FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Relatório: Implementação e análise de performance dos algoritmos de decomposição LU e Crivo de Eratóstenes.

4º Ano do Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela

Aluno:
Cleverson Veloso Nahum - 201710783 - up201710783@fe.up.pt

01 de maio de 2018

1 Introdução

O presente trabalho tem o objetivo de implementar os algoritmos de decomposição LU e o Crivo de Eratóstenes utilizando a linguagem C++, além de implementar diferentes versões desses algoritmos utilizando as bibliotecas OpenMP (para memória partilhada) e OpenMPI (para memória distribuída).

Após a implementação dos algoritmos é realizada a análise de desempenho de forma a coletar os dados relevantes para realizar as comparações de cada cenário proposto e gerar as conclusões a partir dos resultados obtidos sobre o comportamento em cada cenário.

2 Descrição do problema

O primeiro problema sugerido tem relação com a geração de números primos até determinado valor. Números primos se caracterizam como números que possuem apenas dois divisores naturais distintos sendo o número 1 e o próprio número. Dessa forma para gerar os números primos até um determinado valor é necessário que seja garantido que todos os números selecionados são unicamente divisíveis (considerando divisões exatas com resto igual a zero) por 1 e por eles mesmos.

O segundo problema apresentado tem relação com a solução de sistemas de equação linear, onde sistema de equações lineares são definidos como um conjunto finito de equações lineares aplicados em um mesmo conjunto também finito de variáveis.

3 Algoritmos propostos

3.1 Crivo de Eratóstenes - Geração de números primos

O Crivo de Eratóstenes [1] é um algoritmo que oferece uma proposta para a geração de números primos até determinado valor. O algoritmo pode ser feito através da escrita em uma tabela de todos os números inteiros de 1 até o valor N que se deseja verificar a existência de números primos, após isso todos os números múltiplos de 2 devem ser retirados (o número 2 não deve ser retirado) pois sabe-se que todos possuem mais divisores do que 1 e ele mesmo. Após esse passo deve se utilizar o número posterior não eliminado na etapa anterior para realizar a avaliação dos múltiplos, continuando o exemplo utilizamos o número 3 (sem eliminá-lo) verificamos

todos os seus múltiplos e o retiramos da tabela. O maior número a ser verificado os múltiplos corresponde a raiz quadrada de N, o que diminui bastante a quantidade de passos do algoritmo.

Após a descrição dos passos anteriores restam apenas os números primos na tabela (números não eliminados).

```
std::vector<bool> primes(n+1, true);
      for (int p=2; (p*p)<=n; p++) {
                  if (primes[p] == true) {
                        for (int i=p*2; i<=n; i += p)
                                    primes[i] = false;
                  }
      }
```

Código Fonte 1: Trecho de código da implementação realizada do algoritmo do Crivo de Eratóstenes

No código fonte 1 podemos perceber que foi criado um array com valores iguais a true, e após isso é feito um laço até a raiz quadrada do valor limite a verificar se é número primo, e após isso segue-se a lógica explicada e os valores múltiplos são setados para false, obtendo todos os números primos com valores iguais a true.

3.2 Decomposição LU

O algoritmo de decomposição LU [2] é um método que fatora uma matriz não singular como o produto de uma matriz triangular superior e uma matriz triangular inferior de forma a facilitar a resolução de sistemas de equação ou encontrar matrizes inversas. Conceitualmente temos que sendo A uma matriz não singular, $A = L * U$, onde L é a matriz triangular inferior e U a superior. Para a implementação do algoritmo foi utilizado como base a implementação de Doolittle onde assumimos sempre que L e U existem e resolvemos as equações para achar as matrizes superior e inferior.

```
for (int i = 0; i < n; i++) {
                  //Calculating Upper matrix
                  for (int k = i; k < n; k++) {
                        // Sum L(i, j) * U(j, k)
```

```
                    int sum = 0;
                    for (int j = 0; j < i; j++)
                              sum += (lower[i][j] * upper[j][k]);

                    // U(i, k)
                    upper[i][k] = mat[i][k] - sum;
               }


               // Lower Matrix
               for (int k = i; k < n; k++) {
                    if (i == k)
                              lower[i][i] = 1; // Diagonal
                    else {
                              // Sum L(k, j) * U(j, i)
                              int sum = 0;
                              for (int j = 0; j < i; j++)
                                   sum += (lower[k][j] *
upper[j][i]);
                             // L(k, i)
                              lower[k][i] = (mat[k][i] - sum) /
upper[i][i];
                    }
               }
     }
```

Código Fonte 2: Trecho de código da implementação realizada para decomposição LU

No código fonte 2 visualizamos como foi implementado o código que calcula os valores para a matriz superior e inferior a cada interação da variável i de forma a percorrer toda a matriz.

3.3 Paralelismo com memória partilhada (Utilizado tanto para o Crivo de Eratóstenes quanto para a decomposição LU)

O paralelismo utilizando memória partilhada foi desenvolvida na implementação através da utilização da biblioteca OpenMP através da utilização do “pragma omp parallel” disponível ilustrado no trecho de código fonte 3 a seguir.

```
#pragma omp parallel for num_threads(nt)
      for (int p=2; p<=sn; p++) {
                  if (primes[p] == true) {
                        for (int i=p*2; i<=n; i += p)
                                    primes[i] = false;
                  }
      }
```

Código Fonte 3: Trecho de código da implementação realizada para crivo de Eratóstenes utilizando OpenMP

A versão do código fonte 3 foi implementada para o crivo de Eratóstenes, mas o mesmo pragma utilizado também foi colocado no laço “for” demonstrado no código fonte 2 mostrado acima para a decomposição LU.

