lista Terminou
    socketPuts(hostSlave->socketFD, "END SLAVES\r\n");
    socketGets(hostSlave->socketFD, solicitacao);
  }

  // --> ENVIA LISTA DE NUMEROS <--
  if (DEBUG>3) printf("[%d][%d][qspThread] Enviando Lista de Numeros .. \n", slaveID, hostSlave->id);
  // Diz que vai colocar a lista
```

```
  socketPuts(hostSlave->socketFD, "PUT LIST\r\n");
  // Espera pela Resposta "012 - OK."
  if (socketGets(hostSlave->socketFD, solicitacao) >= 0) {
    // Envia os numeros
    for (i=0; i<hostSlave->numLista.tamLista; i++) {
      sprintf(solicitacao, "%ld\r\n", hostSlave->numLista.numeros[i]);
      socketPuts(hostSlave->socketFD, solicitacao);
    }
    // Diz que acabou os numeros
    socketPuts(hostSlave->socketFD, "END LIST\r\n");
    // Espera pela resposta "014 - OK."
    socketGets(hostSlave->socketFD, solicitacao);
  }

  // Libera Memoria
  if (DEBUG>3) printf("[%d][%d][qspThread] Liberando Memoria .. \n", slaveID, hostSlave->id);
  free(hostSlave->numLista.numeros);

  // --> PEGA O RETORNO DA LISTA <--
  if (DEBUG>3) printf("[%d][%d][qspThread] Pegando o Retorno da Lista .. \n", slaveID, hostSlave->id);
  unsigned long int numeroAtual;  // Numero atualmente lido
  novaListaNum(&hostSlave->numLista);
  // Solicita o envio da Lista
  socketPuts(hostSlave->socketFD, "GET LIST\r\n");
  // Espera pela Resposta "052 - OK."
  socketGets(hostSlave->socketFD, resposta);
  if (DEBUG>3) printf("[%d][%d][qspThread]       Resposta Recebida .. \n", slaveID, hostSlave->id);
  // Comeca a receber os numeros
  while (socketGets(hostSlave->socketFD, resposta) >= 0) {
    if (strncasecmp(resposta, "052 - Ok. Fim da Lista de Numeros", 30)) {
      sscanf(resposta, "%ld", &numeroAtual);
      addNumero(numeroAtual, &hostSlave->numLista);
      if (DEBUG>5) printf("[%d][%d][qspThread] Recebendo numero %ld ..\n", slaveID, \
        hostSlave->id, numeroAtual);
    }
    else break;
  }

  // Desconecta
  if (DEBUG>3) printf("[%d][%d][qspThread] Desconectando .. \n", slaveID, hostSlave->id);
  desconecta(hostSlave->socketFD);

  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao funcaoFilhoQsp                              *
 *   Funcao que e executada sempre que uma conexao no- *
 *   va eh recebida.                                  *
 *                                                    *
 *    @param voidargs Ponteiro que contem o socket.    *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void *funcaoFilhoQsp(void *voidargs) {
  char solicitacao[MAXMSG];       // String para as Solicitacoes
  char resposta[MAXMSG];          // String para as Respostas
  int continua = 1;               // Se continua na Funcao ou Sai
  struct strSchArgs *schArgs;     // Estrutura de Arg. do Thread
  int socket;                     // Ponteiro para o Socket

  // Seta a Estrutura enviada pelo Thread
  schArgs = (struct strSchArgs *)voidargs;
  // Seta o Socket
  socket = schArgs->conexao;
  if (DEBUG>3) printf("[%d][funcaoFilho]     Conexão recebida !!\n", slaveID);
  fflush(NULL);
  // Conexao Recebida. Vamos dar as Boas Vindas.
  sprintf(resposta, "001 %s %s (QuickSort Paralelo - User Default Installation)\r\n",\
     PROGRAM_NAME, PROGRAM_VERSION);
  socketPuts(socket, resposta);
  // Fica escutando os Comandos
  while (continua) {
```

```
    fflush(NULL);
    // Recebe a Solicitacao
    if (socketGets(socket, solicitacao) < 0) {
      close(socket);
      continua = 0;
    }
    else {
      // Responde a Solicitaca
      continua = respSolicitacaoQsp(solicitacao, resposta, socket);
      // Envia a Resposta
      socketPuts(socket, resposta);
    }
  }
  // Fecha o Socket
  close(socket);
  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao respSolicitacao                               *
 *   Recebe uma Solicitacao, executa as operacoes  ne- *
 *   cessarias e envia uma Resposta.                    *
 *                                                      *
 *    @param solicitacao String com a Solicitacao.      *
 *    @param resposta    String com a Resposta.         *
 *    @param socket      Ponteiro para o Socket.        *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int respSolicitacaoQsp(char *solicitacao, char *resposta, int socket) {
  strcpy(resposta, "");

  // Processa Comandos Remotos
  if (!strcmp(solicitacao, "")) return 1;
  // Alguem esta falando quem eu, Slave, sou. Enviando o ID
  else if (!strncmp(solicitacao, "YOU ARE ", 8)) {
    sscanf(solicitacao, "YOU ARE %d", &slaveID);
    sprintf(resposta, "005 - Ok. Agora eu tenha o ID %d.\r\n", slaveID);
    if (DEBUG>4) printf("[%d][respSolicitacao] YOU ARE Recebido. ID setado para %d !!\n", \
      slaveID, slaveID);
    return 1;
  }
  // Alguem quer colocar a lista de Slaves
  else if (!strncasecmp(solicitacao, "PUT SLAVES", 10)) {
    int  continua = 1;              // Se continua lendo os Numeros
    if (DEBUG>4) printf("[%d][respSolicitacao] PUT SLAVES Recebido.\n", slaveID);
    // Seta o Numero de Slaves para Um (Deixa o Primeiro em Branco)
    numHostsSlaves = 1;
    // Diz que esta pronto para receber a lista
    socketPuts(socket, "022 - OK. Pronto para receber lista de Slaves.\r\n");
    // Enqanto tiver linhas ..
    while (continua) {
      // Pega a proxima Linha
      if (socketGets(socket, solicitacao) < 0) continua = 0;
      else {
        if (!strncmp(solicitacao, "SLAVE ", 6)) {
          sscanf(solicitacao, "SLAVE %d %s %d", &hostsSlaves[numHostsSlaves].id, \
            hostsSlaves[numHostsSlaves].IP,\
             &hostsSlaves[numHostsSlaves].porta);
          if (DEBUG>3) printf("[%d][lerSlavesConexao] Adicionado Slave %d, %s, %d !!\n", slaveID, \
            hostsSlaves[numHostsSlaves].id, hostsSlaves[numHostsSlaves].IP, \
              hostsSlaves[numHostsSlaves].porta);
          hostsSlaves[numHostsSlaves].id = numHostsSlaves;
          numHostsSlaves++;
        }
        else if (!strncasecmp(solicitacao, "END SLAVES", 10)) {
          // Diz que recebeu a Lista
          socketPuts(socket, "022 - OK. Recebido os Slaves.\r\n");
          continua = 0;
        }
        else socketPuts(socket, "022 - ERRO. Esperando lista de Slaves. Saia com o comando END SLAVES.\r\n");
      }
    }
```

```
    return 1;
  }
  // Alguem quer enviar a Lista de Numeros pela Rede
  else if (!strncasecmp(solicitacao, "PUT LIST", 7)) {
    unsigned long int numeroAtual;  // Numero atualmente lido
    int  continua = 1;              // Se continua lendo os Numeros
    int  i=0;                       // Contador de Numeros Lidos
    if (DEBUG>4) printf("[%d][respSolicitacao] PUT LIST Recebido.\n", slaveID);
    // Cria uma Nova Lista de Numeros
    novaListaNum(&numLista);
    // Diz que esta pronto para receber os numeros
    socketPuts(socket, "012 - OK. Pronto para receber Lista de Numeros.\r\n");
    // Enquanto ainda tiver numeros ..
    while (continua) {
      // Recebe dados
      if (socketGets(socket, solicitacao) < 0) continua = 0;
      else {
        // Verifica se eh um Numero
        if (isNumero(solicitacao)) {
          sscanf(solicitacao, "%ld", &numeroAtual);
          if (DEBUG>14) printf("[%d][lerListaConexao] Adicionado Numero %ld !!\n", slaveID, numeroAtual);
          addNumero(numeroAtual, &numLista);
          //if (DEBUG>8) { printf("[%d][lerListaConexao] Adicionado Numero %d = %ld !!\n", \
            slaveID, i, numeroAtual); fflush(NULL); }
          if ( !(i%100000) &&  DEBUG>3) { printf("[%d][lerListaConexao] Adicionado Numero %d = %ld !!\n",\
            slaveID, i, numeroAtual); fflush(NULL); }
          i++;
        }
        else if (!strncasecmp(solicitacao, "END LIST", 10)) {
          socketPuts(socket, "014 - OK. Recebido a Lista.\r\n");
          continua = 0;
        }
        else socketPuts(socket, "011 - ERRO. Esperando lista de Numeros. Saia com o comando END LIST.\r\n");
      }
    }
    return 1;
  }
  // Alguem esta solicitando a Lista Final
  else if (!strncasecmp(solicitacao, "GET LIST", 10)) {
    int i;                     // Contador
    if (DEBUG>4) printf("[%d][respSolicitacao] GET LIST Recebido. Enviando Lista Final ..\n", slaveID);
    if (DEBUG>4) printf("[%d][respSolicitacao]     Lista Trocada: ", slaveID);
    if (DEBUG>4) printLista(&numLista);
    if (DEBUG>3) printf("[%d][respSolicitacao]     Executando o executarQSP ..\n", slaveID);
    executarQSP();
    if (DEBUG>3) printf("[%d][respSolicitacao]     executarQSP Terminado !! ..\n", slaveID);
    // Diz que vai comecar a enviar a lista
    socketPuts(socket, "052 - OK. Lista Trocada:\r\n");
    // Comeca a enviar os numeros
    if (numHostsSlaves==1) {
      if (DEBUG>3) printf("[%d][respSolicitacao]     Enviando Lista da Memoria !! ..\n", slaveID);
      if (DEBUG>4) printf("[%d][respSolicitacao]     NOVA LISTA: ", slaveID);
      if (DEBUG>4) printLista(&numLista);
      for (i=0; i<numLista.tamLista; i++) {
        sprintf(resposta, "%ld\r\n", numLista.numeros[i]);
        socketPuts(socket, resposta);
      }
    }
    else {
      if (DEBUG>3) printf("[%d][respSolicitacao]     Enviando Lista dos Threads !! ..\n", slaveID);
      if (DEBUG>4) printf("[%d][respSolicitacao]     NOVA LISTA (em duas partes): ", slaveID);
      if (DEBUG>4) printLista(&hostsSlaves[0].numLista);
      if (DEBUG>4) printLista(&hostsSlaves[1].numLista);
      for (i=0; i<hostsSlaves[0].numLista.tamLista; i++) {
        sprintf(resposta, "%ld\r\n", hostsSlaves[0].numLista.numeros[i]);
        socketPuts(socket, resposta);
      }
      for (i=0; i<hostsSlaves[1].numLista.tamLista; i++) {
        sprintf(resposta, "%ld\r\n", hostsSlaves[1].numLista.numeros[i]);
        socketPuts(socket, resposta);
      }
    }
```

