Universidade de Coimbra - Faculdade de Ciências e Tecnologia
Departamento de Matemática

COMPUTAÇÃO PARALELA - 2005/2006
4º ANO

PROBLEMA 1 – *DETERMINAÇÃO DOS NÚMEROS PRIMOS*

*** RELATÓRIO ***

Sara Catarina Morgado Menoita – ma0308@mat.uc.pt
Sara Joana Fino dos Santos Rodrigues de Carvalho – tc01037@mat.uc.pt
Sara Margarida Gaspar da Silva – tc01038@mat.uc.pt

Coimbra, 7 Novembro 2005

" O degrau de uma escada não serve simplesmente para que alguém permaneça em cima dele, destina-se a sustentar o pé de um homem pelo tempo suficiente para que ele coloque o outro um pouco mais alto"
(Thomas Huxley)

Índice

1. Problema Proposto

Computação Paralela 2005/2006

Problema 1 - determinação dos números primos

Data Limite de entrega: 31 Outubro 2005

Algoritmo:

Seja S(k) o conjunto dos números primos inferiores a k. Podemos verificar se os números n, tal que $k \leq n < k^2$, são primos testando se têm ou não divisores em S(k).

Problema:

1. Considere S(16) = {2, 3, 5, 7, 11, 13}. Escrever um programa que lhe permita determinar sucessivamente os números primos até $n_{\max}$.

2. Discuta as estratégias de paralelização do algoritmo acima descrito (sugestão: consulte o livro L. R. Scott, T. Clark, B. Bagheri, Scientific Parallel Computing) e escreva um programa que, com recurso ao MPI, seja executado em várias máquinas.

3. Definindo o *speedup* como

   $$S_P = \frac{T_1}{Tp},$$

   em que T_1 é o tempo que demora a executar o algoritmo no código série e T_P o tempo de execução do código paralelo para P máquinas. Estude a dependência de S_P com a dimensão do problema ($n_{\max}$).

2. Introdução

A busca de números primos não é recente, entretanto, apenas quando se iniciou o processo de busca computacional, os números foram-se tornando maiores de uma forma exponencialmente surpreendente. A computação paralela nesta área é fundamental para que se possa agilizar cada vez mais a busca destes números já que, actualmente, se precisa de muito processamento para os cálculos exigidos no teste de primalidade, pois os maiores primos já encontrados passam da casa dos milhões de dígitos.

O objectivo deste trabalho é a implementação de um algoritmo paralelo que realiza o cálculo computacional de números primos de uma forma rápida e eficiente. Este algoritmo foi desenvolvido para realizar o processamento paralelo.

3. Crivo de Eratosthenes

A procura de números primos é um exemplo interessante onde se podem usar estratégias de paralelização. Recordemos duas definições importantes para este estudo:

* Divisor: um inteiro j é um divisor de outro inteiro p se p/j é ainda um inteiro;
* Primo: um número primo é um número inteiro que tem exactamente dois divisores: ele mesmo e 1.

O Crivo de Eratosthenes é o algoritmo mais antigo, e talvez o mais simples e rápido de implementar para encontrar números primos. Ele foi criado por Eratosthenes, um matemático grego do mundo antigo.

3.1 Algoritmo

O algoritmo do Crivo de Eratosthenes possui um único argumento k, que é um número inteiro. Ao final de sua execução, uma lista com todos os números primos inferiores ou igual a k é retornada.

Sequencialmente, escreve-se todos os inteiros de 2 até k. Risca-se então todos os números maiores que 2 que são divisíveis por 2. No próximo passo, selecciona-se (de entre os números não riscados) o menor número que é superior a 2. Neste caso, esse número é o 3. Assim, risca-se todos os números maiores que 3 que são por ele divisíveis.

O processo então recomeça, desta vez o menor número maior que 3 e não riscado é o 5. Portanto, risca-se todos os inteiros maiores que 5 que são por ele divisíveis. Continua-se a fazer este procedimento até que todos os números divisíveis por $\sqrt{k}$ tenham sido riscados. Os números restantes que não foram riscados são todos primos: 2, 3, 5, 7, 11, 13… Na Fig.1 é exemplificado a execução para k=16, e portanto riscam-se números até os divisíveis por $\sqrt{16} = 4$.

