Universidade de São Paulo
IME-USP

Relatório EP1
MAC0219

Tiago Martins Napoli - 9345384

1 Implementação

Foram testadas diversas variantes de implementações a fim de procurar a que possuía a melhor performance para o problema. Em todas elas, na multiplicação das matrizes $A * B$, a segunda matriz foi transposta, daí a multiplicação deve ser feita linha com linha, ao invés de linha com coluna. A vantagem disso vem da representação das matrizes na memória em *C/C++*: Elementos em mesma linha ficam em posições adjacentes na memória. Elementos em mesma coluna já não ficam. Assim, ao transpormos a matriz B, o acesso aos elementos de mesma linha fica sequencial.

Primeiramente foram implementadas funções de *dot product*, a fim de encapsular as operações de multiplicar cada elemento de uma linha por um elemento de uma coluna, na operação da multiplicação de matrizes:

Abaixo está a implementação do *dot product* sequencial. Os parâmetros x e y são os operandos:

```
1  //ll e uma macro para long long
2  inline double dot_product(double *x, double *y, ll sz) {
3      double res = 0.0;
4      for(ll i=0;i<sz;i++) {
5          res += x[i] * y[i];
6      }
7      return res;
8  }
```

Foi implementada também a versão do *dot product* paralelo, com uso do **OpenMP**. Entretanto, o uso deste tornou as operações muito mais lentas. Com um pouco de pesquisa foi encontrado uma referência que afirmava que a paralelização do *dot product* só vale a pena para tamanhos de vetores realmente muito grandes, pois assim seria compensado os custos adicionais de criação e sincronização de threads. Como no caso geral da nossa aplicação uma única coordenada não será tão grande, mas a multiplicação das duas que será, temos que o uso do *dot product* paralelo não traz benefícios, que foi constatado em vários experimentos realizados, com matrizes de até 1000×1000.

```
1  inline double parallel_dot_product(double *x, double *y, ll sz) {
2      #pragma omp parallel for reduction(+:res)
3      for(ll i=0;i<sz;i++) {
4          res += x[i] * y[i];
5      }
6      return res;
7  }
```

1.1 Multiplicação Matricial Sequencial

A implementação sequencial mostrada abaixo foi usada a fim de se constatar a melhora nos tempos de execução das funções paralelizadas.

A função recebe as matrizes A e B, juntamente com suas dimensões, e a matriz C, onde será armazenado o resultado de $A * B$.

```
1  //pll e uma macro para pair<long long, long long>
2  double sequential_mul(double **A, pll dimA,
3                        double **B, pll dimB,
4                        double **C, pll dimC) {
5      double t1,t2;
6      t1 = omp_get_wtime();
7      for(ll i=0;i < dimA.fi;i++) {
8          for(ll j=0;j < dimB.fi;j++) {
9              C[i][j] = dot_product(A[i], B[j], dimA.se);
10         }
11     }
12     t2 = omp_get_wtime();
13     return (t2-t1)*1e3;
14 }
```

1.2 Multiplicação Matricial com OpenMP

Foram feitas duas implementações com *OpenMP*. Elas foram comparadas e a com menor rendimento em média foi retirada do programa.

A primeira é mostrada a seguir. Nesta implementação o *for* interno foi paralelizado:

```
1  double openmp1_mul(double **A, pll dimA,
2                     double **B, pll dimB,
3                     double **C, pll dimC) {
4      //Parallel on B lines.
5      double t1,t2;
6      t1 = omp_get_wtime();
7      for(ll i=0;i<dimA.fi;i++) {
8          #pragma omp parallel for
9          for(ll j=0;j<dimB.fi;j++) {
10             C[i][j] = dot_product(A[i], B[j], dimA.se);
11         }
12     }
13     t2 = omp_get_wtime();
14     return (t2-t1)*1e3;
15 }
```

A segunda implementação é apresentada abaixo. Nela o *for* externo foi paralelizado.

```
1  double openmp2_mul(double **A, pll dimA,
2                     double **B, pll dimB,
3                     double **C, pll dimC) {
```

```
4       //Parallel on A lines.
5       double t1,t2;
6       t1 = omp_get_wtime();
7
8       #pragma omp parallel for
9       for(ll i=0;i<dimA.fi;i++) {
10          for(ll j=0;j<dimB.fi;j++) {
11              C[i][j] = dot_product(A[i], B[j], dimA.se);
12          }
13      }
14
15      t2 = omp_get_wtime();
16      return (t2-t1)*1e3;
17  }
```

Durante a realização dos experimentos, a segunda implementação obteve os melhores resultados. Uma explicação para esse fato talvez seja que na primeira os custos com esperas são muito maiores, já que o número de threads criada é muito maior: Para cada iteração do *for* externo são criadas diversas threads para parelizar o *for* interno, e para avançar para a próxima iteração do *for* externo, é necessário esperar todas as threads terminarem as operações do *for* interno.