O pragma utilizado irá dividir o loop em intervalos para o determinado número de threads “nt”, de forma que cada thread realize as operações referentes aquele intervalo e divida o trabalho.

3.4 Paralelismo com memória distribuída - Crivo de Eratóstenes

Para a implementação de paralelismo com memória distribuída foi utilizada a biblioteca OpenMPI disponível para C++. Com a utilização desta biblioteca o processo passa a ser distribuídos entre processos diferentes que utilizam espaços de memórias diferentes mas que mantém a comunicação entre seu grupo de processos de forma a trocar informações necessárias para a execução do algoritmo.

No algoritmo do crivo de Eratóstenes o trabalho pode ser executado de forma independente em cada processo desde que seja definido que intervalos de múltiplos cada processo deve analisar, basicamente isso significa definir o intervalo em que cada processo deverá executar seus laços. No trecho do código fonte 4 a seguir, percebe-se a definição da quantidade de interações no laço que cada processo deve executar sendo calculado na variável “ri”, de forma a equilibrar o trabalho entre os processos.

```
int ri= ceil(((float)sn-2.0)/(float)size);
      int limit;
```

```
      if(rank!=(size-1))
                  limit = ((2+rank*ri)+ri-1);
      else
                  limit = sn;

      for (int p=(2+rank*ri); p<=limit; p++) {
```

Código Fonte 4: Trecho de código da implementação realizada para crivo de Eratóstenes utilizando OpenMPI

Outra característica importante é a definição dos limites do laço em cada processo que é feita dentro da cláusula “if else” que define a variável limite, pois se o processo não possuir o último rank o limite do seu loop passa a ser “2+rank*ri+ri-1” e caso o seu rank seja o último dos processos ele deve executar o loop até o valor definido como limite, no caso “sn”. O início do intervalo de cada loop é definido como “2+Número_rank*Quantidade De Iterações _por Processo” no qual foi atribuído ao valor de “p” dentro do loop.

Na divisão entre os processos definiu-se que o processo inicial (rank=0) seria o responsável por receber a resposta dos outros processos e comparar com o seu array de números primos gerados de forma a completar a análise comparando os seus valores com os valores recebidos e assim poder gerar o arquivo CSV com as respostas. O que foi explicado e percebido no trecho de código 5 explicado a seguir.

```
if(rank!=0) { //Sending primes array to process with rank 0
                  MPI_Request status;
                  MPI_Isend(primes.data(), primes.size(), MPI_INT, 0,
rank, MPI_COMM_WORLD, &status);
      }
      else { //Process 0
                  vector<vector<int>> number(size-1,
vector<int>(n+1));
                  MPI_Request req[size-1];
                  MPI_Status status[size-1];

                  for(int i = 0; i<(size-1); i++) //Receiving from
processes
                        MPI_Irecv(&number[i][0], (n+1), MPI_INT, (i+1),
```

```
(i+1), MPI_COMM_WORLD, &req[i]);

                for(int i = 0; i<(size-1); i++) //Waiting Received
values to Number array
                        MPI_Wait(&req[i], &status[i]);

                for(int j=0; j<(size-1);j++) //Comparing received
values with primes generated
                        for(int i=0; i<primes.size();i++)
                                primes[i] &= number[j][i];

                file.open(filename);
                file << "Primes until " << n << "\n";
                if(rank==0)
                        for (int p=0; p<primes.size(); p++)
                                if (primes[p])
                                        file << p << " ";
                file.close();
        }
```

Código Fonte 5: Trecho de código da implementação realizada para crivo de Eratóstenes utilizando OpenMPI onde são enviadas e recebidas as mensagens dos processos

3.5 Paralelismo com memória distribuída - Decomposição LU

Para a implementação da versão com memória distribuída do algoritmo de decomposição LU também foi utilizada a biblioteca OpenMPI. Mas a função utilizada para trocar mensagens entre os processos foi diferente, pois não seria possível dividir intervalos de loop para cada processo, já que para o cálculo tanto da matriz superior quanto da matriz inferior requerem valores anteriores calculados e colocados na matriz principal. Dessa forma optou-se pela utilização da função broadcast disponível na biblioteca do OpenMPI que permite enviar uma determinada variável para todos os processos, de forma a manter atualizado os respectivos valores da matriz utilizada como referência [3].