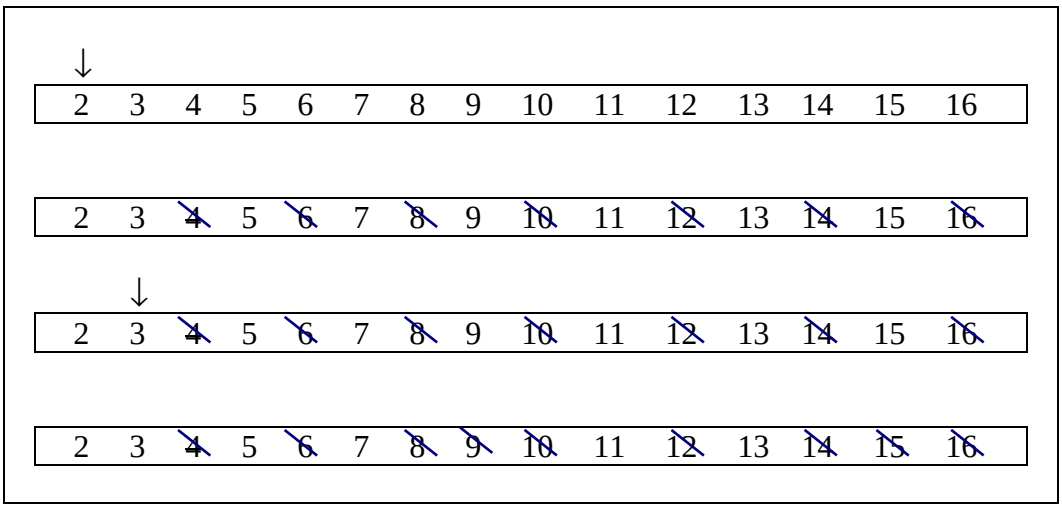


Fig.1 – Execução do Crivo de Eratosthenes para k = 16

4. Algoritmo serial do problema 1

- Inicializar o MPI;

→ RANK=0 (visto que estamos a elaborar um programa serial)

- Pedir ao usuário do programa o valor de k;
- Inicializar um vector vi com os elementos de 2 a k;
- Aplicar o crivo de Eratosthenes ao vector vi:

Fixando o primeiro elemento do vector vi (que é 2 qualquer que seja k) vamos colocar 1's em todos os seus múltiplos. De seguida procuramos o próximo elemento de vi diferente de 1 (que é 3 qualquer que seja k) e colocamos 1's em todos os múltiplos deste. E assim sucessivamente até que todos os números divisíveis por sqrt(k) (convertido em inteiro) tenham sido substituídos por 1's.

- Construir S(k) (armazenando os seus elementos num vector vp) a partir do vector vi, "ignorando" os 1's e "aproveitando" apenas os números primos;
- Criar um vector cujo primeiro elemento é k, o segundo k+1,….., até k^2 -1;
- Aplicar o crivo de Eratosthenes ao vector acima mencionado usando o S(k);
- Imprimir no ecrã os números primos entre k a k^2 ;
- Finalizar.

NOTA: O código do programa serial encontra-se em anexo com o nome ***projecto_serial.c***

5. Algoritmo paralelo do problema 1

5.1. Estratégias de paralelização

Para paralelizar o algoritmo serial decidimos pegar no vector que é constituído por números de k a k^2 -1 e dividir a sua dimensão pelo número de processos. Assim cada processo vai ser responsável pela aplicação do crivo de Eratosthenes ao segmento do vector que lhe corresponde.

Exemplo: k = 4 , número de processos = 4

Vector que é constituído por números de k a k^2 -1:

4	5	6	7	8	9	10	11	12	13	14	15

Este vector tem dimensão $(4^2 - 4) = 12$.

Vamos então dividir o vector em subvectores de modo a que cada subvector seja posteriormente enviado para o processo correspondente. Para isso, vamos dividir o vector em blocos de (12/4) = 3 elementos:

4	5	6

↓
processo 1
rank = 0

7	8	9

↓
processo 2
rank = 1

10	11	12

↓
processo 3
rank = 2

13	14	15

↓
processo 4
rank = 3

Cada processo efectua o crivo de Eratosthenes substituindo os números não primos por 1's .
O resultado será o seguinte:

1	5	1

7	1	1

1	11	1

13	1	1

Colectamos os subvectores no rank = 0 e temos como resultado o seguinte vector:

1	5	1	7	1	1	1	11	1	13	1	1

Construímos o vector final a partir do vector anterior "ignorando" os 1's e "aproveitando" apenas os números primos (também esta operação é feita no rank = 0):

5	7	11	13

Os elementos deste vector constituem o conjunto solução do nosso problema!!!