```
    // Diz que eh o final da Lista e Nao espera nada do Master
    socketPuts(socket, "052 - Ok. Fim da Lista de Numeros\r\n");
    if (DEBUG>3) printf("[%d][respSolicitacao]     Terminei de Enviar as Listas !! ..\n", slaveID);
    return 1;
  }
  // Responde ao Comando de Ajuda (HELP)
  if (!strcasecmp(solicitacao, "HELP")) {
    strcpy(resposta, "001 - Os seguintes comandos são reconhecidos:\r\n\tHELP\tQSP\tQUIT\r\n001 -\
    Comentários para horacio@dcc.ufmg.br\r\n");
  }
  // Responde ao Comando de Saida (QUIT). Desconecta
  else if (!strcasecmp(solicitacao, "QUIT")) {
    if (DEBUG>4) printf("[%d][respSolicitacao] QUIT Recebido. Desconectando !!\n", slaveID);
    strcpy(resposta, "001 - T+ !!\r\n");
    return 0;
  }
  // Responde ao Comando de Saida (QUIT). Desconecta
  else if (!strcasecmp(solicitacao, "CLEAN UP")) {
    //int id;
    if (DEBUG>22) printf("[%d][respSolicitacao] CLEAN UP Recebido. Limpando !!\n", slaveID);
    fflush(NULL);
    // Limpando Memoria
    free(numLista.numeros);
    free(samLista.numeros);
    free(pivLista.numeros);
    strcpy(resposta, "001 - Tudo Limpo !!\r\n");
    return 1;
  }
  // Comando nao Reconhecido
  else strcpy(resposta, "003 - Comando não Reconhecido !!\r\n");
  return 1;
}
```

Arquivo ListaNumeros.c.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                *
 * --------- --------                                *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                          *
 *                                                   *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>        *
 *            http://www.dcc.ufmg.br/~horacio        *
 *                                                   *
 *    Principal Referência: Quinn (1994 - 10.5.3)    *
 *                                                   *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao novaListaNum                                *
 *    Cria uma nova Lista de Numeros Unsigned Long Int *
 *                                                    *
 *    @param lista Ponteiro de retorno da nova Lista.  *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
void novaListaNum(TLista *lista) {
  if (slaveID==-1) lista->numeros = (unsigned long int *) malloc(NUM_TAM * sizeof(unsigned long int));
  else             lista->numeros = (unsigned long int *) malloc((NUM_TAM/(numHostsSlaves)) * \
    sizeof(unsigned long int));
  lista->tamLista = 0;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao addNumero                                   *
 *    Adiciona um novo numero a lista.                *
 *                                                    *
 *    @param numero Numero a ser Adicionado.          *
 *    @param lista Ponteiro para a Lista.             *
```

```
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void addNumero(unsigned long int numero, TLista *lista) {
  if (DEBUG>44) printf("[addNumero] Adicionando Numero %ld ..\n", numero);
  lista->numeros[lista->tamLista] = numero;
  lista->tamLista++;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao addNumeroOrdenado                            *
 *    Adiciona um novo numero a lista em sua ordem.    *
 *                                                     *
 *    @param numero Numero a ser Adicionado.           *
 *    @param lista Ponteiro para a Lista.              *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void addNumeroOrdenado(unsigned long int numero, TLista *lista) {
  int i, n, esq=0, dir=0; // Contador

  if (DEBUG>14) printf("[addNumeroOrdenado] Adicionando Numero %ld ..\n", numero);

  // Determina a Posicao
  if (lista->tamLista == 0) n=0;
  else {
    esq = 0;
    dir = lista->tamLista-1;

    do {
      i = (int) ((esq+dir) / 2);
      if (numero > lista->numeros[i]) esq = i+1;
      else dir = i-1;
    } while (esq<dir);
  }

  if (numero<=lista->numeros[esq]) n = esq;
  else n=esq+1;

  // Libera Espaco para colocar o novo numero
  for (i=lista->tamLista; i>=n; i--) lista->numeros[i+1] = lista->numeros[i];
  // Inclui o Elemento
  lista->numeros[n] = numero;
  // Aumenta o tamanho da Lista
  lista->tamLista++;

}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao addNumerosAleatorios                         *
 *    Adiciona numeros aleatorios em uma lista.        *
 *                                                     *
 *    @param quantidade Quantidade de nums a adicionar *
 *    @param lista Ponteiro para a Lista.              *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void addNumerosAleatorios(int quantidade, TLista *lista) {
  int i;                          // Contador

  novaListaNum(lista);

  if (DEBUG>3) printf("[addNumerosAleatorio] Adicionando %d Numeros Aleatorios ..\n", quantidade);
  // Inicia a Geracao de Numeros Aleatorios
  iniciaMyRand();
  // Gera e Adicionar os Numeros
  for (i=0; i<quantidade; i++) {
    addNumero( myRand(MAX_RAND) , lista );
    if (!(i%1000000) && DEBUG>4) printf("[addNumerosAleatorio] Gerado Numero %d ..\n", i);
  }
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao printLista                                   *
 *    Imprime uma lista de numeros na  tela  caso  ela *
 *    nao seja muito grande.                           *
 *                                                     *
 *    @param lista Ponteiro para a Lista.              *
```

```
 ** * * * * * * * * * * * * * * * * * * * * * * * * * * */
void printLista(TLista *lista) {
  int i;
  if (DEBUG>3 && lista->tamLista<500) {
    printf("[printLista] Lista: ");
    for (i=0; i<lista->tamLista; i++) printf("%ld ", lista->numeros[i]);
    printf("\n");
  }
}


/* ********************************************** *
 * FUNCOES RELACIONADAS DIRETAMENTE AO   --MASTER--   *
 * ********************************************** */

/** * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER enviarListaThread                     *
 *    Thread executado pelo MASTER para cada  processo *
 *    slave para enviar a Lista de Numeros.            *
 *                                                     *
 *    @param voidargs Ponteiro para um HostSlave.      *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * * */
void *enviarListaThread(void *voidargs) {
  char solicitacao[MAXMSG];                      // String com a Solicitacao
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual
  int i = hostSlave->id * (numLista.tamLista / numHostsSlaves);
  int j = i + (numLista.tamLista / numHostsSlaves) - 1;

  if (DEBUG>8) printf("[enviaSlavesThread] Thread %d Criado. Enviando Lista de Numeros.\n", hostSlave->id);
  // Diz que vai colocar a lista
  socketPuts(hostSlave->socketFD, "PUT LIST\r\n");
  // Espera pela Resposta "012 - OK."
  if (socketGets(hostSlave->socketFD, solicitacao) >= 0) {
    // Envia os numeros
    if (ENVIO_ASCII) {
      for (; i<=j; i++) {
sprintf(solicitacao, "%ld\r\n", numLista.numeros[i]);
socketPuts(hostSlave->socketFD, solicitacao);
      }
    }
    else {
      novaListaNum(&hostSlave->numLista);
      for (; i<=j; i++) addNumero(numLista.numeros[i], &hostSlave->numLista);
      SendVector(hostSlave->socketFD, hostSlave->numLista.tamLista, hostSlave->numLista.numeros);
      free(hostSlave->numLista.numeros);
    }

    // Diz que acabou os numeros
    socketPuts(hostSlave->socketFD, "END LIST\r\n");
    // Espera pela resposta "014 - OK."
    socketGets(hostSlave->socketFD, solicitacao);

  }

  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER execOrdenaListaThread                 *
 *    Thread executado pelo MASTER para cada  processo *
 *    slave para solicitar que eles comecem a  ordenar *
 *    a listas de numeros.                             *
 *                                                     *
 *    @param voidargs Ponteiro para um HostSlave.      *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * * */
void *execOrdenaListaThread(void *voidargs) {
  char resposta[MAXMSG];                         // String com a Resposta
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual

  if (DEBUG>5) printf("[execOrdenaListaThread] Thread %d Criado. Pedindo para Ordenar as Listas.\n", \
    hostSlave->id);
```

```
  // Pede para o Slave Atual comecar a ordenar sua lista
  socketPuts(hostSlave->socketFD,"ORDER LIST\r\n");
  // Espera pela resposta "005 - Ok."
  socketGets(hostSlave->socketFD, resposta);

  if (DEBUG>5) printf("[execTrocaListaThread] Processo %d terminou de Ordenar a Lista.\n", hostSlave->id);

  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER execTrocaListaThread                  *
 *    Thread executado pelo MASTER para cada  processo *
 *    slave para solicitar que eles comecem  a  trocar *
 *    as listas de numeros entre si.                   *
 *                                                     *
 *    @param voidargs Ponteiro para um HostSlave.      *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void *execTrocaListaThread(void *voidargs) {
  char resposta[MAXMSG];                        // String com a Resposta
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual

  if (DEBUG>5) printf("[execTrocaListaThread] Thread %d Criado. Pedindo para Trocar as Listas.\n", \
    hostSlave->id);

  // Pede para o Slave Atual comecar a trocar as Listas
  socketPuts(hostSlave->socketFD,"EXCH LIST\r\n");
  // Espera pela resposta "044 - Ok."
  socketGets(hostSlave->socketFD, resposta);

  if (DEBUG>5) printf("[execTrocaListaThread] Processo %d terminou de Trocar as Listas.\n", hostSlave->id);

  // Termina a execucao do Thread
  pthread_exit (0);
}

void *execMergeListasThread(void *voidargs) {
  char resposta[MAXMSG];                        // String com a Resposta
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual

  if (DEBUG>3) printf("[execMergeListasThread] Thread %d Criado. Pedindo para Mergir as Listas.\n", hostSlave->id);

  // Pede para o Slave Atual comecar a trocar as Listas
  socketPuts(hostSlave->socketFD,"MERGE LISTS\r\n");
  // Espera pela resposta "044 - Ok."
  socketGets(hostSlave->socketFD, resposta);

  if (DEBUG>3) printf("[execMergeListasThread] Processo %d terminou de Mergir as Listas.\n", hostSlave->id);

  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER getListasTrocadasThread               *
 *    Thread executado pelo MASTER para cada  processo *
 *    slave para solicitar que eles comecem  a  enviar *
 *    a lista trocada final (unificada) para o Master. *
 *                                                     *
 *    @param voidargs Ponteiro para um HostSlave.      *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void *getListasTrocadasThread(void *voidargs) {
  char resposta[MAXMSG];                        // String com a Resposta
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual
  unsigned long int numeroAtual;                // Numero Temporario

  // Cria uma nova Lista
  novaListaNum(&hostSlave->numLista);
  if (DEBUG>5) printf("[getListasTrocadasThread] Thread %d Criado. Requisitando Listas Trocadas.\n", hostSlave->id);
  // Solicita o envio da Lista
```

```
  socketPuts(hostSlave->socketFD, "GET EXCH LIST\r\n");
  // Espera pela Resposta "052 - OK."
  socketGets(hostSlave->socketFD, resposta);
  // Comeca a receber os numeros
  if (ENVIO_ASCII) {
    while (socketGets(hostSlave->socketFD, resposta) >= 0) {
      if (strncasecmp(resposta, "052 - Ok. Fim da Lista de Troca", 30)) {
sscanf(resposta, "%ld", &numeroAtual);
addNumero(numeroAtual, &hostSlave->numLista);