Com isso, percebe-se que os processos trocam informações entre si antes de acabar as suas instruções e ao final não é necessário enviar os valores processados para um único rank como foi feito na paralelização do algoritmo do crivo de

Eratóstenes, pois ao final é feito o broadcast de partes da matriz de forma a completá-la em todos os processos. Esse processo pode ser melhor percebido no trecho de código fonte 6 abaixo, onde cada linha da matriz processada é enviada em broadcast para os outros processos.

```
for (int i = 0; i < n-1; i++, dref++) {
                float *drow = &mat[dref * n + dref]; //Diagonal row
                for (int j = dref + 1; j < n; j++) {
                     if (j % size == rank) {
                                float *save = &mat[j * n + dref];
                                if (*save == 0) //Forward Gauss
Elimination
                                     return;
                                float k = *save / drow[0];
                                for (int i = 1; i < (n-dref); i++)
                                     save[i] = save[i] - k *
drow[i];
                                *save = k;
                     }
                }

                for (int j = dref + 1; j < n; j++) {
                     float *save = &mat[j * n + dref];
                     MPI_Bcast(save, n - dref, MPI_FLOAT, j % size,
MPI_COMM_WORLD);
                }
     }
```

Código Fonte 6: Trecho de código da implementação realizada para decomposição LU utilizando OpenMPI onde são enviadas as mensagens em broadcast

4 Metodologia e Métricas

Para realizar a avaliação das implementações propostas serão utilizadas diferentes métricas de acordo com as características a serem levadas em consideração de forma a avaliar cada cenário proposto.

Os experimentos foram realizados em um notebook 1 com CPU Intel Core i3-5005U de frequência de 2.0 GHz com 4 cores. E outro notebook 2 com CPU Intel Core i5-7200U de frequência de 2.5 GHz Dual Core. Além de permitir a execução de até 2 threads por core. Para a implementação utilizando openMPI foram utilizados containers Docker para realizar a execução da aplicação openMPI em diferentes ambientes isolados mas que ainda sim compartilham os mesmos recursos da máquina principal.

Cada algoritmo implementado como solução tanto para a geração de números primos quanto para a decomposição LU serão avaliados em 3 diferentes cenários, sendo eles um cenário com o código sem paralelização, código com paralelismo de memória partilhada utilizando OpenMP e código com paralelismo de memória distribuída utilizando OpenMPI. Na avaliação d implementação com OpenMPI cada máquina era limitada a no máximo dois processos executando.

As métricas obtidas para realizar a comparação em todos os cenários são o speedup (1), eficiência (2) e a eficiência da escalabilidade (3). Cujos cálculos estão descritos a seguir:

$$Speedup = S(n,p) = \frac{T(n,1)}{T(n,p)} \quad (1)$$

$$Eficiência = E(n,p) = \frac{S(n,p)}{p} \quad (2)$$

$$Escalabilidade = SE(n,p) = \frac{T(n,1)}{T(pn,p)} \quad (3)$$

4.1 Comparação Crivo de Eratóstenes

Nesse cenário houve a implementação do algoritmo do Crivo de Eratóstenes na linguagem de programação C++. As versões sem paralelismo, paralelismo com memória compartilhada (OpenMP) e com memória distribuída (OpenMPI) foram avaliada levando em consideração a performance do processador, speedup, eficiência e escalabilidade de 1 até 4 processos para a versão paralelizadas.

Em todas as versões implementadas foram consideradas a geração de números primos de 100 milhões até 600 milhões com passos de 100 milhões. Para a coleta das informações de performance foram utilizadas a biblioteca time.h disponível para C++ para obter a métrica de tempo de execução de cada cenário do algoritmo.

4.2 Comparação Decomposição LU

Nesse cenário foi feita a implementação do algoritmo de Doolittle para a decomposição de LU na linguagem de programação C++. As versões sem paralelismo, paralelismo com memória compartilhada (OpenMP) e com memória distribuída (OpenMPI) foram avaliada levando em consideração a performance do processador, speedup, eficiência e escalabilidade de 1 até 4 processos para a versão paralelizadas

Em todas as versões implementadas foram consideradas a geração de matrizes quadradas com dimensões variando de 1 mil a 6 mil com passos de 1 mil entre cada execução. Para a coleta dos dados também foram utilizadas as funções da biblioteca time.h para obter as mesmas métricas da comparação citada para o Crivo de Eratóstenes.

5 Análises e Resultados

5.1 Comparação Decomposição LU

Da figura 1 a 6 é apresentado gráficos simbolizando os tempos de execução dos algoritmos sequenciais, os implementados com OpenMP e OpenMPI. Juntamente com os resultados do speedup, eficiência e a eficiência da escalabilidade.

É observado que tanto o algoritmo com a implementação com memória partilhada (OpenMP) na figura 2 ou memória distribuída (OpenMPI) na figura 3 apresentaram melhorias consideráveis de tempo de execução em relação a execução sequencial do algoritmo (figura 1). Isso se deve ao melhor aproveitamento e divisão do trabalho entre os recursos computacionais disponíveis para a execução.

Ainda comparando os modelos com memória distribuída com o modelo com memória partilhada podemos perceber que a implementação com OpenMP obteve uma performance melhor em relação a implementação com OpenMPI, apesar de teoricamente os recursos computacionais serem maiores na versão com OpenMPI. Isso se deve ao fato da implementação com memória distribuída realizar uma série de envio de mensagens em Broadcast para todos os processos a cada interação do seu Loop, além de que foram necessárias algumas adições de código de forma a possibilitar a paralelização do algoritmo.