Conclusão: Vamos utilizar o paralelismo de dados, isto é, cada processador aplica a mesma operação (marcar múltiplos de um primo) à sua porção do conjunto de dados.
Nota: Apesar te termos enveredado por esta estratégia não implica que esta seja única.

5.2. Algoritmo

- Inicializar o MPI;

→ RANK=0 :

- Pedir ao usuário do programa o valor de k;
- Inicializar um vector vi com os elementos de 2 a k;
- Aplicar o crivo de Eratosthenes ao vector vi:

Fixando o primeiro elemento do vector vi (que é 2 qualquer que seja k) vamos colocar 1's em todos os seus múltiplos. De seguida procuramos o próximo elemento de vi diferente de 1 (que é 3 qualquer que seja k) e colocamos 1's em todos os múltiplos deste. E assim sucessivamente até que todos os números divisíveis por sqrt(k) (convertido em inteiro) tenham sido substituídos por 1's.

- Construir S(k) (armazenando os seus elementos num vector vp) a partir do vector vi, "ignorando" os 1's e "aproveitando" apenas os números primos;
- Criar um vector vaux cujo primeiro elemento é k, o segundo k+1,….., até k^2 -1;
- Criar uma variável d que vai ser o resultado da divisão da dimensão do vector vaux pelo número de processos:

se este número for inteiro então d:= (k^2 -k) / nproc;
senão resto := (k^2 -k) % nproc
d:= ((k^2 -k) + (nproc – resto)) / nproc
acrescentar a vaux (nproc – resto) 1's ;

→ TODOS OS RANK'S

- Enviar a dimensão do vector vp para todos os processos;
- Enviar o vector vp para todos os processos;
- Enviar o valor d para todos os processos;
- Enviar subvectores de vaux com d elementos para todos os processos;
- Em cada processo aplicar o crivo de Eratosthenes ao subvector de vaux recebido;
- Colectar todos os subvectores de vaux;
- Construir um vector vk a partir do vector vaux, "ignorando" os 1's e "aproveitando" apenas os números primos;
- Imprimir no ecrã os números primos entre k a k^2 -1 (os elementos do vector vk);
- Finalizar.

NOTA: O código do programa paralelo encontra-se em anexo com o nome ***projecto_paralelo.c***

OBSERVAÇÕES:

- Após a análise do algoritmo, optámos pelo uso de rotinas de comunicação colectivas (MPI_Scatter, MPI_Gather) em detrimento das rotinas de comunicação ponto-a-ponto (MPI_Send, MPI_Recv) visto que torna a resolução do problema mais simples (uma vez que diminui o número de rotinas MPI utilizadas) e o programa fica visualmente mais compreensível (diminui também o número de ciclos).
- Vamos também ter em conta que para trabalhar com valores de k grandes iremos ter como resultados números primos muito elevados e portanto optámos por usar variáveis do tipo *unsigned long int*, visto que este tipo tem uma maior capacidade de memória e portanto maior número de dígitos inteiros sem sinal.

6. MPI

Um programa MPI consiste de um conjunto de processos que cooperam (comunicam), utilizando as rotinas de comunicação MPI, para resolver um problema. Em geral desenvolve-se um único programa fonte, que quando compilado gera um único programa executável que é executado por todos os processos da aplicação. Então, utilizando a estrutura condicional if e o rank dos processos define-se cada o processo.

A maioria das rotinas do MPI exigem que seja especificado um "communicator" como argumento. MPI_COMM_WORLD é o comunicador predefinido, que inclui todos os processos, definido pelo usuário.

Existem os seguintes tipos de comunicação entre os processos:

Ponto a Ponto – São **a**s rotinas de comunicação ponto a ponto que executam a transferência de dados entre dois processos (Exemplo: MPI_Send , MPI_Recv).

Colectiva - É a comunicação padrão que invoca todos os processos num grupo - colecção de processos que podem comunicar entre si. Normalmente a comunicação colectiva envolve mais de dois processos. As rotinas de comunicação colectiva são voltadas para a comunicação/coordenação de grupos de processos (Exemplo: MPI_Bcast, MPI_Scatter, MPI_Gather).