if (DEBUG>44) printf("[getListasTrocadasThread] Thread %d: Recebendo numero %ld ..\n", \
  hostSlave->id, numeroAtual);
      }
      else break;
    }
  }
  else {
    ReceiveVector(hostSlave->socketFD, &hostSlave->numLista.tamLista, hostSlave->numLista.numeros);
    if (DEBUG>13) printf("[getListasTrocadasThread] Recebido a lista com %d elementos !!\n", \
      hostSlave->numLista.tamLista);
    socketGets(hostSlave->socketFD, resposta);
  }

  if (DEBUG>13) printf("[getListasTrocadasThread] Thread %d. Lista Trocada (%d):\n", \
    hostSlave->id, hostSlave->numLista.tamLista);
  if (DEBUG>5) printLista(&hostSlave->numLista);

  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER joinListas                            *
 *    Funcao executada pelo MASTER para cada  unificar *
 *    as listas recebidas pelos slaves.                *
 *                                                     *
 *    @param voidargs Ponteiro para um HostSlave.      *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
void joinListas() {
  int procId, i;
  TLista aux;
  novaListaNum(&numLista);
  for (procId=0;procId<numHostsSlaves;procId++){
    aux = hostsSlaves[procId].numLista;
    for (i=0; i<aux.tamLista; i++) addNumero(aux.numeros[i], &numLista);
  }
}

/* ************************************************* *
 * FUNCOES RELACIONADAS DIRETAMENTE AO   --SLAVE--    *
 * ************************************************* */

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao SLAVE lerListaConexao.                       *
 *    Le uma lista de numeros no SLAVE  enviadas  pelo *
 *    master a partir de uma conexao.                  *
 *                                                     *
 *    @param socket Socket de onde ler os numeros.     *
 *    @return Se deu tudo certo ou nao.                *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
int lerListaConexao(int socket) {
  char solicitacao[MAXMSG];       // String de Solicitacao
  unsigned long int numeroAtual;  // Numero atualmente lido
  int  continua = 1;              // Se continua lendo os Numeros
  int  i=0;                       // Contador de Numeros Lidos
  int numElem;

  // Cria uma Nova Lista de Numeros
  novaListaNum(&numLista);
  // Diz que esta pronto para receber os numeros
  socketPuts(socket, "012 - OK. Pronto para receber Lista de Numeros.\r\n");
  // Enquanto ainda tiver numeros ..
  if (ENVIO_ASCII) {
```

```
    while (continua) {
      // Recebe dados
      if (socketGets(socket, solicitacao) < 0) continua = 0;
      else {
// Verifica se eh um Numero
if (isNumero(solicitacao)) {
  sscanf(solicitacao, "%ld", &numeroAtual);
  if (DEBUG>14) printf("[%d][lerListaConexao] Adicionado Numero %ld !!\n", slaveID, numeroAtual);
  addNumero(numeroAtual, &numLista);
  if ( !(i%100000) &&  DEBUG>3) { printf("[%d][lerListaConexao] Adicionado Numero %d = %ld !!\n",\
     slaveID, i, numeroAtual); fflush(NULL); }
  i++;
}
else if (!strncasecmp(solicitacao, "END LIST", 10)) {
  socketPuts(socket, "014 - OK. Recebido a Lista.\r\n");
  continua = 0;
}
else socketPuts(socket, "011 - ERRO. Esperando lista de Numeros. Saia com o comando END LIST.\r\n");
      }
    }
  }
  else {
    ReceiveVector(socket, &numElem, numLista.numeros);
    numLista.tamLista = numElem;
    if (DEBUG>13) printf("[%d][lerListaConexao] Recebido a lista com %d elementos !!\n", slaveID, numElem);
    socketGets(socket, solicitacao);  if (DEBUG>3) printf("[%d][lerListaConexao] Final da Lista: %s ..\n", \
      slaveID, solicitacao);
    socketPuts(socket, "014 - OK. Recebido a Lista.\r\n");
  }

  // Ordena a Lista Recebida
  // quickSort(&numLista);

  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao SLAVE execTrocaListaThread                 *
 *    Thread executado pelo SLAVE  para cada  processo *
 *    slave para enviar parte da Lista de Numeros para *
 *    este Slave Atual.                                *
 *                                                     *
 *    @param voidargs Ponteiro para um HostSlave.      *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void *trocaListaThread(void *voidargs) {
  char solicitacao[MAXMSG];                       // String com a Solicitacao
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual
  unsigned long int min, max;                    // Min e Max a enviar
  int i;                                         // Contador

  // Define os numeros minimo e maximo a ser enviado
  min = pivLista.numeros[hostSlave->id];
  max = pivLista.numeros[hostSlave->id + 1];

  // Verifica se o Thread Atual Slave eh do Processo Slave Atual. Se for,
  //   manda a lista diretamente, sem precisar usar o socket.
  if (slaveID == hostSlave->id) {
    novaListaNum(&hostSlave->numLista);
    for (i=0; (numLista.numeros[i]<=max && i<numLista.tamLista); i++)
      if (numLista.numeros[i]>min) addNumero(numLista.numeros[i], &hostSlave->numLista);
    if (DEBUG>4) {
      printf("[%d][Thd %d][trocaListaThread] Recebido lista de %d: ", slaveID, hostSlave->id, slaveID);
      printLista(&hostSlave->numLista);
    }
  }
  // Vai usar o Socket ..
  else {
    // Conecta no Slave Atual
    if ((hostSlave->socketFD = conecta(hostSlave->IP, hostSlave->porta)) <= 0) {
      pthread_exit (0);
    }
    // Diz que vai comecar a enviar a Lista de Troca
```

```
    sprintf(solicitacao, "PUT EXCH LIST %d\r\n", slaveID);
    socketPuts(hostSlave->socketFD, solicitacao);
    // Espera por "032 - OK."
    if (socketGets(hostSlave->socketFD, solicitacao) >= 0) {
      if (ENVIO_ASCII) {
// Comeca a enviar os numeros se eles forem maior que o Minimo
for (i=0; (numLista.numeros[i]<=max && i<numLista.tamLista); i++) {
  if (numLista.numeros[i]>min) {
    sprintf(solicitacao, "%ld\r\n", numLista.numeros[i]);
    socketPuts(hostSlave->socketFD, solicitacao);
  }
}
      }
      else {
TLista tempLista;
novaListaNum(&tempLista);
for (i=0; (numLista.numeros[i]<=max && i<numLista.tamLista); i++)
   if (numLista.numeros[i]>min)  addNumero(numLista.numeros[i], &tempLista);
SendVector(hostSlave->socketFD, tempLista.tamLista, tempLista.numeros);
free(tempLista.numeros);
      }


      // Diz que a Lista de Troca Acabou
      socketPuts(hostSlave->socketFD, "END EXCH LIST\r\n");
      // Espera por "034 - OK."
      socketGets(hostSlave->socketFD, solicitacao);
    }

    // A Conexao nao sera mais necessaria. Desconecta.
    desconecta(hostSlave->socketFD);
  }
  printf("[%d][Thd %d][trocaListaThread] Ternminei a troca .. \n", slaveID, hostSlave->id);

  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao SLAVE receberTrocaLista                        *
 *    Um Slave se conectou neste Slave  e  pediu  para *
 *    comecar a enviar  uma lista de troca.  Vamos re- *
 *    cebe-la e salva-la.                               *
 *                                                      *
 *    @param socket Socket de onde ler a lista troca.  *
 *    @param procID Numero do Slave que esta enviando. *
 *    @return Se deu tudo certo ou nao.                 *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int receberTrocaLista(int socket, int procID) {
  int continua = 1;              // Se ele continua lendo a lista
  char solicitacao[MAXMSG];      // String com a Solicitacao
  unsigned long int numeroAtual; // Numero Temporario

  // Cria uma Nova lista no SLAVE no Processo ProcID
  novaListaNum(&hostsSlaves[procID].numLista);

  // Diz que esta pronto para receber a lista
  socketPuts(socket, "032 - OK. Pronto para receber Lista de Troca.\r\n");
  if (ENVIO_ASCII) {
    while (continua) {
      // Recebe os Dados
      if (socketGets(socket, solicitacao) < 0) continua = 0;
      else {
// Verifica se e um numero. Se for Adiciona.
if (isNumero(solicitacao)) {
  sscanf(solicitacao, "%ld", &numeroAtual);
  if (DEBUG>18) printf("[%d][receberTrocaLista] Recebido %ld de %d !!\n", \
    slaveID, numeroAtual, procID);
  addNumero(numeroAtual, &hostsSlaves[procID].numLista);
}
// Verifica se e o fim da lista
else if (!strncasecmp(solicitacao, "END EXCH LIST", 10)) {
```

```
  socketPuts(socket, "034 - OK. Recebido a Lista.\r\n");
  continua = 0;
}
// Ops .. Comando Invalido.
else socketPuts(socket, "031 - ERRO. Esperando lista de Troca. Saia com o comando END EXCH LIST.\r\n");
      }
    }
  }
  else {
    ReceiveVector(socket, &hostsSlaves[procID].numLista.tamLista, hostsSlaves[procID].numLista.numeros);
    if (DEBUG>13) printf("[%d][receberTrocaLista] Recebido a lista com %d elementos de %d!!\n", \
      slaveID, hostsSlaves[procID].numLista.tamLista, procID);
    socketGets(socket, solicitacao);  if (DEBUG>3) printf("[%d][receberTrocaLista] Final da Lista: %s ..\n", \
      slaveID, solicitacao);
    socketPuts(socket, "034 - OK. Recebido a Lista.\r\n");
  }

  if (DEBUG>14) {
    printf("[%d][receberTrocaLista] Recebido lista de %d COM %d ELEMENTOS: \n",\
       slaveID, procID, hostsSlaves[procID].numLista.tamLista);
    printLista(&hostsSlaves[procID].numLista);
    printf("\n\n[%d][receberTrocaLista] CERIFICANDO ORDEVACAO : \n\n", slaveID);
    verificaOrdenacao(&hostsSlaves[procID].numLista);
  }

  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao SLAVE enviarListaTrocada                     *
 *    Envia, por solicitacao do Master, a lista troca- *
 *    da do SLAVE atual.                               *
 *                                                     *
 *    @param socket Socket de onde ler a lista troca.  *
 *    @return Se deu tudo certo ou nao.                *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int enviarListaTrocada(int socket) {
  char resposta[MAXMSG];    // String com a Resposta
  int i;                    // Contador

  // Antes de Enviar, vamos unificar as listas recebidas.
  //gerarListaTrocada();

  if (DEBUG>3) printf("[%d][enviarListaTrocada]  Enviando lista Trocada .. ", slaveID);

  // Diz que vai comecar a enviar a lista
  socketPuts(socket, "052 - OK. Lista Trocada:\r\n");
  if (ENVIO_ASCII) {
    // Nao espera nada do Master e comeca a enviar os numeros
    for (i=0; i<numLista.tamLista; i++) {
      sprintf(resposta, "%ld\r\n", numLista.numeros[i]);
      socketPuts(socket, resposta);
    if (!(i%1000000) && DEBUG>2) printf("[enviarListaTrocada] Numero %d = %ld\n", i, numLista.numeros[i]);
    }
    // Diz que eh o final da Lista e Nao espera nada do Master
  }
  else {
    SendVector(socket, numLista.tamLista, numLista.numeros);
  }
  socketPuts(socket, "052 - Ok. Fim da Lista de Troca\r\n");

  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao SLAVE gerarListaTrocada                      *
 *    O SLAVE atual recebeu a Lista Trocada  de  todos *
 *    os slaves. Vamos fazer um merge das listas.      *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void gerarListaTrocada() {
  int i;                       // Contador
```

```
  free(numLista.numeros);
  // Comeca dizendo que a lista unificada eh igual a primeira lista trocada
  numLista = hostsSlaves[0].numLista;
  // Adiciona Ordenadamente (MERGE) os numeros das outras listas na lista unificada
  for (i=1; i<numHostsSlaves; i++) {
    if (DEBUG>4) printf("[%d][gerarListaTrocada]  Lista do Processo %d: ", slaveID, i);
    if (DEBUG>4) printLista(&hostsSlaves[i].numLista);
    mergeListas(&numLista, &hostsSlaves[i].numLista);
    free(hostsSlaves[i].numLista.numeros);
  }
  // quickSort(&numLista);
  if (DEBUG>4) printf("[%d][gerarListaTrocada]  Lista Final Trocada: ", slaveID);
  if (DEBUG>4) printLista(&numLista);
}
```