Na figura 4, temos os resultados de Speedup que demonstram que a cada processo adicionado na execução distribuída do algoritmo temos melhoras significativas no seu tempo de execução. Mas ainda sim, percebemos que conforme adicionamos processos a execução também perdemos em eficiência, como representado na figura 5, onde a cada processo adicionado a eficiência quanto a utilização dos recursos diminui, não se mantendo as taxas de diminuição de processamento tão altas quanto na passagem de 1 para 2 processos. Outro fator que

sofre perdas significativas é representado na figura 6 que é a eficiência da escalabilidade, onde 2 processos acabam por escalar a aplicação relativamente bem, mas para 4 processos já temos uma perda que o coloca próximo a 0.5 em eficiência da escalabilidade.

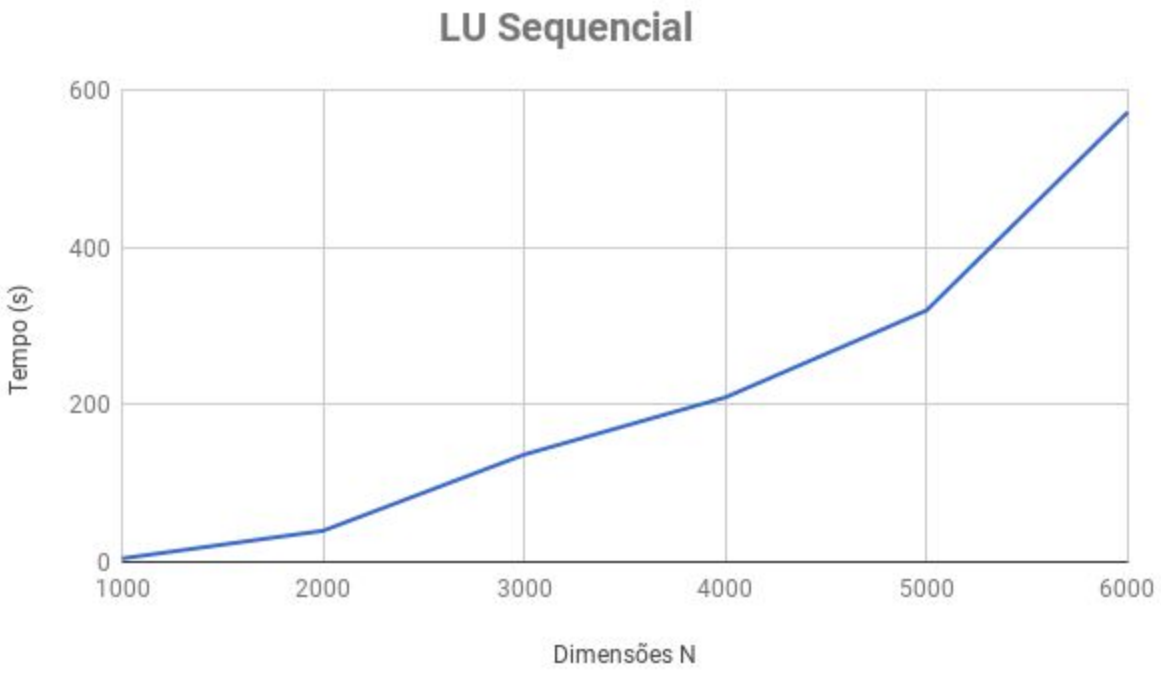

Figura 1: Tempo de execução do algoritmo sequencial da decomposição LU para diferentes dimensões de Matrizes

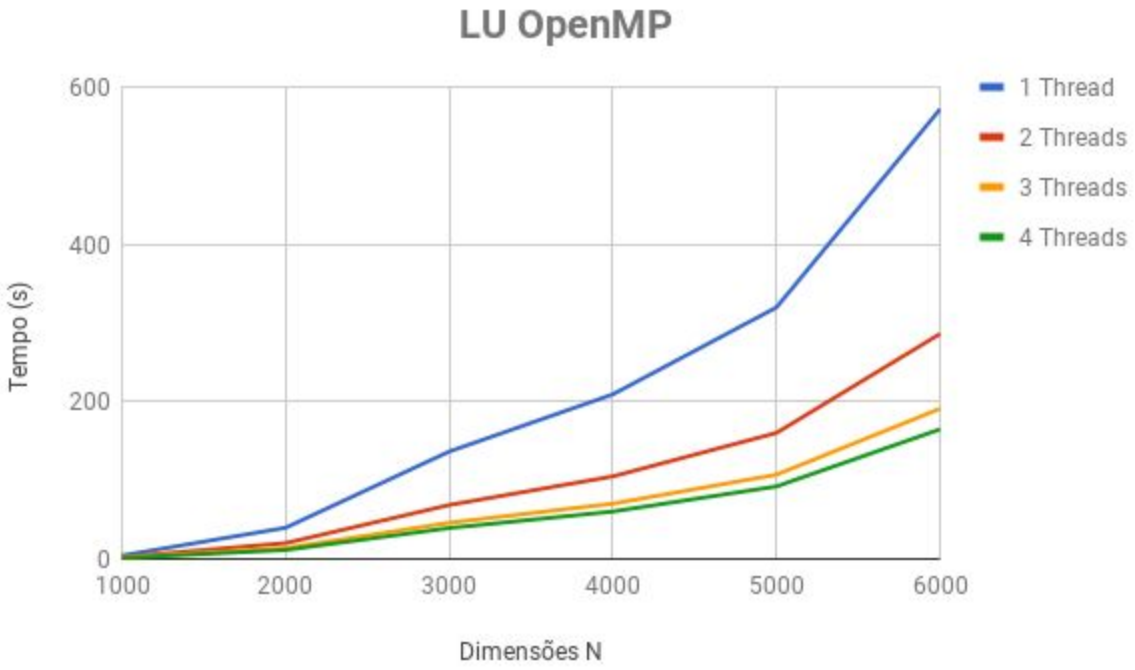

Figura 2: Tempo de execução do algoritmo de decomposição LU utilizando a biblioteca OpenMP para diferentes dimensões de Matrizes e quantidade de threads