6.1. Rotinas básicas utilizadas

6.1.1. MPI_Init

Inicia um processo MPI:

Definição: Inicializa um processo MPI. Portanto, deve ser a primeira rotina a ser chamada por cada processo, pois estabelece o ambiente necessário para executar o MPI. Ela também sincroniza todos os processos na inicialização de uma aplicaçãoMPI.

Sintaxe em C:
int MPI_Init (int *argc, char *argv[])

Parâmetros:
argc - Apontador para a quantidade. de parâmetros da linha de comando;
argv - Apontador para um vector de strings.

6.1.2. MPI_Comm_rank

Identifica um processo no MPI:

Definição:
Identifica um processo MPI dentro de um determinado grupo. Retorna sempre um valor inteiro entre 0 e n-1, onde n é o número de processos.

Sintaxe em C:
int MPI_Comm_rank (MPI_Comm comm, int *rank)

Parâmetros:
comm - Comunicador do MPI;
rank - Variável inteira com o numero de identificação do processo.

6.1.3. MPI_Comm_size

Conta processos no MPI:

Definição:
Retorna o número de processos dentro de um grupo.

Sintaxe em C:
int MPI_Comm_size (MPI_Comm comm, int *size)

Parâmetros:
comm - Comunicador do MPI;
size - Variável interna que retorna o numero de processos iniciados pelo MPI.

6.1.4. MPI_Finalize

Encerra um processo no MPI:

Definição:
Finaliza um processo MPI. Portanto deve ser a última rotina a ser chamada por cada processo. Sincroniza todos os processos na finalização de uma aplicação MPI.

Sintaxe em C:
int MPI_Finalize ()

Parâmetros:
(Nenhum)

6.2. Mais rotinas utilizadas

6.2.1. MPI_Barrier

Sincroniza Processos num Grupo:

Definição:
Esta rotina sincroniza todos os processos de um grupo. Um processo de um grupo que utiliza MPI_Barrier interrompe a sua execução até que todos os processos do mesmo grupo executem também um MPI_Barrier.

Sintaxe em C:
int MPI_Barrier (MPI_Comm comm)

Parâmetros:
comm - Identificação do communicator;

6.2.2. MPI_Bcast

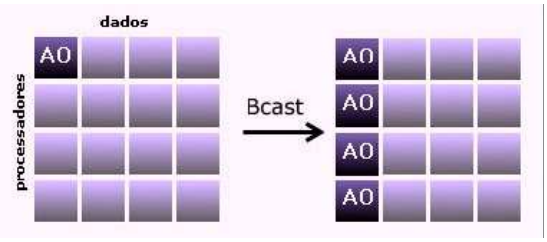
Envia dados para todos os processos:

Definição:
Diferentemente da comunicação ponto-a-ponto, na comunicação colectiva é possível enviar/receber dados simultaneamente de/para vários processos.

Sintaxe em C:
int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Parâmetros:
buffer - Endereço do dado a ser enviado;
count - Número de elementos a serem enviados;
datatype - Tipo do dado;
root - Identifica o processo que irá efetuar o broadcast (origem);
comm - Identifica o Communicator;

Ilustração:



6.2.3. MPI_Scatter

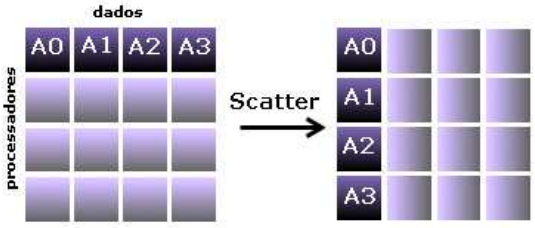
Envia mensagens colectivamente:

Definição:
É uma rotina para o envio de mensagens para um subgrupo de processos. Com a rotina MPI_Scatter a mensagem pode ser segmentada e enviada para processos diferentes.

Sintaxe em C:
int MPI_Scatter (void* sbuffer, int scount, MPI_Datatype sdatatype, void* rbuffer, int rcount,
MPI_Datatype rdatatype,int root, MPI_Comm comm)

Parâmetros:
sbuffer - Endereço dos dados a serem distribuídos;
scount - Número de elementos enviados para cada processo;
sdatatype - Tipo do dado a ser enviado;
rbuffer - Endereço onde os dados serão armazenados;
rcount - Quantidade de dados recebidos;
rdatatype - Tipo do dado recebido;
root - Identifica o processo que irá distribuir os dados;
comm - Identifica o Communicator;

Ilustração:



6.2.4. MPI_Gather

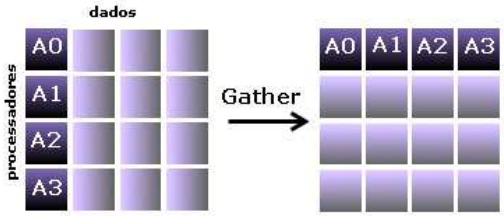
Colecta mensagens de processos:

Definição:
Rotina para coleta de mensagens de um subgrupo de processos.