Arquivo ListaPivots.c.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                  *
 * --------- --------                                  *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                            *
 *                                                     *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>          *
 *            http://www.dcc.ufmg.br/~horacio          *
 *                                                     *
 *    Principal Referência: Quinn (1994 - 10.5.3)      *
 *                                                     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao novaListaPiv                                  *
 *    Cria uma nova Lista de Pivots Unsigned Long Int. *
 *                                                      *
 *    @param lista Ponteiro de retorno da nova Lista.  *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void novaListaPiv(TLista *lista) {
  lista->numeros = (unsigned long int *) malloc(PIV_TAM * sizeof(unsigned long int));
  lista->tamLista = 0;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER gerarPivots                            *
 *    Gera os Pivots no Master com base  na  Lista  de *
 *    Samples recebida pelos Slaves.                    *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void gerarPivots() {
  int i;                          // Contador

  // Cria a Lista de Pivots
  novaListaPiv(&pivLista);
  // Adiciona o numero 0 a Lista de Pivots
  addNumero(0, &pivLista);
  // Adiciona os outros Numeros na Lista
  for (i=1; i<=numHostsSlaves-1; i++) {
    addNumero( samLista.numeros[i*numHostsSlaves + ((int) numHostsSlaves/2)-1], &pivLista );
  }
  // Adiciona o maior Numero possivel na Lista de Pivots
  addNumero(LONG_MAX, &pivLista);

  if (DEBUG>4) printf("[GeraPivots] Pivots Gerados: ");
  if (DEBUG>4) printLista(&pivLista);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER enviarPivotsThread                     *
 *    Thread executado pelo MASTER para cada  processo *
```

```
 *    slave para enviar a Lista de Pitovs.            *
 *                                                    *
 *    @param voidargs Ponteiro para um HostSlave.     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
void *enviarPivotsThread(void *voidargs) {
  char resposta[MAXMSG];                        // String com a Resposta
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual
  int i;                                        // Contador

  if (DEBUG>4) printf("[putPivotsThread] Thread %d Criado. Enviando Pivots.\n", hostSlave->id);
  // Diz que vai enviar os Pivots ao Slave Atual
  socketPuts(hostSlave->socketFD, "PUT PIVOTS\r\n");
  // Espera a resposta "032 - OK."
  socketGets(hostSlave->socketFD, resposta);
  // Comeca a enviar a lista
  for (i=0; i<pivLista.tamLista; i++) {
    sprintf(resposta, "%ld\r\n", pivLista.numeros[i]);
    socketPuts(hostSlave->socketFD, resposta);
  }
  // Diz que acabou a Lista de Pivots
  socketPuts(hostSlave->socketFD, "END PIVOTS\r\n");
  // Espera pela rsposta "034 - OK."
  socketGets(hostSlave->socketFD, resposta);

  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao SLAVE receberPivots                         *
 *    Le uma lista de Pivots  no SLAVE  enviadas  pelo *
 *    master a partir de uma conexao.                 *
 *                                                    *
 *    @param socket Socket de onde ler os numeros.    *
 *    @return Se deu tudo certo ou nao.               *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
int receberPivots(int socket) {
  char solicitacao[MAXMSG];       // String de Solicitacao
  unsigned long int numeroAtual;  // Numero atualmente lido
  int  continua = 1;              // Se continua lendo os Numeros

  // Cria a Lista de Pivots no Slave
  novaListaPiv(&pivLista);
  // Diz que esta pronto para receber a lista
  socketPuts(socket, "032 - OK. Pronto para receber Lista de Pivots.\r\n");
  // Recebe os numeros
  while (continua) {
    if (socketGets(socket, solicitacao) < 0) continua = 0;
    else {
      if (isNumero(solicitacao)) {
        sscanf(solicitacao, "%ld", &numeroAtual);
        if (DEBUG>3) printf("[%d][receberPivots] Recebido Pivot %ld !!\n", slaveID, numeroAtual);
        // Adiciona o numero na lista de Pivots
        addNumero(numeroAtual, &pivLista);
      }
      else if (!strncasecmp(solicitacao, "END PIVOTS", 10)) {
        // Confirma que recebeu a lista
        socketPuts(socket, "034 - OK. Recebido a Lista.\r\n");
        continua = 0;
      }
      else socketPuts(socket, "031 - ERRO. Esperando lista de Pivots. Saia com o comando END PIVOTS.\r\n");
    }
  }