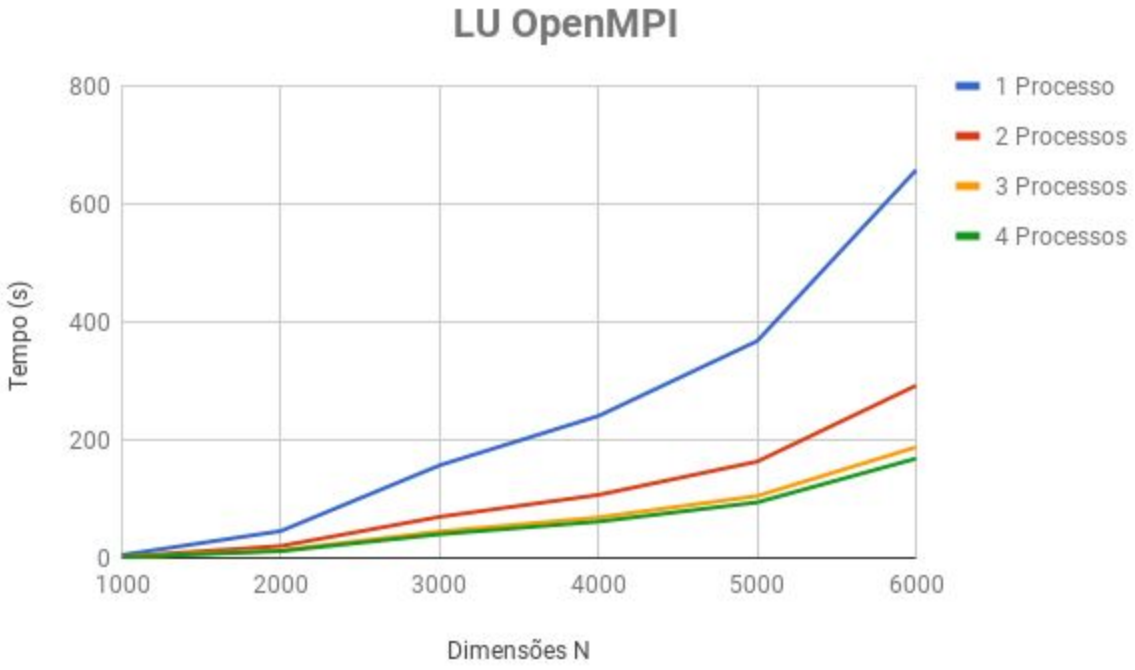

Figura 3: Tempo de execução do algoritmo de decomposição LU utilizando a biblioteca OpenMPI para diferentes dimensões de Matrizes e quantidade de processos

Figura 4: Speedup do algoritmo de decomposição LU utilizando a biblioteca OpenMPI para diferentes dimensões de Matrizes e quantidade de processos