1.3 Multiplicação Matricial com Pthreads

A implementação com *Pthreads* foi feita paralelizando o *for* externo. Na operação $A * B$, as linhas de A foram divididas igualmente entre as threads, a fim de realizar as operações relacionadas a elas (se não é possível dividir igualmente, a última thread ficará com o resto da divisão). A função chamada para cada thread é a seguinte:

```
1   void *worker(void *thread_arg) {
2       thread_data *args;
3       args = (thread_data*) thread_arg;
4       for(ll i=args->l_ini; i<args->l_fim; i++) {
5           for(ll j=0; j<args->dimB.fi; j++) {
6               args->C[i][j] = dot_product(args->A[i],
7                                           args->B[j],
8                                           args->dimA.se);
9           }
10      }
11      pthread_exit(NULL);
12  }
```

O parâmetro que a função recebe é uma *struct* com as matrizes A, B, C e suas dimensões, além das variáveis *l_ini* e *l_fim*, que indicam o intervalo de

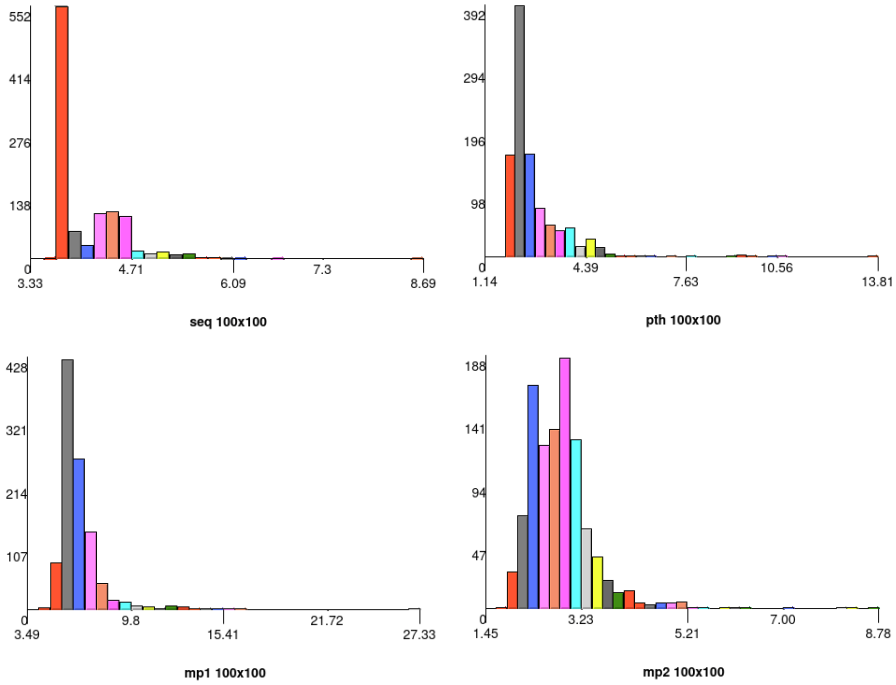
linhas de A, incluindo a primeira e excluindo a última, pelo qual aquela thread é responsável.

2 Testes

Foram realizados testes com matrizes quadradas com certo tamanho fixado. As matrizes foram geradas aleatoriamente, com 80% de ocupação, no sentido de posições na matriz com valores diferentes de 0.0. O número de threads usadas foram 4 (o computador em que foram rodados os testes é dual-core, cada core com hyperthreading).