  return 1;
}
```

Arquivo Slaves.c.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                   *
 * --------- --------                                   *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                            *
 *                                                      *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>          *
 *            http://www.dcc.ufmg.br/~horacio          *
 *                                                      *
 *    Principal Referência: Quinn (1994 - 10.5.3)      *
 *                                                      *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER conectaSlaves                          *
 *   Conecta nos Slaves.                                *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int conectaSlaves() {
  int procId;                    // ID temporario do Processo Atual

  // Abre Conexao com cada um dos Slaves
  for (procId=0;procId<numHostsSlaves;procId++) {
    if ((hostsSlaves[procId].socketFD = conecta(hostsSlaves[procId].IP, hostsSlaves[procId].porta)) <= 0) {
      printf("[conectaSlaves] Nao foi possivel Conectar a %s porta %d !!\n", hostsSlaves[procId].IP,\
        hostsSlaves[procId].porta);
      return 0;
    }
  }
  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER desconectaSlaves                       *
 *   Desconecta dos Slaves.                             *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int desconectaSlaves() {
  int procId;                    // ID Temporario do Processo Atual

  // Desconecta cada um dos Slaves
  for (procId=0;procId<numHostsSlaves;procId++){
    if ( (hostsSlaves[procId].socketFD = desconecta(hostsSlaves[procId].socketFD)) < 0){
      printf("[desconectaSlaves] Nao foi possivel Conectar a %s porta %d !!\n", hostsSlaves[procId].IP,\
        hostsSlaves[procId].porta);
      return 0;
    }
  }
  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER lerSlavesArq.                          *
 *    Ler os escravos a partir de um arquivo.          *
 *                                                      *
 *    @param arqEntrada Nome do Arquivo de Entrada     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int lerSlavesArq(char *arqEntrada) {
  FILE *arqEntradaFD;            // Arquivo de Entrada
  char linha[256];               // Linha atual do Arquivo de Entrada

  // Tenta abrir o Arquivo de Entrada
  arqEntradaFD = abreArquivo(arqEntrada, "r");
  if (arqEntradaFD == NULL) {
    printf("[lerEntrada] Erro no abreArqEntrada. Saindo.\n");
    return 0;
  }
  numHostsSlaves = 0;
  // Le cada uma das Linhas
  while (fgets(linha, sizeof(linha), arqEntradaFD)) {
```

```
    chomp(linha);
    hostsSlaves[numHostsSlaves].id = numHostsSlaves;
    hostsSlaves[numHostsSlaves].socketFD = 0;
    sscanf(linha, "%s %d %s", hostsSlaves[numHostsSlaves].IP, &hostsSlaves[numHostsSlaves].porta,\
      hostsSlaves[numHostsSlaves].arqEntrada);
    numHostsSlaves++;
  }
  // Fecha o Arquivo
  fclose(arqEntradaFD);
  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER enviarSlavesThread                  *
 *    Thread executado pelo MASTER para cada  processo *
 *    slave para enviar a Lista de Slaves.            *
 *                                                    *
 *    @param voidargs Ponteiro para um HostSlave.     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
void *enviarSlavesThread(void *voidargs) {
  char solicitacao[MAXMSG];                     // String com a Solicitacao
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual
  int i;                                        // Contador

  if (DEBUG>8) printf("[enviaSlavesThread] Thread %d Criado. Enviando Slaves.\n", hostSlave->id);
  // Diz que vai comecar a enviar os Slaves
  socketPuts(hostSlave->socketFD, "PUT SLAVES\r\n");
  // Espera pela Resposta "022 - OK."
  if (socketGets(hostSlave->socketFD, solicitacao) >= 0) {
    // Comeca a enviar a Lista de Slaves
    for (i=0; i<numHostsSlaves; i++) {
      sprintf(solicitacao, "SLAVE %d %s %d\r\n", hostsSlaves[i].id, hostsSlaves[i].IP, \
        hostsSlaves[i].porta);
      socketPuts(hostSlave->socketFD, solicitacao);
    }
    // Diz que aLista Terminou
    socketPuts(hostSlave->socketFD, "END SLAVES\r\n");
    socketGets(hostSlave->socketFD, solicitacao);
  }
  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER enviarIDsThread                      *
 *    Thread executado pelo MASTER para cada  processo *
 *    slave para enviar os IDs dos Slaves.            *
 *                                                    *
 *    @param voidargs Ponteiro para um HostSlave.     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
void *enviarIDsThread(void *voidargs) {
  char solicitacao[MAXMSG];                     // String com a Solicitacao
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual

  if (DEBUG>8) printf("[enviaSlavesThread] Thread %d Criado. Enviando ID.\n", hostSlave->id);
  // Diz quem o Slave e
  sprintf(solicitacao, "YOU ARE %d\r\n", hostSlave->id);
  socketPuts(hostSlave->socketFD, solicitacao);
  // Espera pela Resposta "005 - Ok."
  socketGets(hostSlave->socketFD, solicitacao);
  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER cleanUpThread                        *
 *    Thread executado pelo MASTER para cada  processo *
 *    slave para limpar a memoria deles.              *
 *                                                    *
 *    @param voidargs Ponteiro para um HostSlave.     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
void *cleanUpThread(void *voidargs) {
```

```
  char solicitacao[MAXMSG];                       // String com a Solicitacao
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual

  if (DEBUG>13) printf("[enviaSlavesThread] Thread %d Criado. Enviando Clean Up.\n", hostSlave->id);
  // Diz quem o Slave e
  socketPuts(hostSlave->socketFD, "CLEAN UP\r\n");
  // Espera pela Resposta "001 - Ok."
  socketGets(hostSlave->socketFD, solicitacao);
  // Termina a execucao do Thread
  pthread_exit (0);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao SLAVE lerSlavesConexao.                      *
 *    Ler os escravos enviados pelo master a partir de *
 *    uma Conexao.                                     *
 *                                                     *
 *    @param socket Ponteiro para o Socket             *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int lerSlavesConexao(int socket) {
  char solicitacao[MAXMSG];       // String de Solicitacao
  int  continua = 1;              // Se continua lendo os Numeros

  // Seta o Numero de Slaves para Nenhum
  numHostsSlaves = 0;
  // Diz que esta pronto para receber a lista
  socketPuts(socket, "022 - OK. Pronto para receber lista de Slaves.\r\n");
  // Enqanto tiver linhas ..
  while (continua) {
    // Pega a proxima Linha
    if (socketGets(socket, solicitacao) < 0) continua = 0;
    else {
      if (!strncmp(solicitacao, "SLAVE ", 6)) {
        sscanf(solicitacao, "SLAVE %d %s %d", &hostsSlaves[numHostsSlaves].id, hostsSlaves[numHostsSlaves].IP,\
          &hostsSlaves[numHostsSlaves].porta);
        if (DEBUG>4) printf("[%d][lerSlavesConexao] Adicionado Slave %d, %s, %d !!\n", slaveID,\
          hostsSlaves[numHostsSlaves].id, hostsSlaves[numHostsSlaves].IP, hostsSlaves[numHostsSlaves].porta);
        numHostsSlaves++;
      }
      else if (!strncasecmp(solicitacao, "END SLAVES", 10)) {
        // Diz que recebeu a Lista
        socketPuts(socket, "022 - OK. Recebido os Slaves.\r\n");
        continua = 0;
      }
      else socketPuts(socket, "022 - ERRO. Esperando lista de Slaves. Saia com o comando END SLAVES.\r\n");
    }
  }
  return 1;
}
```

Arquivo Ordenacao.c.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                  *
 * --------- --------                                  *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                            *
 *                                                     *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>          *
 *            http://www.dcc.ufmg.br/~horacio          *
 *                                                     *
 *    Principal Referência: Quinn (1994 - 10.5.3)      *
 *                                                     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao quickSort                                    *
```

```
 *    Ordena uma lista utilizando o QuickSort  sequen- *
 *    cial.                                            *
 *                                                     *
 *    @param lista Ponteiro para a Lista.              *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void quickSort(TLista *lista) {
  ordena(0, lista->tamLista-1, lista->numeros);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao ordena                                       *
 *    Funcao utilizada pelo QuickSort.                 *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void ordena(int esq, int dir, unsigned long int *A) {
  int i, j;
  particao(esq, dir, &i, &j, A);
  if (esq<j) ordena(esq, j, A);
  if (i<dir) ordena(i, dir, A);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao particao                                     *
 *    Funcao utilizada pelo QuickSort.                 *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void particao(int esq, int dir, int *i, int *j, unsigned long int *A) {
  int x, w;
  *i = esq;
  *j = dir;

  x = A[(*i + *j) / 2];

  if (DEBUG>13) printf("[particao] Pivo eh %d ..\n", x);

  do {
    while (x > A[*i]) (*i)++;
    while (x < A[*j]) (*j)--;
    if (*i<=*j) {
      w = A[*i];
      A[*i] = A[*j];
      A[*j] = w;
      (*i)++;
      (*j)--;
    }
  } while (*i<=*j);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao verificaOrdenacao                            *
 *    Verifica se uma lista esta ordenada.             *
 *                                                     *
 *    @param lista Ponteiro para a Lista.              *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void verificaOrdenacao(TLista *lista) {
  unsigned long ultimo = 0;       // Ultimo Numero Verificado
  int i, ok = 1;                  // Contador e Se estah Ok

  // Verifica os Numeros
  for (i=0; i<lista->tamLista; i++) {
    if (ultimo > lista->numeros[i]) {
      if (DEBUG>3) printf("[verificaOrdenacao]     Numero na posicao %d (%ld) maior na posicao %d (%ld) !!\n",\
         i, lista->numeros[i], i-1, lista->numeros[i-1]);
      ok = 0;
    }
    ultimo = lista->numeros[i];
  }
  if (DEBUG>3) printf("[verificaOrdenacao]     Numero de Elementos: %d\n", lista->tamLista);
  if (ok  && DEBUG>3) printf("[verificaOrdenacao]     Ordenacao OK !! !!\n");
  if (!ok && DEBUG>2) printf("[verificaOrdenacao]     ERRO na ordem !! !!\n");
}


/** * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
 * Funcao mergeListas                              *
 *    Faz um merge de duas listas ordenadas.        *
 *                                                  *
 *    @param lista1 Ponteiro para a primeira Lista.  *
 *    @param lista2 Ponteiro para a segunda Lista.   *
 *    @return Retorno na lista1                      *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
void mergeListas(TLista *lista1, TLista *lista2) {
  TLista listaMerged;
  int i, j;