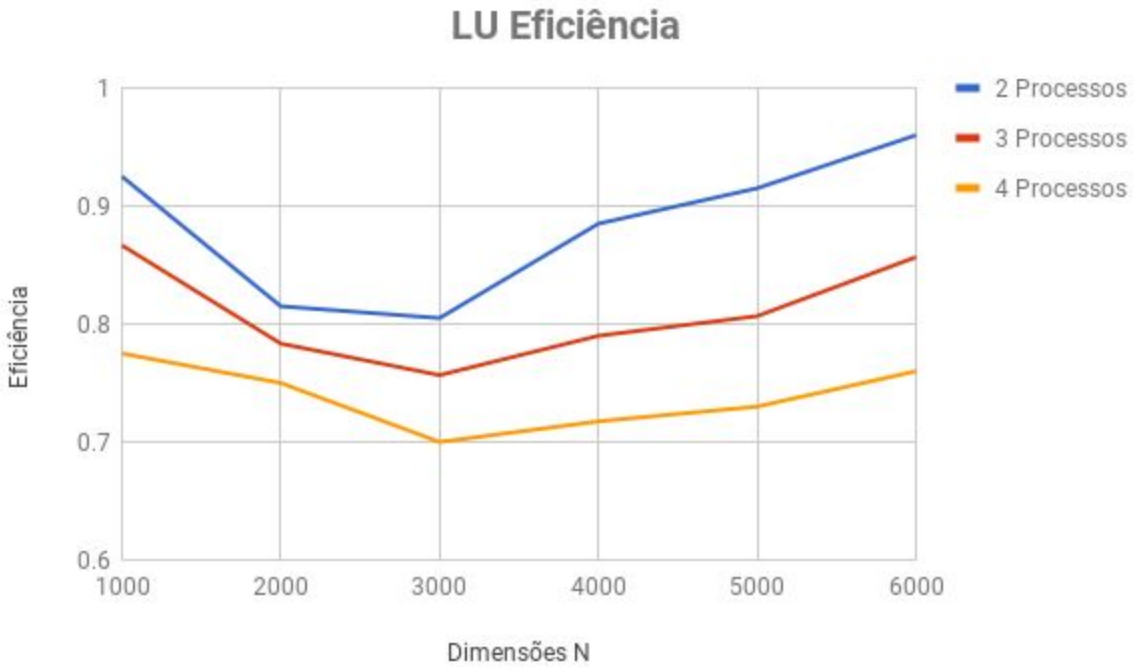

Figura 5: Eficiência do algoritmo de decomposição LU utilizando a biblioteca OpenMPI para diferentes dimensões de Matrizes e quantidade de processos

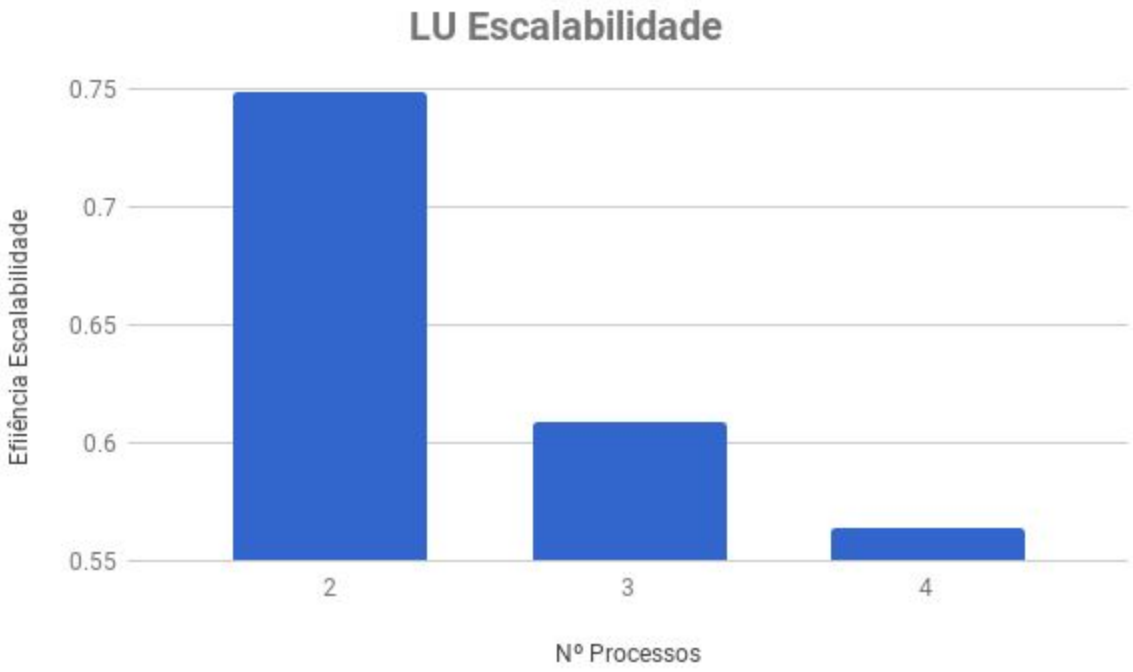

Figura 6: Eficiência da escalabilidade do algoritmo de decomposição LU utilizando a biblioteca OpenMPI para diferentes quantidades de processos

5.2 Comparação Crivo de Eratóstenes

Partindo de uma análise parecida do tópico anterior, temos as figuras 7, 8 e 9 representando respectivamente a implementação sequencial, com OpenMP e OpenMPI. Além dos resultados de speedup, eficiência e eficiência da escalabilidade nas figuras restantes (figura 10, 11 e 12).

Analisando as figuras 7, 8 e 9 podemos perceber que as versões com memória partilhada e distribuída melhoraram consideravelmente o tempo de execução de cada aplicação em relação a implementação sequencial, justificado pela questão da melhor utilização dos recursos computacionais e divisão de trabalho proporcionada pelo uso de threads e processos.

Comparando as versões com OpenMP com o OpenMPI, podemos perceber que ao contrário do que ocorreu na implementação da decomposição LU, a versão com OpenMPI obteve resultados melhores, isso se deve ao fato de que todos os processos executam suas instruções de forma independente, bastando ao final das suas instruções enviar uma mensagem para o processo inicial com os valores da matriz encontrada, tendo um fluxo de mensagens entre os processos bem maior e contando com maiores recursos já que conta com duas máquinas para processá-lo.

Na figura 10, temos os resultados do speedup do algoritmo que demonstram ganhos de velocidade na execução do algoritmo para cada processo adicionado e com ganhos bem semelhantes a cada adição. Na figura 11 temos a relação entre a adição de processos e a eficiência de execução, temos um decaimento considerável na eficiência com a adição de processos assim como na decomposição LU, mas em compensação podemos perceber analisando a figura 12 que o algoritmo do Crivo de Eratóstenes implementado possui melhores índices de eficiência de escalabilidade, fator que pode ser explicado pela maior independência atingida pelos processos devido a não necessidade de troca de mensagens durante a sua execução.

