UNIVERSIDADE DE SÃO PAULO – USP
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

Trabalho 1 -
Comparação de desempenho computacional entre execução sequencial e paralela.

SSC0143 – Programação Concorrente
Prof: Júlio Cezar Estrella

Nomes:	**NºUSP:**
Raul Zaninetti Rosa	8517310
Henrique Pasquini Santos	8532252

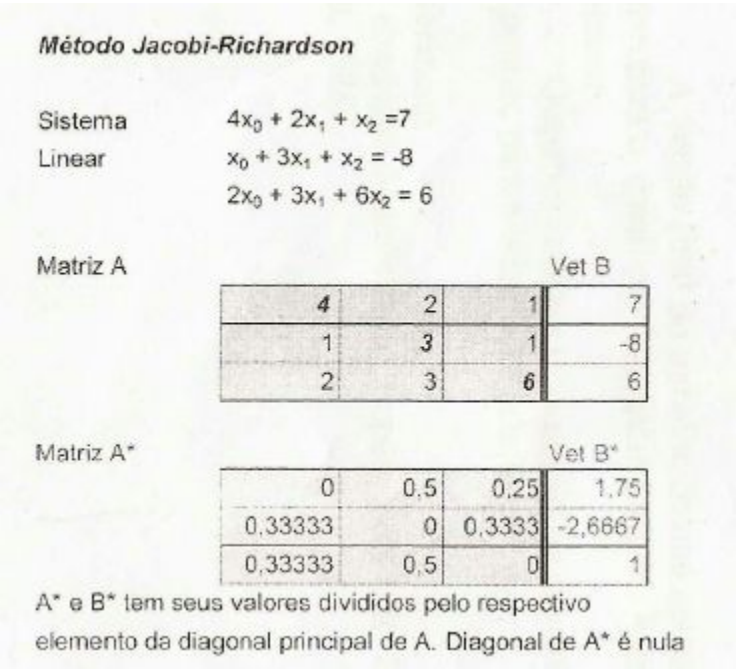
Sumário

1. Introdução

O método iterativo de Jacobi-Richardson é utilizado para encontrar soluções aproximadas de um sistema linear $Ax = b$. O objetivo deste trabalho é comparar o desempenho deste método em duas aplicações diferentes: uma sequencial ,e outra em paralelo utilizando POSIX threads. Ambas foram implementadas na lingaugem C.

1.1 Algoritmo sequencial

Tendo como exemplo a imagem 1.1.1, a versão sequencial foi implementada da seguinte maneira:



Método Jacobi-Richardson

Sistema Linear

$4x_0 + 2x_1 + x_2 = 7$
$x_0 + 3x_1 + x_2 = -8$
$2x_0 + 3x_1 + 6x_2 = 6$

Matriz A

			Vet B
4	2	1	7
1	***3***	1	-8
2	3	***6***	6

Matriz A*

			Vet B*
0	0,5	0,25	1,75
0,33333	0	0,3333	-2,6667
0,33333	0,5	0	1

A* e B* tem seus valores divididos pelo respectivo elemento da diagonal principal de A. Diagonal de A* é nula

Imagem 1.1.1 - Exemplo de criação das matrizes utilizadas no método iterativo

Primeiramente, são realizadas as leituras das matrizes A e B (armazenados em *matriz_A_aux* e *matriz_B*), e logo em seguida é feito o cálculo de A* e B* (armazenados em *matriz_A* e *vet_B*).

Em seguida, tem-se a a parte principal do método: as iterações. Antes de qualquer cálculo, é realizado uma atualização de valores dos vetores da iteração anterior, além de um “reset” nas variáveis que serão reutilizadas a cada iteração, como na imagem a seguir:

```
for (i = 0 ; i < J_ORDER ; i++) {
    vet_aux[i] = matriz_B[i];
    matriz_B[i] = vet_B[i];
    diff[i] = 0;
}
Mr = 0;
soma_row_test = 0;
```

Imagem 1.1.2 - Atualização das variáveis no ínicio de uma iteração

$$\text{Vetor } x[i]^{k+1} = B^*[i] - \sum_{j=0}^{j<n} (A^*[i\ j].x[j]^k),\ \text{para } i <> j$$

Imagem 1.1.3 - Cálculo de x para cada iteração

Logo após é feito o o cálculo da Imagem 1.1.3, sendo o vetor $x[i]^{k+1}$ a variável *matriz_B*, B*$[i]$ a variável *vet_B*, e a somatória é feita da seguinte maneira:

```
for(i = 0; i < J_ORDER; i++) {
      for(j = 0; j < J_ORDER; j++) {
          if (i != j) {
              matriz_B[i] = matriz_B[i] - vet_aux[j]*matriz_A[i][j];
          }
      }
```

Imagem 1.1.4 - Código utilizado para o cálculo da Imagem 1.1.3

Sendo *vet_aux* o vetor x da iteração anterior ($x[i]^k$), e *matriz_A* a matriz A*.