  novaListaNum(&listaMerged);
  for (i=0, j=0; (i<lista1->tamLista || j<lista2->tamLista); ) {
    if (i<lista1->tamLista && j<lista2->tamLista) {
      if (lista1->numeros[i] <= lista2->numeros[j]) {
if (DEBUG>12) printf("[mergeListas] %ld <= %ld ou %d<%d .. Adicionando %ld em Lista 1 ..\n",\
  lista1->numeros[i], lista2->numeros[j], i, lista1->tamLista, lista1->numeros[i]);
addNumero(lista1->numeros[i], &listaMerged);
i++;
      }
      else {
if (DEBUG>12) printf("[mergeListas] %ld > %ld ou %d<%d .. Adicionando %ld em Lista 1 ..\n",\
  lista1->numeros[i], lista2->numeros[j], j, lista2->tamLista-1, lista2->numeros[j]);
addNumero(lista2->numeros[j], &listaMerged);
j++;
      }
    }
    else {
      if (i<lista1->tamLista) {
addNumero(lista1->numeros[i], &listaMerged);
i++;
      }
      else {
addNumero(lista2->numeros[j], &listaMerged);
j++;
      }
    }
  }
  free(lista1->numeros);
  //free(lista2->numeros);
  *lista1 = listaMerged;
}
```

Arquivo `Arquivo.c`.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                *
 * --------- --------                                *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                          *
 *                                                   *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>        *
 *            http://www.dcc.ufmg.br/~horacio        *
 *                                                   *
 *    Principal Referência: Quinn (1994 - 10.5.3)    *
 *                                                   *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao abreArquivo.                                *
 *    Abre o arquivo de entrada com os dados.         *
 *                                                    *
 *    @param nomeArqEntrada Nome do arquivo.          *
 *    @param arqEntrada  Ponteiro para o arquivo.     *
 *    @result Sucesso ou Erro.                        *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
FILE *abreArquivo(char *nomeArquivo, char *modo) {
```

```
  FILE *arquivoFD;                // Ponteiro para o arquivo

  if (DEBUG>8) printf("[abreArquivo] Abrindo Arquivo %s .. \n", nomeArquivo);
  if ((arquivoFD = fopen(nomeArquivo, modo)) == NULL) {
    printf("Erro ao abrir arquivo %s !! \n", nomeArquivo);
    return NULL;
  }

  return arquivoFD;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao lerListaArquivo                              *
 *    Le o arquivo numero por numero e os adiciona na  *
 *    Lista de Numeros.                                *
 *                                                     *
 *    @param lista Lista onde os numeros ficarao.      *
 *    @param arqEntrada  Ponteiro para o arquivo.      *
 *    @result Sucesso ou Erro.                         *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int lerListaArquivo(TLista *lista, char *arqEntrada) {
  FILE *arqEntradaFD;             // Arquivo de Entrada
  unsigned long int numeroAtual;  // Numero Temporario

  novaListaNum(lista);

  // Tenta abrir o Arquivo de Entrada
  arqEntradaFD = abreArquivo(arqEntrada, "rb");
  if (arqEntradaFD == NULL) {
    printf("[lerListaArquivo] Erro ao ler arquivo %s.\n", arqEntrada);
    return 0;
  }
  // Percorre o arquivo o adiciona os numeros a Lista
  while (!feof(arqEntradaFD)) {
    if (fread(&numeroAtual, sizeof(numeroAtual), 1, arqEntradaFD)) addNumero(numeroAtual, lista);
    if (!(lista->tamLista%1000000) && DEBUG>3) printf("[lerListaArquivo] Numero %d = %ld\n",\
      lista->tamLista, numeroAtual);
  }
  if (DEBUG>4) printf("[lerListaArquivo] Lido %d Numeros !!\n", lista->tamLista);
  // Fecha o arquivo
  fclose(arqEntradaFD);

  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao salvaListaArquivo                            *
 *    Le os numeros de uma lista e os salva em um  Ar- *
 *    quivo Binario.                                   *
 *                                                     *
 *    @param lista Lista de Numeros.                   *
 *    @param arqSaida Ponteiro para o arquivo.         *
 *    @result Sucesso ou Erro.                         *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int salvaListaArquivo(TLista *lista, char *arqSaida) {
  FILE *arqSaidaFD;     // Arquivo de Entrada
  int i;

  if (DEBUG>4) printf("[salvaListaArquivo] Salvando a lista do Arquivo %s ..\n", arqSaida);
  // Tenta abrir o Arquivo de Entrada
  arqSaidaFD = abreArquivo(arqSaida, "wb");
  if (arqSaidaFD == NULL) {
    printf("[lerEntrada] Erro no abreArqEntrada. Saindo.\n");
    return 0;
  }
  // Salva as informacoes da Lista no Arquivo Binario
  for (i=0; i<lista->tamLista; i++)  {
    fwrite(&lista->numeros[i], sizeof(lista->numeros[i]), 1, arqSaidaFD);
    if (!(i%1000000) && DEBUG>4) printf("Salvo Numero %d = %ld ..\n", i, lista->numeros[i]);
  }
  // Fecha o Arquivo
  fclose(arqSaidaFD);
```

```
  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao MASTER enviarListaArqThread                  *
 *    Thread executado pelo MASTER para cada  processo *
 *    slave para solicitar que estes leiam  o  arquivo *
 *    de entrada localmente.                           *
 *                                                     *
 *    @param voidargs Ponteiro para um HostSlave.      *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void *enviarListaArqThread(void *voidargs) {
  char solicitacao[MAXMSG];                    // String com a Solicitacao
  HostSlave *hostSlave = (HostSlave *)voidargs;  // Host Slave do Thread Atual

  if (DEBUG>8) printf("[enviaSlavesThread] Thread %d Criado. Pedindo para ler o arq %s.\n",\
    hostSlave->id, hostSlave->arqEntrada);
  // Enviando Solicitacao de Leitura
  sprintf(solicitacao, "READ %s\r\n", hostSlave->arqEntrada);
  socketPuts(hostSlave->socketFD, solicitacao);
  // Espera pela a Resposta "005 - Ok. .."
  socketGets(hostSlave->socketFD, solicitacao);

  if (DEBUG>8) printf("[enviaSlavesThread] Processo %d terminou de ler o arquivo.\n", hostSlave->id);

  // Termina a execucao do Thread
  pthread_exit (0);
}
```

Arquivo Utils.c.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                  *
 * --------- --------                                  *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                            *
 *                                                     *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>          *
 *            http://www.dcc.ufmg.br/~horacio          *
 *                                                     *
 *    Principal Referência: Quinn (1994 - 10.5.3)      *
 *                                                     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao criaThreads                                  *
 *    Cria um thread para cada processo Slave e execu- *
 *    ta a funcao passada por parâmetro.               *
 *                                                     *
 *    @param funcThread Funcao executada no Thread.    *
 *    @return           Sucesso ou Nao                 *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int criaThreads(void *funcThread) {
  int procId;
  for (procId=0;procId<numHostsSlaves;procId++){
    pthread_create(&hostsSlaves[procId].thread, NULL,funcThread,&hostsSlaves[procId]);
  }
  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao joinThreads                                  *
 *    Espera o termino da execucao de todos os Threads *
 *                                                     *
 *    @return Sucesso ou Nao                           *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
int joinThreads() {
  int procId;
  for (procId=0;procId<numHostsSlaves;procId++){
    if (DEBUG>5) printf("[joinThreads] Esperando pelo processo %d .. \n", procId);
    if (pthread_join(hostsSlaves[procId].thread,NULL) != 0 ) printf("\nERRO no Thread JOIN !! !!\n");
  }
  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao printAjuda.                                 *
 *    Imprime a Ajuda do Sistema.                     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void printAjuda() {
    printf("[main] Utilizacao ..\n");
    printf("[main]       Parallel Sorting by Regular Sampling:\n");
    printf("[main]           - Master: # QSP -s <NomeArquivoClientes>\n");
    printf("[main]           - Slave:  # QSP -c <NumPortaTCP>\n");
    printf("[main]       Parallel QuickSort:\n");
    printf("[main]           - Master: # QSP -t -1\n");
    printf("[main]           - Slave:  # QSP -t <NumPortaTCP>\n");
    printf("[main]       Quicksort Sequencial:\n");
    printf("[main]           -         # QSP -q <NomeArqNumeros>\n");
    printf("[main]       Gerar Massa de Dados:\n");
    printf("[main]           -         # QSP -g <QuantidadeNumeros> <SaidaArquivo>\n");
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao chmop.                                      *
 *    Retira o retorno de linha do final de uma string *
 *    caso esse caractere exista. Diminui o  tamamanho *
 *    da string em 1.                                 *
 *                                                    *
 *    @param string Ponteiro para a string modificada. *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void chomp(char *string) {
  if (string[strlen(string)-1]=='\n') string[strlen(string)-1] = string[strlen(string)];
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao stricmp                                     *
 *    Comparacao  de  Caracteres  sem  Case  Sensitive *
 *    compativel para Windows e Unix.                 *
 *                                                    *
 *    @param str1 Primeira String.                    *
 *    @param str2 Segunda String.                     *
 *    @return Numero de Diferencas.                   *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
#ifdef _IS_UNIX_
int stricmp(char *str1, char *str2) {
  return (strcasecmp(str1, str2));
}
#endif

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao iniciaMyRand                                *
 *    Inicia a geracao de Numeros Aleatorios.         *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
void iniciaMyRand() {
  struct timeval tempo;
  gettimeofday(&tempo, NULL);

  srand((long) tempo.tv_usec);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao myRand.                                     *
 *    Gera um Numero Aleatorio de 0 ate Max.          *
 *                                                    *
 *    @param Max Maximo Numero Aleatorio.             *
 *    @return Numero Aleatorio                        *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
int myRand(int Max) {
  return rand() % Max;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao isNumero                                     *
 *    Verifica se uma String eh um Numero.             *
 *                                                     *
 *    @param string String a ser Verificada.           *
 *    @return       Se eh um Numero ou Nao.            *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int isNumero(char *string) {
  int i;
  for (i=0; i<strlen(string); i++) if (!isdigit(string[i])) return 0;
  return 1;
}
```

A.2 Biblioteca de Programação Paralela

Arquivo Sockets.h.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                 *
 * --------- --------                                 *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                           *
 *                                                    *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>         *
 *            http://www.dcc.ufmg.br/~horacio         *
 *                                                    *
 *    Principal Referência: Quinn (1994 - 10.5.3)     *
 *                                                    *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#ifndef _SOCKETS_H_
#define _SOCKETS_H_ "@(#)Sockets.h $Revisao 0.1"

#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "Module.h"

// Constantes
#define MAXMSG 1024

// Argumentos do Thread do Socket
struct strSchArgs{
  int conexao;                   // Ponteiro para a Conexao
  int socket;                    // Ponteiro para o Socket Principal
};

// Definicao das Funcoes
int  abrePorta(unsigned short numPorta);
int  esperaConexao(int Socket);
int  socketPuts(int Socket, char *String);
int  socketGets(int Socket, char *String);
int  socketGetLinha(int Socket, char *String);
void gerenciaConexoes(unsigned short numPorta, int funcao);
int  socketClient(char *NomeHost, unsigned short numPorta);
```

```
int  socketPutsArq(int Socket, char *ArquivoBin);
int  socketPutBin( char* buf, unsigned int sz, int clientSock );
extern void *funcaoFilho(void *voidargs);
extern void *funcaoFilhoQsp(void *voidargs);
int  conecta(char *nomeHost, unsigned short numPorta);
int  desconecta(int socket);
void SendVector(int sd, int size, unsigned long int* vector);
void ReceiveVector(int sd, int* numelem, unsigned long int* vector);

#endif /* _SOCKETS_H_ */
```