Sintaxe em C:
int MPI_Gather (void* sbuffer, int scount, MPI_Datatype sdatatype, void* rbuffer, int rcount,
MPI_Datatype rdatatype,int root,MPI_Comm comm)

Parâmetros:
sbuffer - Endereço inicial do dado a ser coletado;
scount - Número de dados a serem coletados;
sdatatype - Tipo do dado a ser coletado;
rbuffer - Endereço onde os dados serão armazenados;
rcount - Número de elementos recebidos por processo;
rdatatype - Tipo do dado recebido;
root - Identifica o processo que irá efetuar a coleta;
comm - Identifica o Communicator;

Ilustração:



6.2.5. MPI_Wtime

Contabiliza o tempo decorrido:

Definição:
Retorna o número de segundos decorridos desde algum tempo no passado.

Sintaxe em C:
double MPI_Wtime ()

Parâmetros:
(Nenhum)

Observação:
A função MPI_Wtime retornará o tempo decorrido desde o início do programa. Sua precisão, pode ser obtida através da rotina MPI_Wtick.

6.2.6. MPI_Wtick

Fornece a precisão do tempo:

Definição:
Retorna (em valor real) a precisão de tempo computada pelo comando MPI_Wtime.

Sintaxe em C:
double MPI_Wtick ()

Parâmetros:
(Nenhum)

Observação:
A função MPI_Wtime retornará a precisão em múltiplos ou submúltiplos de segundo. Por exemplo: se MPI_Wtime for incrementado a cada milésimo de segundo, esta rotina retornará 0.001.

7. Análise da Performance

Este capítulo é reservado à análise de dados que recolhemos através de testes efectuados nos *tatus* do laboratório de cálculo do Departamento de Matemática da Universidade de Coimbra.

Executámos várias vezes os programas serial e paralelo de onde obtivemos diversos resultados de tempos com grandes oscilações para os mesmos testes; no entanto achámos normal, pois cada vez que nos ligávamos em rede trabalhávamos com tatus diferentes e para além disso os tatus estavam a ser utilizados por mais alunos e pelo próprio sistema o que condicionava os tempos de execução dos programas. Após constatarmos estes factos optámos por realizar os testes de forma sequencial para manter as mesmas condições em todos os testes.

7.1. Análise de gráficos

NOTA: A unidade de tempo usada em todos estes gráficos foi o segundo.