Ainda dentro da iteração, é feito o cálculo do critério de parada.

$$Dif[i]^{k+1} = Abs(x[i]^{k+1} - x[i]^k),\ \text{para } 0 >= i < n$$
$$Mr^{k+1} = Max(Dif[0]^{k+1}\ ;\ ...;\ Dif[n-1]^{k+1}\ ) / Max(Abs(x[0]^{k+1}\ ;\ ...;\ x[n-1]^{k+1}))$$

.

Imagem 1.1.5 - Cálculo do critério de parada

A função *maximum()* já retorna o valor absoluto e máximo do vetor desejado. Sendo assim, caso *Mr* seja menor que o valor de parada determinado, o programa encerra a função.

Na função *main*, é calculado o tempo para cada execução do método iterativo, além da média e desvio padrão destes tempos.

1.2 Algoritmo paralelo

Antes de utilizar qualquer recurso da POSIX Thread, é necessário utilizar a biblioteca pthread.h. Para isso, basta dar um include da biblioteca no inicio do código e ao compilar, utilizar o parâmetro *-lpthread.*

Para o uso das POSIX Threads primeiramente deve-se declarar uma variável do tipo pthread_t. Como utilizamos mais de uma thread, é criado um vetor do tipo pthread_t com uma quantidade NUM_THREADS de threads. A variável NUM_THREAD é um *define* para facilitar a mudança da quantidades de threads desejada.

Para inicializar uma thread, é utilizado a função pthread_create(). O primeiro parâmetro é uma variável do tipo pthread_t. O segundo são variáveis que dizem a forma que a thread deve ser executada. É utilizado a forma *default,* então é passado NULL nesse parâmetro. O terceiro parâmetro deve conter a função que deverá ser executada naquela thread passada no primeiro parâmetro. Por último, o quarto parâmetro deve possuir a variável que será parâmetro da função que será executada nessa thread criada.

```
ret = pthread_create(&threads[i], NULL, iteracao_thread, (void*)&thr_data[i]);
```

Imagem 1.2.1 Inicialização de um thread usando a função pthread_create()

A função pthread_create() retorna um valor inteiro dizendo se a inicialização da thread foi executada com sucesso. Para verificar isso, esse valor é recebido pela variável ret, e com ret é feita a verificação. Caso tenha dado algum problema, é escrito na tela o erro para avisar ao usuário e a função é cancelada utilizando um *return*.

Como a função pthread_create() só suporta apenas uma variável como parâmetro para a função paralelizada, e precisamos de muitas variáveis para executar o método, foi necessário criar a struct _thread_data_t com todo os atributos necessários, presentes na imagem a seguir.

```
typedef struct _thread_data_t {
    double **matriz_A, *matriz_B, **matriz_A_aux, *matriz_B_aux;
    double *vet_aux, *diff;
    double soma_row_test;
    int ini, end, J_ORDER, J_ROW_TEST;
} thread_data_t;
```

Imagem 1.2.2 Struct com os dados necessários para se calcular o método. Entre as variáveis da struct, é possível destacar as matrizes A e B que possuem os valores do sistema linear, J_ORDER que possui a ordem da matriz, etc.

É então criado um vetor do tipo struct _thread_data_t do tamanho NUM_THREADS, ou seja, um para cada thread a ser criada. Essa struct é inicializada em um *loop* junto com a chamada da função pthread_create(). Esse loop executa NUM_THREADS vezes.