Arquivo Sockets.c.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                  *
 * --------- --------                                  *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                            *
 *                                                     *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>          *
 *            http://www.dcc.ufmg.br/~horacio          *
 *                                                     *
 *    Principal Referência: Quinn (1994 - 10.5.3)      *
 *                                                     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Sockets.h"
#include "Module.h"

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao abrePorta.                                    *
 *    Funcao utilizada para abrir a porta. numPorta  e *
 *    o numero da porta a ser aberta.                  *
 *                                                     *
 *    @param numPorta Numero da porta a ser aberta.    *
 *    @result Descritor de Arquivo do Socket Aberto.   *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int abrePorta(unsigned short numPorta) {
  int    Socket;                 // Ponteiro para o Socket
  struct sockaddr_in EndSocket;   // Endereco de Conecao

  // Salva as informacoes na Estrutura
  EndSocket.sin_family = AF_INET;
  EndSocket.sin_port = htons(numPorta);
  EndSocket.sin_addr.s_addr = INADDR_ANY;
  // Abre um Socket TCP (Internet stream socket)
  if ((Socket = socket(AF_INET, SOCK_STREAM, 0)) < 0) return (-1);
  // Liga o nosso enderesso local para que outros possam se conectar
  if (bind(Socket,(struct sockaddr *)&EndSocket,sizeof(struct sockaddr_in))<0) {
    close(Socket);
    return (-1);
  }
  // Manda escutar na porta
  listen(Socket, 3);

  // Retorna o Socket
  return(Socket);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao esperaConexao.                                *
 *    Espera ate que uma conexao seja  requisitada  em *
 *    um determinado Socket.                           *
 *                                                     *
 *    @param socket Descritor de Arquivo do Socket.    *
 *    @result Ponteiro para a nova Conexao.            *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int esperaConexao(int socket) {
  int conexao;                   // Ponteiro para a nova Conexao
```

```
  // Fica esperando a Conexao
  if ((conexao = accept(socket,NULL,NULL)) < 0) return (-1);
  // Retorna a Conexao
  return(conexao);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao socketPuts                                  *
 *    Escreve dados no socket.                        *
 *                                                    *
 *    @param socket Descritor de Arquivo do Socket.   *
 *    @param string Conteudo do Dado a ser enviado.   *
 *    @result Quantidade de Bytes enviados.           *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
int socketPuts(int socket, char *string) {
  size_t tamanho  = strlen(string);    // Quantidade de Bytes a enviar
  size_t enviados = 0;                 // Quantidade de Bytes enviados
  int this_write;                      // Tamanho temporario

  // Envia os dados
  while (enviados < tamanho) {
    do this_write = write(socket, string, tamanho - enviados);
    while ( (this_write < 0) && (errno == EINTR) );
    // Verifica se esta tudo certo
    if (this_write <= 0) return this_write;
    // Atualiza variaveis
    enviados += this_write;
    string += this_write;
  }

  // Retorna a quantidade de bytes enviados
  return tamanho;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao socketPutsArq                               *
 *    Escreve dados binarios no socket.               *
 *                                                    *
 *    @param socket Descritor de Arquivo do Socket.   *
 *    @param arquivoBin Arquivo Binario a ser enviado. *
 *    @result Quantidade de Bytes enviados.           *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
int socketPutsArq(int socket, char *arquivoBin) {
  FILE *descArq;                  // Descritor do Arquiovo Binario de entrada
  char carac;                     // Caractere atual

  if ((descArq=fopen(arquivoBin, "rb")) == NULL) {
    printf("Nao peguei arquivo...\n");
    return 0;
  }
  //sendfile(socket,  descArq, 0, 1024);
  carac = getc(descArq);
  printf("Primeiro Caracter e: %c\nOs Proximos sao: ", carac);
  while (carac != EOF) {
    //write(socket, carac, 1);
    printf("%c", carac);
    carac = getc(descArq);
  }

  fclose(descArq);
  return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao socketGets                                  *
 *    Recebe dados do socket.                         *
 *                                                    *
 *    @param socket Descritor de Arquivo do Socket.   *
 *    @param string Conteudo do dado recebido.        *
 *    @result Quantidade de Bytes recebidos.          *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
int socketGets(int socket, char *string)
```

```
{
  int lidos;                     // Caracteres Lidos
  unsigned int total = 0;        // Tamanho Total
  char *posicaoAtual;            // Posicao Atual
  char ultimoLido = 0;           // Ultimo caractere Lido
  size_t tamanho = MAXMSG;       // Tamanho Maximo da Mensagem

  posicaoAtual = string;
  // Vai lendo os caracteres
  while (ultimoLido != 10) {
    lidos = read(socket, &ultimoLido, 1);
    // Verifica se nao deu erro ..
    if (lidos <= 0) {
      printf("[socketGets] Erro de Socket !!\n");
      fflush(NULL);
      return -1; /* Parece qua nao funciona fora do Unix !! !! */
    }
    // Coloca o caractere na String
    if ((total<tamanho) && (ultimoLido!=10) && (ultimoLido!=13)) {
      posicaoAtual[0] = ultimoLido;
      posicaoAtual++;
      total++;
    }
  }
  // Termina a String
  if (tamanho>0) posicaoAtual[0] = 0;

  // Retorna o numero de caracteres lidos
  return total;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao socketGetLinha                               *
 *  Recebe várias linhas. Como? No estilo do FTP.      *
 *  Pega o código da primeira linha e fica             *
 *  recebendo ate que o mesmo codigo seja recebido     *
 *  (indicando a ultima linha).                        *
 *  OBS: Nao execute esta funcao dentro de um loop     *
 *  dele mesmo. (static)                               *
 *  OBS2: Se o ID final nao aparecer, vai ficar        *
 *  esperando infinitamente. Use com cuidado           *
 *  OBS3: Ignora a primeira linha...                   *
 *                                                     *
 *  Problemas: horacio@dcc.ufmg.br                     *
 *                                                     *
 *    @param socket Descritor de Arquivo do Socket.    *
 *    @param string Conteudo do dado recebido.         *
 *    @result Sucesso ou nao.                          *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int socketGetLinha(int socket, char *string) {
  static int numId;
  int numIdTemp=0;

  if (!numId) {
    socketGets(socket, string);
    sscanf(string, "%d - ", &numId);
    if (!numId) return 0;
  }

  socketGets(socket, string);
  sscanf(string, "%d - ", &numIdTemp);

  if (numIdTemp==numId) {
    numId = 0;
    return 0;
  }

  else return 1;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao gerenciaConexoes                             *
```

```
 *    Cria o socket e gera um processo filho  a   cada *
 *    conexao  requisitada.  No  processo  filho,  a *
 *    funcao funcaoFilho() eh executada.              *
 *                                                    *
 *    @param numPorta Numero da Porta a ser Aberta.   *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
void gerenciaConexoes(unsigned short numPorta, int funcao) {
  int socket, conexao;                          // Ponteiros para o Socket e a Conexao
  struct sigaction sa;                          // Gernecia das Mensagens
  pthread_t proc[MAX_HOSTS*MAX_HOSTS];          // Lista de Threads
  struct strSchArgs schArgs[MAX_HOSTS*MAX_HOSTS]; // Argumentos dos Threads
  int procId = 0;                               // Numero do ultimo Thread

  // Informacoes de Mensagem
  sa.sa_handler = SIG_IGN;
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  sigaction(SIGCHLD, &sa, NULL);

  // Abre a Porta
  if ((socket = abrePorta(numPorta)) < 0) {
    exit(1);
  }
  // Fica experando pelas Conexoes
  for (;;) {
    fflush(NULL);
    // Espera por uma conexao
    if ((conexao = esperaConexao(socket)) < 0) {
      if (errno == EINTR) continue;
      exit(1);
    }

    if (DEBUG>4) printf("[gerenciaConecoes ]Conexao Recebida .. Vou abrir tread ..\n");
    // Seta os argumentos do Thread
    schArgs[procId].conexao = conexao;
    schArgs[procId].socket = socket;
    // Cria o Thread
    if (funcao==1) pthread_create(&proc[procId], NULL,funcaoFilhoQsp,&schArgs[procId]);
    else pthread_create(&proc[procId], NULL,funcaoFilho,&schArgs[procId]);
    if (DEBUG>4) printf("[gerenciaConexoes] Tread %d Aberto .. Vou continuar .. \n", procId);
    procId++;
    fflush(NULL);
  }
}

/** * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao socketClient                                *
 *    Cliente do Socket.                              *
 *                                                    *
 *    @param nomeHost Nome do host Master a conectar. *
 *    @param numPorta Numero da porta do Master.      *
 *    @return Ponteiro para o Socket.                 *
 ** * * * * * * * * * * * * * * * * * * * * * * * * */
int socketClient(char *nomeHost, unsigned short numPorta) {
  int    Socket;                 // Socket da Conexao Criada
  struct sockaddr_in endSocket;  // Estrutura do Socket
  struct hostent *host;          // Host Master a se Conectar

  if ((host = gethostbyname(nomeHost)) == NULL) return (-1);

  memset(&endSocket, 0, sizeof(endSocket));
  memcpy((char *) &endSocket.sin_addr, host->h_addr, host->h_length);

  endSocket.sin_family = host->h_addrtype;
  endSocket.sin_port = htons((u_short) numPorta);

  if ((Socket = socket(host->h_addrtype, SOCK_STREAM, 0)) < 0) return (-1);

  if (connect(Socket,(struct sockaddr *)&endSocket,sizeof endSocket)<0) {
    close(Socket);
    return(-1);
  }
```