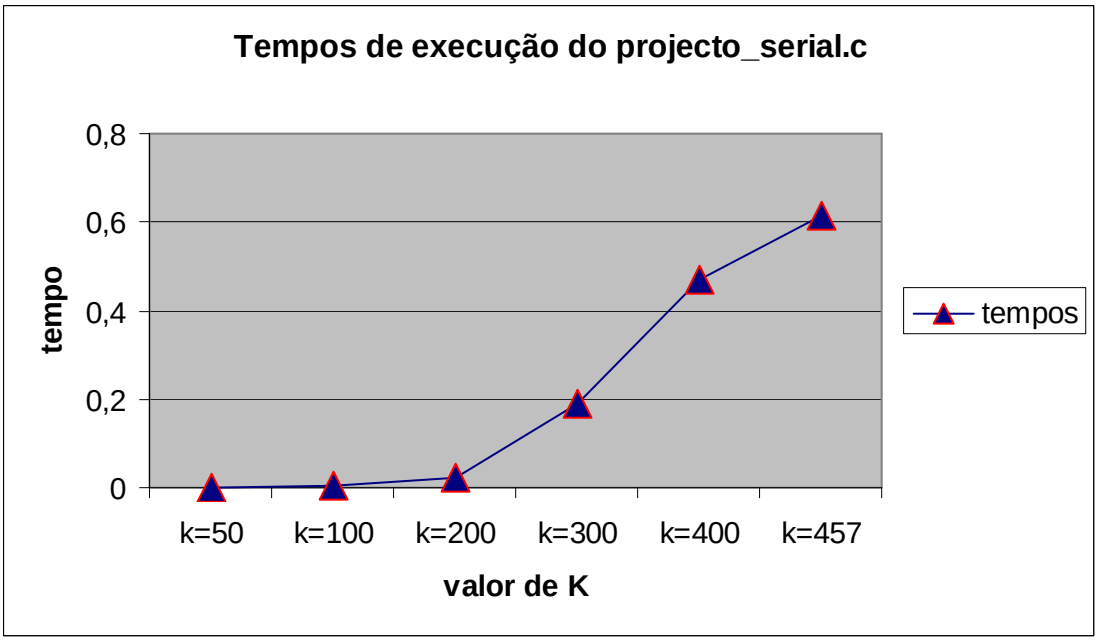


Figura 2

Este gráfico mostra-nos a relação entre o tempo de execução do programa serial e o valor de k. Como seria de esperar à medida que k aumenta o tempo de execução aumenta pois, o número de operações aumentar com k.