A função iteração_thread() possui a parte principal do código do método Jacobi-Richardson. Essa função tem como parâmetro a variável *arg* que é do tipo *void ** que recebera uma variável de qualquer tipo da chama da função pthread_create(). Posteriormente é necessário criar dentro da função uma variável do mesmo tipo que a variável recebida por *arg* e passar para essa nova variável fazendo um *cast.* No fim da função é necessário executar a função pthread_exit() que finalizará a thread criada.

```
void *iteracao_thread(void *arg) {
        int i,j;
        thread_data_t *thr_data = (thread_data_t *)arg;

      for(i = thr_data->ini; i < thr_data->end; i++) {
            for(j = 0; j < thr_data->J_ORDER; j++) {
                if (i != j) {
                    thr_data->matriz_B[i] = thr_data->matriz_B[i] - thr_data->vet_aux[j]*thr_data->matriz_A[i][j];
                }
            }
            thr_data->diff[i] = thr_data->matriz_B[i] - thr_data->vet_aux[i];
            thr_data->soma_row_test += thr_data->matriz_A_aux[thr_data->J_ROW_TEST][i]*thr_data->vet_aux[i];
      }
        pthread_exit(NULL);
}
```

Imagem 1.2.3 Imagem que mostra a função que será executada nas threads criadas.

Logo em seguida do loop de criação/inicialização das threads é necessário executar a função pthread_join(). Essa função tem o papel de deixar as threads acabarem apenas quando todas estão aptas a acabar. Necessita rodar a função pthread_join() NUM_THREADS vezes, ou seja, uma para cada thread criada.

As demais partes do código são semelhantes à versão sequencial.

2. Resultados

	Ordem da matriz						
Sequencial	**250**	**500**	**1000**	**1500**	**2000**	**3000**	**4000**
Tempo de execução (em segundos)	0,180	1,210	11,900	38,840	84,410	279,190	496,700
	0,180	1,200	11,880	38,780	84,600	279,600	496,040
	0,170	1,210	11,880	38,890	84,750	279,530	495,710
	0,170	1,210	11,930	38,940	84,490	279,870	496,190
	0,180	1,200	11,880	38,810	84,560	279,660	496,490
	0,170	1,210	11,870	38,860	84,580	279,540	496,040
	0,170	1,210	11,900	39,000	84,600	279,610	496,980
	0,180	1,210	11,880	38,950	84,470	279,720	495,950
	0,180	1,210	11,860	39,010	84,600	279,320	496,470
	0,180	1,210	11,800	38,970	84,470	279,450	496,650
Média	**0,176**	**1,208**	**11,878**	**38,905**	**84,553**	**279,549**	**496,322**
Desvio padrão	**0,005**	**0,004**	**0,034**	**0,081**	**0,097**	**0,195**	**0,398**

2 threads	**250**	**500**	**1000**	**1500**	**2000**	**3000**	**4000**
Tempo de execução (em segundos)	0,262	1,293	7,879	27,114	58,100	188,777	343,854
	0,262	1,300	7,590	27,208	58,238	189,900	335,293
	0,267	1,301	7,481	27,132	58,105	189,274	333,113
	0,266	1,294	7,541	27,242	58,400	190,124	335,842
	0,263	1,287	7,571	28,658	58,189	191,472	341,846
	0,261	1,294	7,680	29,931	58,132	190,892	346,554
	0,256	1,302	7,549	28,452	58,159	189,137	340,498
	0,267	1,298	7,479	27,537	58,252	188,377	331,260
	0,266	1,303	7,306	27,112	58,296	188,704	333,191
	0,263	1,308	7,319	27,236	58,253	190,210	333,670
Média	**0,263**	**1,298**	**7,540**	**27,762**	**58,212**	**189,687**	**337,512**
Desvio padrão	**0,003**	**0,006**	**0,166**	**0,950**	**0,094**	**1,007**	**5,263**

3 threads	250	500	1000	1500	2000	3000	4000
Tempo de execução (em segundos)	0,219	0,958	6,699	24,195	56,855	189,750	325,879
	0,223	0,964	6,726	25,802	56,707	189,804	326,323
	0,221	0,965	6,662	24,710	56,675	190,642	326,398
	0,220	0,973	6,721	24,980	56,676	190,146	326,551
	0,220	0,966	6,359	24,239	56,643	189,199	326,760
	0,226	0,968	6,709	23,737	56,681	189,918	325,751
	0,221	0,971	6,705	25,710	56,803	189,813	326,568
	0,221	0,966	6,711	25,676	56,779	190,330	326,134
	0,222	0,962	6,715	25,799	56,661	190,141	328,533
	0,219	0,991	6,277	24,476	56,587	189,902	325,530
Média	0,221	0,968	6,628	24,932	56,707	189,965	326,443
Desvio padrão	0,002	0,009	0,166	0,773	0,081	0,386	0,833