```
  // Retorna o Socket
  return(Socket);
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao socketPutBin                                  *
 *    Envia um buffer binario para a outra ponta.       *
 *                                                      *
 *    @param buf Buffer binario a ser enviado.          *
 *    @param sz Tamanho em Bytes a ser enviado.         *
 *    @clientSock Ponteiro do Socket.                   *
 *    @return Bytes enviados.                           *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int socketPutBin( char* buf, unsigned int sz, int clientSock ) {
    int ret;
    ret = -1;
    if( buf && sz ) {
        unsigned int enviado = 0;
        int t;
        while( enviado < sz ) {
            t = send( clientSock, buf+enviado, sz-enviado, 0 );
            if( t < 0 ) {
                break;
           }
            enviado += t;
        }
        if (enviado) ret = enviado;
    }
    return ret;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao conecta                                       *
 *    Conecta em um Master.                             *
 *                                                      *
 *    @param nomeHost Nome do Host a se Conectar.       *
 *    @param numPorta Numero da porta do Master.        *
 *    @return Socket Criado.                            *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int conecta(char *nomeHost, unsigned short numPorta) {
  int socket;
  char resposta[MAXMSG];

  socket = socketClient(nomeHost, numPorta);
  if (socket <= 0) return 0;
  socketGets(socket, resposta);

  return socket;
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao desconecta                                    *
 *    Desconecta em um Master.                          *
 *                                                      *
 *    @param socket Ponteiro do Socket a desconectar.   *
 *    @return Se desconectou ou nao.                    *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int desconecta(int socket) {
  char resposta[MAXMSG];
  socketPuts(socket, "QUIT\n");
  socketGets(socket, resposta);
  close(socket);
  return 1;
}

void SendVector(int sd, int size, unsigned long int* vector) {
  long int sizeVector = sizeof(unsigned long int) * size;
  long int* buf = (unsigned long int*) malloc (sizeVector);
  long int i;

  if (DEBUG>13) printf("[sendVector] Enviando %d numeros ..\n", size);
```

```
  /* envia o tamanho do vetor */
  send(sd, (int*)&size, sizeof(unsigned long int),MSG_NOSIGNAL);

  if (DEBUG>13) printf("[sendVector]     Enviou quantidade de numeros ..\n");

  /* envia o numero de linhas desejados da matriz */
  bzero(buf, sizeVector);
  memcpy(buf, vector, sizeVector);
  i = send(sd, buf, sizeVector, MSG_NOSIGNAL);

  if (DEBUG>13) printf("[sendVector]      -  Lista enviada com %ld bytes (era para ser: %ld)..\n", i, sizeVector);

  free(buf);
}
/* fim de envia_matriz */


/* Função recebe_matriz. Recebe como entrada o número do identificador do
socket, variáveis para receber o número de linhas e colunas e a matriz
de dados. Recebe uma mensagem contendo a ordem (linhas e colunas) da
matriz e em seguida recebe um bloco de linhas da matriz, sendo recebido
linha a linha.
*/
void ReceiveVector(int sd, int* numelem, unsigned long int* vector)
{
    int sizevector;
    unsigned long int*  buf;
    int rc, tamanho;

    recv(sd,(int *)&tamanho, sizeof(unsigned long int), MSG_WAITALL|MSG_NOSIGNAL);
    *numelem = tamanho;

    sizevector = tamanho * sizeof(unsigned long int);
    buf = (unsigned long int*) malloc(sizevector);

    /* envia o numero de linhas desejados da matriz */
    bzero(buf, sizevector);
    rc = recv(sd, buf, sizevector, MSG_WAITALL|MSG_NOSIGNAL);
    memcpy(vector, buf, sizevector);
    if (DEBUG>3) printf("[sendVector]        -   Recebi %d bytes .. Era para ser %d ..\n", rc, sizevector);
    fflush(NULL);
    if( rc< 0)
    {
  perror("Dados recebidos");
  close(sd);
    }
    free(buf);
}/* fim de recebe_matriz */
```

A.3 Biblioteca de Cálculo do Tempo

Arquivo `Timer.h`.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                 *
 * --------- --------                                 *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                           *
 *                                                    *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>         *
 *            http://www.dcc.ufmg.br/~horacio         *
 *                                                    *
 *    Principal Referência: Quinn (1994 - 10.5.3)     *
 *                                                    *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <time.h>
```

```
#include <sys/time.h>
#include <sys/types.h>

clock_t inicioClock, ultimoClock, overheadClock; // Tempos de Execucao

struct timeval inicioTime, ultimoTime, tempoDecTime; // Tempos de Execucao


void startTimer();
void printTimer();

void startCTimer();
void printCTimer();
int getCMilesimos(long clock);
int getCSegundos(long clock);
```

Arquivo Timer.c.

```
/** * * * * * * * * * * * * * * * * * * * * * * * * * *
 * QuickSort Paralelo                                  *
 * --------- --------                                  *
 *    Ordenacao de Conjuntos  utilizando  Programacao *
 *    Paralela e QuickSort.                            *
 *                                                     *
 *    Horácio Fernandes <horacio@dcc.ufmg.br>          *
 *            http://www.dcc.ufmg.br/~horacio          *
 *                                                     *
 *    Principal Referência: Quinn (1994 - 10.5.3)      *
 *                                                     *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "Module.h"
#include "Timer.h"

void startTimer() {
  gettimeofday(&inicioTime, NULL);
  gettimeofday(&ultimoTime, NULL);
}

void printTimer() {
  struct timeval agora;
  gettimeofday(&agora, NULL);
  timevalSubtract(&tempoDecTime, &agora, &ultimoTime);
  gettimeofday(&ultimoTime, NULL);
  if (DEBUG>=0) printf("[print Time]        --> Tempo Ultimo: %d.%06d Segundos\n",\
     (int) tempoDecTime.tv_sec, (int) tempoDecTime.tv_usec);
  timevalSubtract(&tempoDecTime, &agora, &inicioTime);
  if (DEBUG>=0) printf("[print Time]        --> Tempo Atual:  %d.%06d Segundos\n", \
     (int) tempoDecTime.tv_sec, (int) tempoDecTime.tv_usec);
}


void startCTimer(void) {

  inicioClock   = clock();
  overheadClock = clock() - inicioClock;
  inicioClock   = clock();
  ultimoClock   = clock();
}


void printCTimer() {
  clock_t agora = clock();
  if (DEBUG>=0) printf("[print Time]        --> Tempo Ultimo: %d Segundos e %d Milesimos\n", \
    getCSegundos(agora - ultimoClock), getCMilesimos(agora - ultimoClock));
  if (DEBUG>=0) printf("[print Time]        --> Tempo Atual:  %d Segundos e %d Milesimos\n", \
    getCSegundos(agora - inicioClock), getCMilesimos(agora - inicioClock));
  ultimoClock = clock();
}
```

```
int getCMilesimos(long clock) {
  return( (int) (((clock%CLOCKS_PER_SEC)*1000)/(CLOCKS_PER_SEC)) );
}

int getCSegundos(long clock) {
  return((int)(clock/CLOCKS_PER_SEC));
}

/** * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Funcao timevalSubtract                                *
 *    Subtrai  os  valores  struct  timeval  X   e   Y *
 *    armazenando o resultado em  RESULT. Retorna 1 se *
 *    a diferenca e negativa, 0 caso contrario.        *
 *                                                      *
 *    @param result Ponteiro onde ficara o resultado.  *
 *    @param x Tempo 1.                                 *
 *    @param y Tempo 2.                                 *
 ** * * * * * * * * * * * * * * * * * * * * * * * * * */
int timevalSubtract (result, x2, y2)
     struct timeval *result, *x2, *y2;
{

  struct timeval x, y;

  // Copia os valores
  x.tv_sec  = x2->tv_sec;
  x.tv_usec = x2->tv_usec;
  y.tv_sec  = y2->tv_sec;
  y.tv_usec = y2->tv_usec;

  /* Perform the carry for the later subtraction by updating y. */
  if (x.tv_usec < y.tv_usec) {
    int nsec = (y.tv_usec - x.tv_usec) / 1000000 + 1;
    y.tv_usec -= 1000000 * nsec;
    y.tv_sec += nsec;
  }
  if (x.tv_usec - y.tv_usec > 1000000) {
    int nsec = (y.tv_usec - x.tv_usec) / 1000000;
    y.tv_usec += 1000000 * nsec;
    y.tv_sec -= nsec;
  }

  /* Compute the time remaining to wait.
     tv_usec is certainly positive. */
  result->tv_sec = x.tv_sec - y.tv_sec;
  result->tv_usec = x.tv_usec - y.tv_usec;

  /* Return 1 if result is negative. */
  return x.tv_sec < y.tv_sec;
}
```

A.4 Scripts de Auxilio aos Testes

Arquivo `xSlaves.sh`.

```
#!/bin/bash

ArqUltimaPorta="/tmp/QSP_UltimaPorta.tmp"
ArqHosts="HostsClientes"
NumClientes=3

QSP="PaaTP4.exe"

if [ -f $ArqUltimaPorta ] ; then
    UltimaPorta=`cat $ArqUltimaPorta`
else
    UltimaPorta=2303
```

```
fi


killall $QSP

rm -f $ArqHosts

for (( i=1 ; i<=$NumClientes ; i=$[ $i + 1 ] )) ; do

    PortaAtual=$[ $UltimaPorta + $i ]

    echo "127.0.0.1 $PortaAtual /tmp/Horacio/dados25MB.bin" >> $ArqHosts

    nohup $QSP -c $PortaAtual &

done

echo $PortaAtual > $ArqUltimaPorta

echo ""
echo "Ultima porta: $PortaAtual"
echo ""

if [ "$1" == "1" ] ; then
telnet localhost $[ $UltimaPorta + 1 ]
fi
```

Arquivo GerarDados.sh.

```
#!/bin/bash

DIREntrada="Logs/"
DIRSaida="Dados/"

NumTestes=12

echo "[GeraDados] Gerando Dados .."

for (( i=1 ; i<=$NumTestes ; i=$[ $i + 1 ] )) ; do

    ArquivoAtual="$DIREntrada/SaidaTeste$i.log"
    ArquivoAtualSaida="$DIRSaida/SaidaTeste$i.dat"

    NomeArqs="$NomeArqs $ArquivoAtualSaida"

    echo "[GeraDados]     Analisando Arquivo $ArquivoAtual .."

    cat $ArquivoAtual | grep "Tempo Ultimo" | cut -d" " -f 13 > $ArquivoAtualSaida

    TempoTotal=`tail -1 $ArquivoAtual | cut -d" " -f14`
    echo $TempoTotal >> $ArquivoAtualSaida

done

echo "[GeraDados]     Unindo dados .. "

paste -d" " $NomeArqs | sed -e 's\  \ 0.000000 \g' > $DIRSaida/TodosDados.dat
```