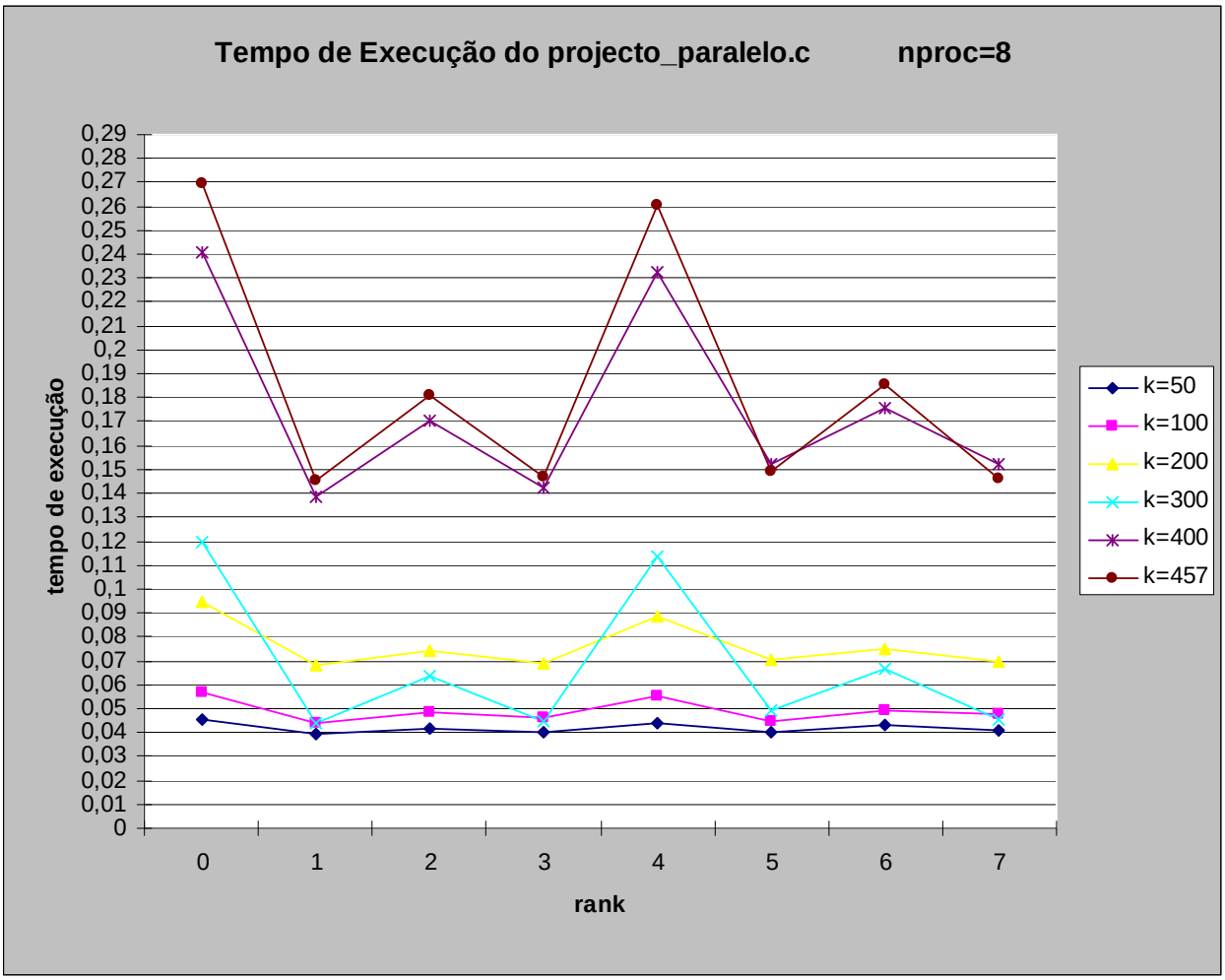


Figura 3

Através da análise deste gráfico podemos constatar que o rank 0 demora sempre mais tempo a executar o programa do que qualquer outro rank, o que não é de estranhar pois o rank 0 é o master e portanto termina a sua execução depois de fazer o seu trabalho e após todos os rank's terem também terminado.

Outra conclusão que não deixa de ser interessante, embora não a saibamos justificar, é que os rank's de ordem ímpar são mais rápidos que os de ordem par.

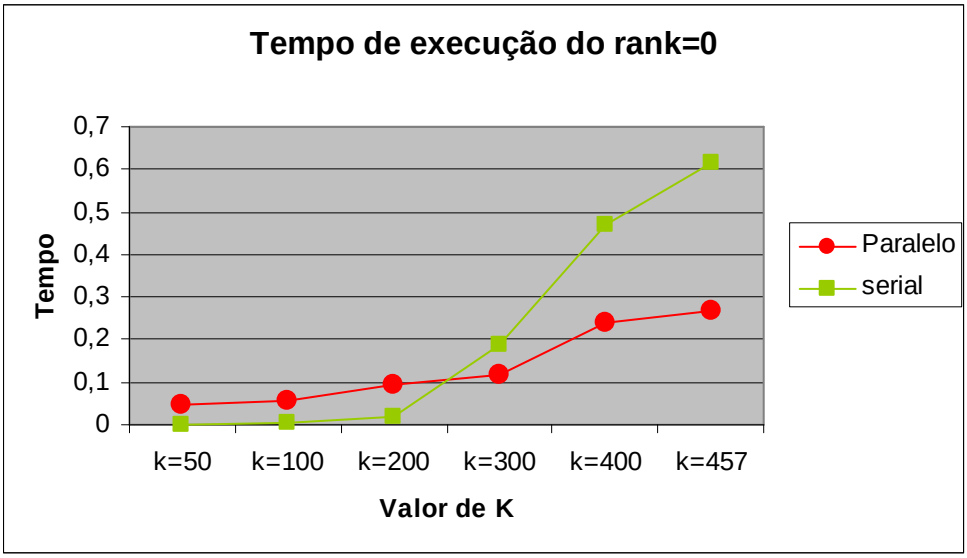


Figura 4

Analisando este gráfico podemos verificar que o tempo que o programa serial leva a executar, para valores de k pequenos, é menor que o tempo de execução do programa paralelo para o mesmo k. Estes tempos vão-se aproximando até que a para certo k o programa paralelo mostra-se mas rápido. À medida que k aumenta é ainda mais notória a rapidez de execução quando comparada com o programa serial.

7.2. Speedup

O principal objectivo na utilização da computação paralela é a diminuição do tempo de execução dos processos envolvidos. Diferentes métricas podem ser utilizadas para verificar se a utilização do processamento paralelo é vantajosa e quantificar o desempenho alcançado. O Speedup é uma das métricas mais utilizadas para atingir esse objectivo.

$$S_P = \frac{T_1}{Tp}$$

onde S_P = speedup observado num computador com P processos;

T_1 = tempo de execução do programa serial;

T_P = tempo de execução do programa paralelo correspondente com P processos.