2.1 Matrizes 100×100

Os dados coletados foram os tempos de 1000 execuções dos algoritmos. Para cada execução foram geradas duas matrizes aleatórias, que foram usadas de entrada para cada algoritmo implementado. Os tempos foram então coletados e foram criados os seguintes histogramas:

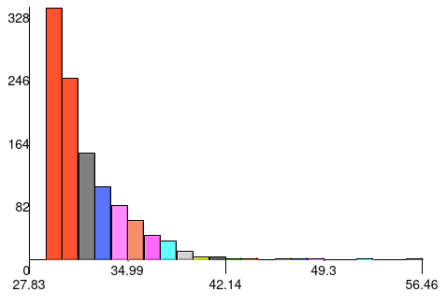


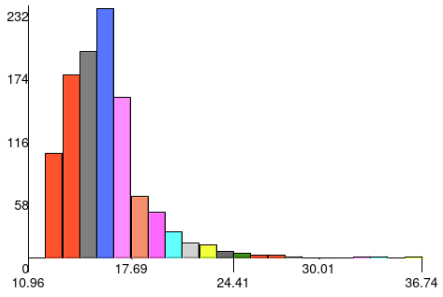
Nos títulos dos histogramas temos que 'seq' refere-se a sequencial, 'pth' a *pthreads*, 'mp1' a *openMP1* e 'mp2' a *openMP2*.
Analisando as distribuições dos dados, se usarmos a média dos dados, não teremos uma informação tão enviesada, uma vez que os dados não estão tão distantes de distribuições normais:

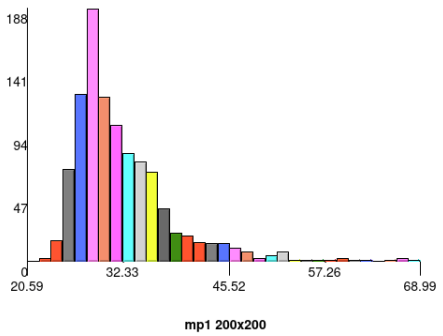
	Média (ms)	Desvio Padrão (ms)	Melhora
Sequencial	4.1	0.5	------
OpenMP1	6.6	1.4	-63.3%
Pthreads	2.8	1.1	32.1%
OpenMP2	2.9	0.7	29.5%

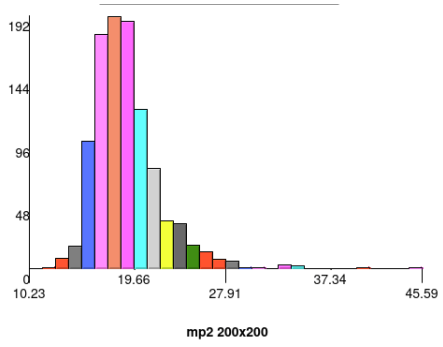
2.2 Matrizes 200×200

Os dados coletados foram os tempos de 1000 execuções de cada algoritmo. Foram criados os seguintes histogramas:







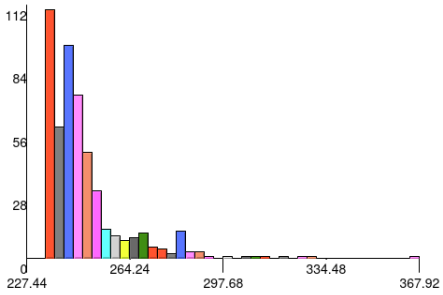


Novamente tomar as médias como aproximador não seria enviesado, já que os histogramas não estão distantes de distribuições normais:

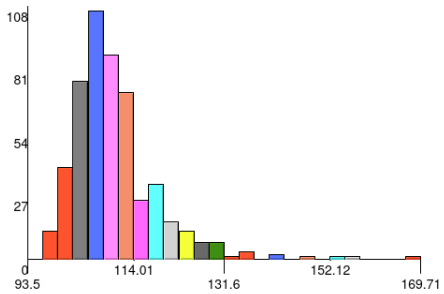
	Média (ms)	Desvio Padrão (ms)	Melhora
Sequencial	32.0	2.8	------
OpenMP1	32.3	6.2	-1.0%
Pthreads	16.0	2.6	49.9%
OpenMP2	19.0	3.1	40.5%

2.3 Matrizes 400×400

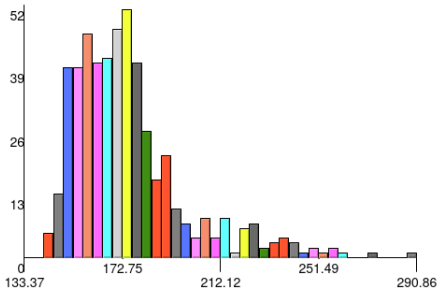
Os dados coletados foram os tempos de 500 execuções de cada algoritmo:



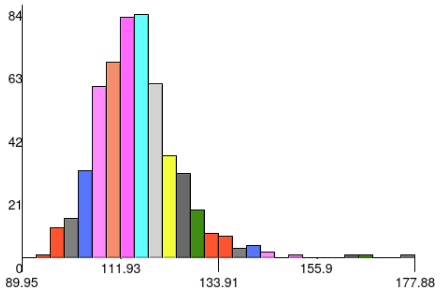
seq 400x400



pth 400x400



mp1 400x400

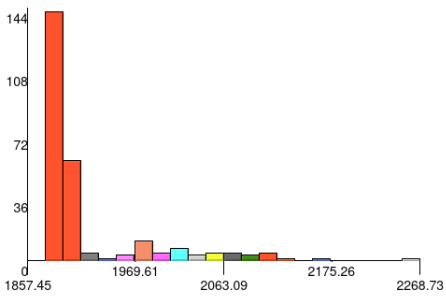


mp2 400x400

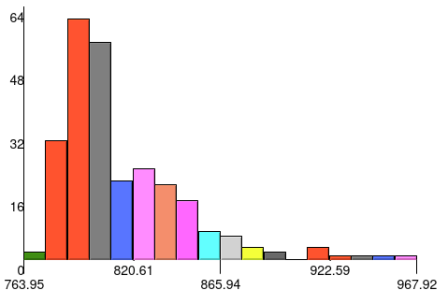
	Média (ms)	Desvio Padrão (ms)	Melhora
Sequencial	248	15	------
OpenMP1	175	23	29.3%
Pthreads	110	8	55.5%
OpenMP2	116	10	53.3%

2.4 Matrizes 800×800

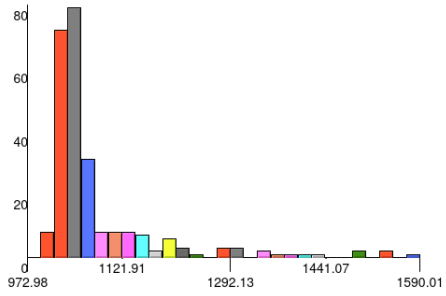
Os dados coletados foram os tempos de 250 execuções de cada algoritmo.



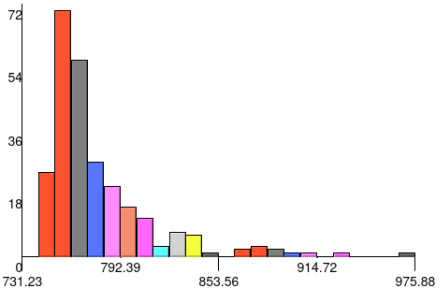
seq 800x800



pth 800x800



mp1 800x800



mp2 800x800

	Média (ms)	Desvio Padrão (ms)	Melhora
Sequencial	1918	62	------
OpenMP1	1082	99	43.6%
Pthreads	815	32	57.5%
OpenMP2	779	34	59.4%

3 Conclusão

Foi possível notar que a paralelização melhorou significativamente os tempos em relação ao sequencial, e quanto maiores as matrizes, mais vantajosa a paralelização ficava. É claro que provavelmente deve haver um limite para a melhora, mas não foi possível confirmar essa hipótese, já que para conseguir uma quantidade razoável de dados para matrizes maiores que 800×800 os tempos, no computador em que os testes foram rodados, ficavam muito grandes.

Quanto ao *OpenMP1* foi possível notar que para matrizes pequenas ele possui performance muito pior que a sequencial, e conforme as matrizes aumentam há uma melhora. Isso pode ocorrer pelo fato de que cada thread desse algoritmo fica responsável por uma quantidade de operações de ordem linear. Quando esse valor é pequeno, o balanço entre os custos para paralelização e a melhora

dos tempos pela divisão de tarefas não compensa. Com o aumento desse valor, aumentando a matriz, o balanço passa a favorecer a paralelização.

Comparando o *OpenMP1* e o *OpenMP2*, temos que no segundo, cada thread é responsável por uma quantidade de ordem quadrática de operações. Assim, o balanço entre os custos de paralelização e a divisão de tarefas favorece mais rapidamente esse algoritmo. Por isso já com matrizes 100×100 temos uma melhora para esse algoritmo em relação ao sequencial, enquanto para o *OpenMP1* há uma piora.

Quanto ao uso de *Pthreads*, ela possui, na maioria dos testes feitos, uma pequena vantagem de performance em relação ao *OpenMP2*. Entretanto, conforme as matrizes aumentam, eles passam a ficar mais próximos, mas comparando a quantidade de código necessária para paralelizar com *Pthreads* e para fazê-lo com *OpenMP*, a vantagem fica para o segundo.