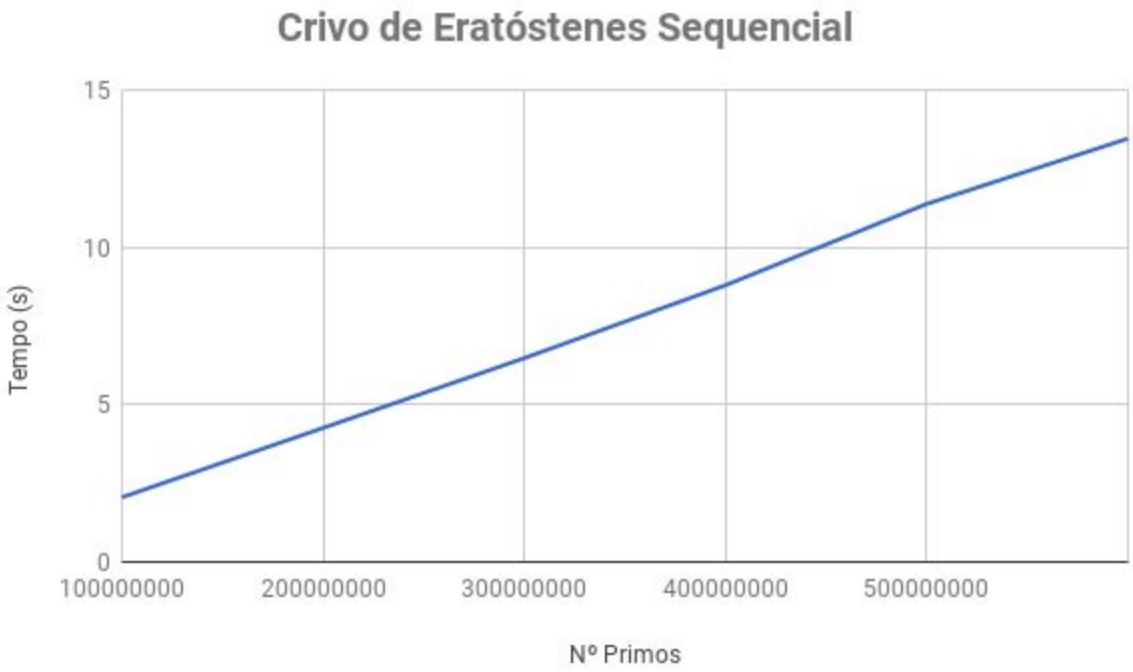

Figura 7: Tempo de execução do algoritmo sequencial do Crivo de Eratóstenes para diferentes quantidades de números primos

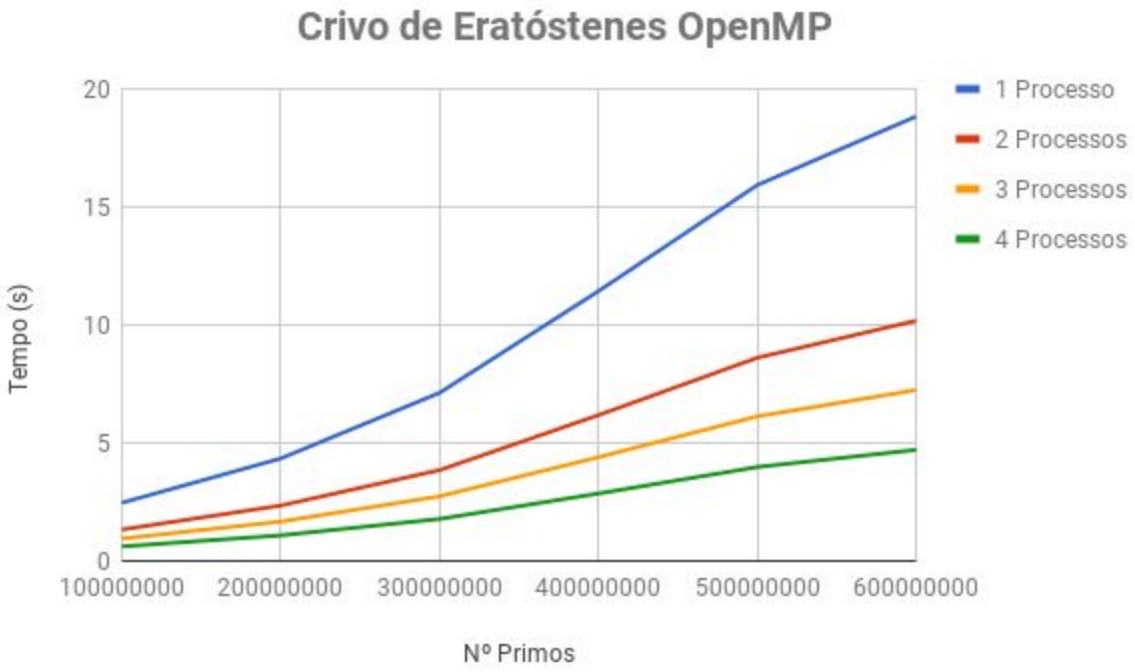

Figura 8: Tempo de execução do algoritmo do Crivo de Eratóstenes utilizando a biblioteca OpenMP para diferentes quantidades de números primos e threads