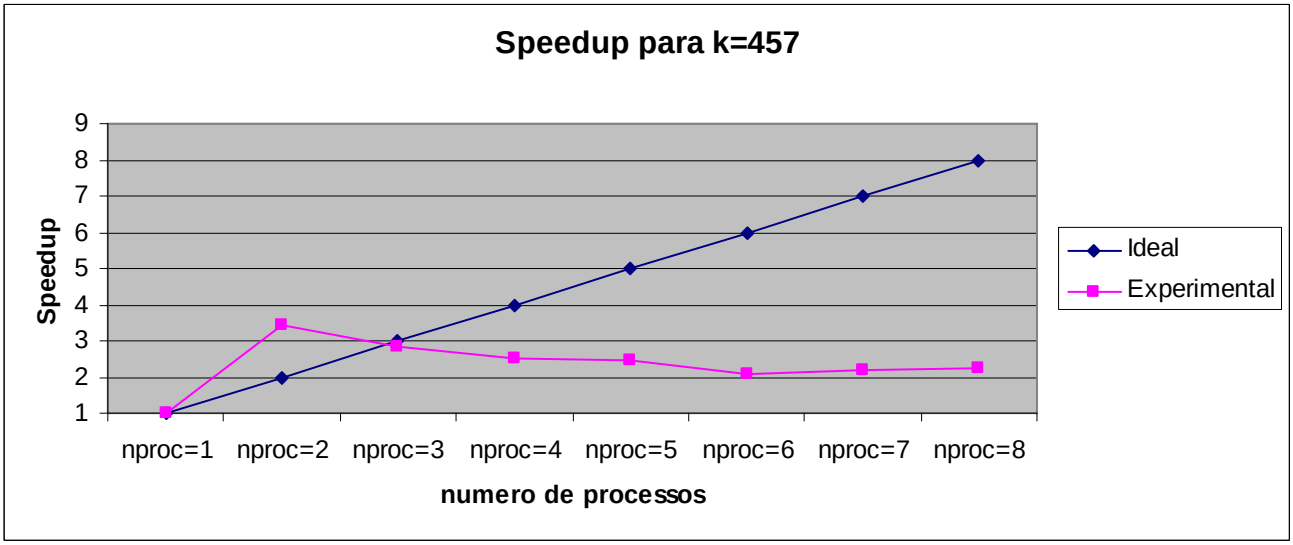


Figura 5

Neste gráfico a recta a azul refere-se à distribuição perfeita dos processos a serem executados em paralelo, ou seja, valor máximo de speedup (S_P = P). A curva a rosa refere-se ao speedup dos testes por nós efectuados.

Analisando o gráfico verificamos que o speedup é ideal quando o nº de processos é 1 e 3. Quando o nº de processos é 2 o speedup obtido é maior que o ideal, ou seja S_P > P, a esta situação anómala chamamos speedup super linear, que pode ter sido causada pelas características do algoritmo paralelo ou pelas características da plataforma de *hardware*. Para o caso em que o número de processos é 4, 5, 6, 7 e 8 o speedup alcançado é menor que o ideal. Alguns factores podem estar na origem destes valores de speedup, tais como sobrecarga de comunicação e sincronismo.

8. Limitações do programa

Temos a consciência que o programa por nós desenvolvido não é perfeito, pois apresenta uma limitação bastante significativa: apenas corre correctamente para $k \leq 457$, quando k excede este valor o programa *"crasha"* (devolve-nos valores que não são primos). Isto acontece devido ao facto do número máximo de elementos de um vector ser 209000, e através do nosso programa ser necessário criar um vector de dimensão $k^2 - k$, dimensão esta que para $k \geq 458$ excede o valor máximo da dimensão de vectores.

Esta limitação é de facto significativa visto que o último primo que o programa nos fornece é 208843 (último elemento do conjunto solução quando k=457) e tendo em conta que o maior número primo já encontrado passa da casa dos milhões de dígitos.

9. Conclusões

Com este trabalho concluímos que para valores de k pequenos, o tempo de execução do programa serial é menor que o do programa paralelo. À medida que o valor de k aumenta estes tempos vão-se aproximando até que a partir de certa ordem (a rondar k = 300, pelos dados que recolhemos e através dos quais fizemos o gráficos do capítulo7 deste relatório) o tempo de execução do programa paralelo é menor que o do serial. Esta diferença vai-se acentuando com o crescimento do k. Constata-se assim que a paralelização do nosso problema é vantajosa em termos de rapidez quando trabalhamos com valores de k suficientemente grandes.

10. Bibliografia

- L. R. Scott, T. Clark, B. Bagheri, Scientific Parallel Computing

- Páginas da web:

  http://www.di.ubi.pt/~crocker/paralel/docs/cursompi.pdf
  http://www-unix.mcs.anl.gov/mpi/
  http://atlas.ucpel.tche.br/~barbosa/sist_dist/mpi/mpi.html

ANEXOS

PROJECTO REALIZADO POR:

(Sara Catarina Morgado Menoita)

(Sara Joana Fino dos Santos Rodrigues de Carvalho)

(Sara Margarida Gaspar da Silva)