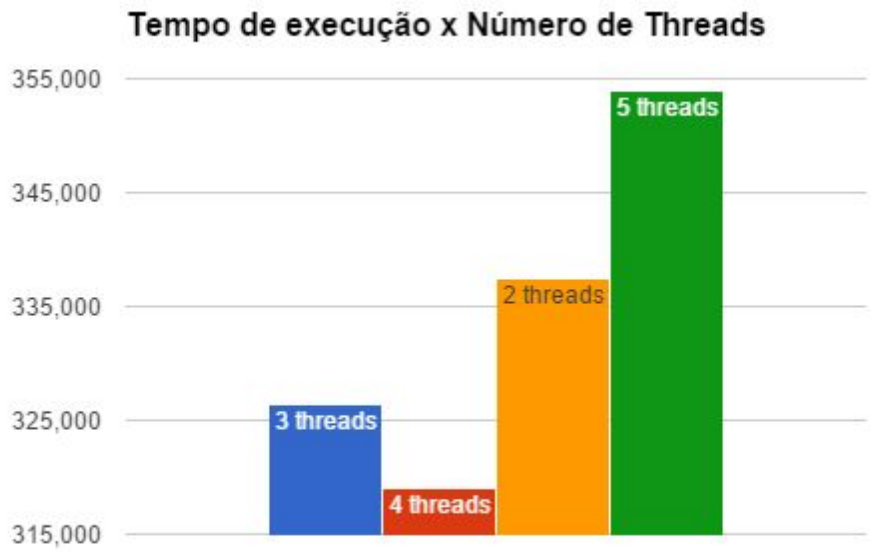
4 threads	250	500	1000	1500	2000	3000	4000
Tempo de execução (em segundos)	0,235	0,844	5,476	31,947	56,678	179,174	324,102
	0,233	0,848	5,765	27,612	56,734	179,222	318,632
	0,233	0,853	5,581	27,361	57,367	179,473	316,798
	0,234	0,862	5,503	26,860	57,270	180,310	317,990
	0,236	0,860	5,408	26,149	56,766	179,688	318,969
	0,229	0,869	5,454	26,188	57,380	179,914	316,422
	0,234	0,887	5,473	25,880	56,790	180,116	323,652
	0,233	0,865	5,594	26,530	56,935	180,439	318,476
	0,236	0,840	5,517	26,700	56,546	180,707	316,630
	0,263	0,870	5,487	25,789	56,666	180,146	318,323
Média	0,237	0,860	5,526	27,102	56,913	179,919	318,999
Desvio padrão	0,009	0,014	0,101	1,805	0,311	0,311	2,722

5 threads	250	500	1000	1500	2000	3000	4000
Tempo de execução (em segundos)	0,304	1,308	8,120	30,960	65,262	212,574	353,648
	0,309	1,251	8,120	30,676	64,833	212,648	354,169
	0,308	1,225	8,170	30,772	65,197	212,766	353,914
	0,353	1,216	8,330	30,568	65,669	213,520	354,115
	0,330	1,244	8,200	30,607	66,567	212,361	354,900
	0,322	1,257	8,510	30,713	66,267	212,349	353,977
	0,306	1,235	8,347	30,591	65,157	212,745	354,222
	0,302	1,268	8,374	30,805	65,653	213,471	353,532
	0,302	1,253	8,156	30,674	67,449	212,455	353,237
	0,302	1,207	8,160	30,660	66,485	213,612	354,011
Média	**0,314**	**1,246**	**8,249**	**30,703**	**65,854**	**212,850**	**353,973**
Desvio padrão	**0,017**	**0,029**	**0,133**	**0,118**	**0,817**	**0,494**	**0,451**

Matriz 250		Matriz 500		Matriz 1000	
Speedup	Sequencial	Speedup	Sequencial	Speedup	Sequencial
2 threads	0,668	2 threads	0,931	2 threads	1,575
3 threads	0,796	3 threads	1,247	3 threads	1,792
4 threads	0,744	4 threads	1,405	4 threads	2,150
5 threads	0,561	5 threads	0,969	5 threads	1,440

Matriz 1500		Matriz 2000		Matriz 3000		Matriz 4000	
Speedup	Sequencial	Speedup	Sequencial	Speedup	Sequencial	Speedup	Sequencial
2 threads	1,401	2 threads	1,452	2 threads	1,474	2 threads	1,471
3 threads	1,560	3 threads	1,491	3 threads	1,472	3 threads	1,520
4 threads	1,436	4 threads	1,486	4 threads	1,554	4 threads	1,556
5 threads	1,267	5 threads	1,284	5 threads	1,313	5 threads	1,402

Gráfico 1 - Relação entre o tempo de execução e o número de threads utlizado na execução



Observando as tabelas, percebemos que o desempenho de todas as versões paralelas é maior que o sequencial, com algumas exceções (que geralmente estão relacionadas a uma ordem matricial menor).

Além disso, vemos que, dentre as demais versões paralelas, a versão com 4 threads apresentou o melhor desempenho, e consequentemente, o maior Speedup.

3. Dificuldades

A principal dificuldade no desenvolvimento do trabalho foi no aprendizado e na utilização das funções pthread. Apesar de threads não ser um assunto novo, a utilização da pthreads especificamente na linguagem C foi um grande desafio, pois até este momento na graduação tínhamos programado apenas sequencialmente C.

Um problema específico que podemos citar em relação a pthreads foi na utilização do pthread_create(). A função pthread_create() é utilizada para criar uma nova thread. O seu quarto e último parâmetro recebe o conteúdo que é desejado passar para dentro da função que será executado na thread. No nosso caso resolvemos passar uma struct contendo todo o conteúdo necessário para realizar uma iteração do método de Jacobi-Richardson. Porém criamos apenas uma struct, e passamos essa única struct para todas as threads geradas e que estão rodando em paralelo. Demoramos para perceber, mas o programa não deixa modificar, um vetor que seja, em uma thread se esse conteúdo já estiver sendo executado em outro thread. Por fim, tivemos então que replicar essa struct de modo que cada thread receba uma struct individual.

4. Hardware utilizado

CPU:
Intel(R) Core(TM)2 Quad CPU Q9650 @ 3.00GHz
Cpu MHz: 2003.000
Tamanho cache: 6144 KB
Cpu cores: 4
tamanho endereço: 36 bits physical, 48 bits virtual
Memória RAM:
Memória Total: 3918076 kB
Buffers: 121568 kB
Em Cache: 657280 kB
Sistema Operacional:
Linux ltsp154 3.2.0-76-generic #111-Ubuntu 12.04.5 LTS

5. Metodologia de execução

Para a execução das aplicações, foi utilizado o compilador GCC, em Linux, com a otimização -O3, a qual aumentou o desempenho das aplicações de maneira muito significativa.

6. Conclusões

Analisando os resultados, podemos concluir que uma aplicação paralela é mais eficiente que uma sequencial, ou seja, que graças as threads, podemos executar diversas linhas diferentes de código ao mesmo tempo, ao passo de que uma aplicação sequencial (na qual o próprio nome já diz) consegue realizar apenas uma única sequência de código em um intervalo de tempo.

Além disso, vemos que, por meio da análise das tabelas, quanto maior o número de threads utilizados no método, melhor é o desempenho do programa. Por exemplo, um programa que utiliza 2 threads consegue dividir a tarefa do cálculo da iteração por 2, sendo que com 4 threads é possível realizar a tarefa dividindo o mesmo cálculo por 4, e consequentemente, em um intervalo de tempo menor.

Porém, há um fato muito importante a ser analisado: a ocorrência de um *overhead*. Vimos que, conforme o número de threads foi aumentando, o tempo de execução diminuiu em quase todas as execuções realizadas e registradas nas tabelas. Porém, quando comparamos a versão de 4 threads com a de 5 threads, vemos uma perda de desempenho, ao invés de um ganho esperado. Isso ocorre devido o surgimento de um *overhead* no processamento, ou seja, o processador não tem a capacidade de executar mais do que 4 threads ao mesmo tempo, e acaba rodando apenas 4 das 5 threads, deixando uma sempre em espera.

As médias e os desvios padrões não apresentam nenhuma anormalidade, portanto podemos dizer que os testes foram realizados sem nenhuma interrupção e de maneira correta.

7. Bibliografia

Multithreaded Programming (POSIX pthreads Tutorial). Disponível em <http://randu.org/tutorials/threads/>. Acesso em 06 de setembro de 2015.