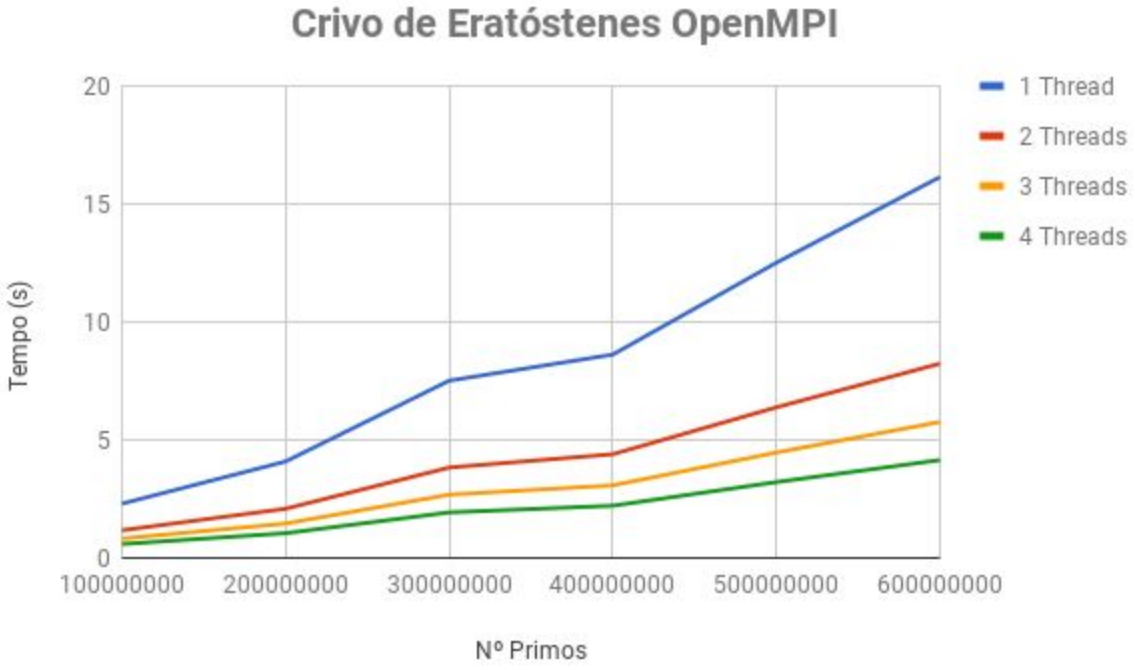

Figura 9: Tempo de execução do algoritmo do Crivo de Eratóstenes utilizando a biblioteca OpenMPI para diferentes quantidades de números primos e processos

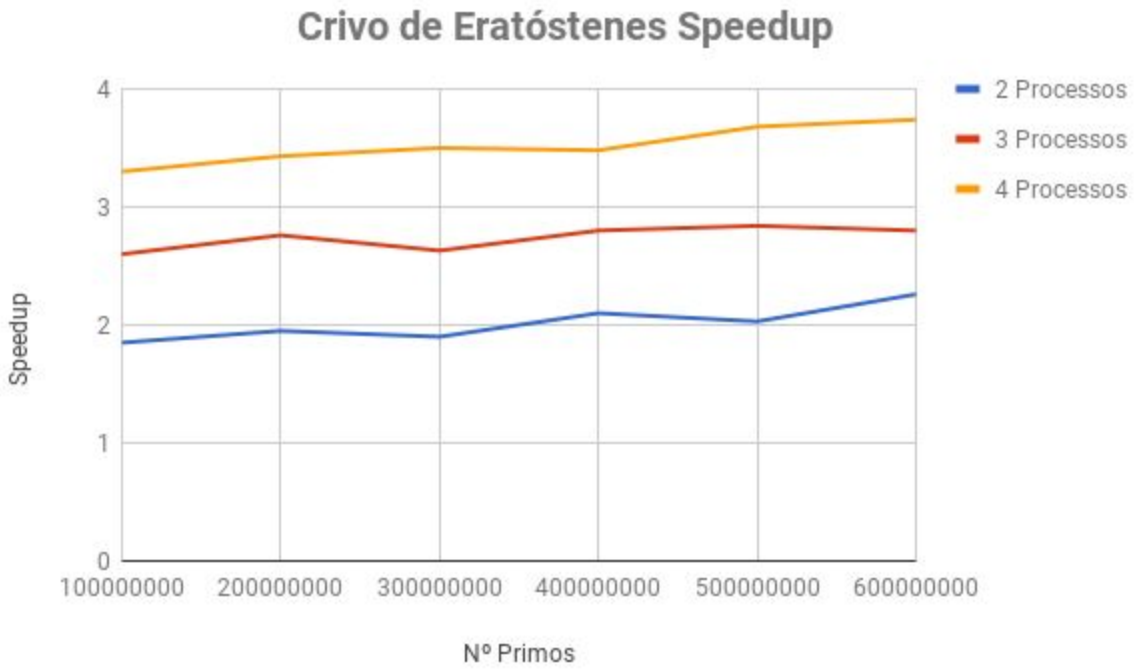

Figura 10: Speedup do algoritmo do Crivo de Eratóstenes utilizando a biblioteca OpenMPI para diferentes quantidades de números primos e processos

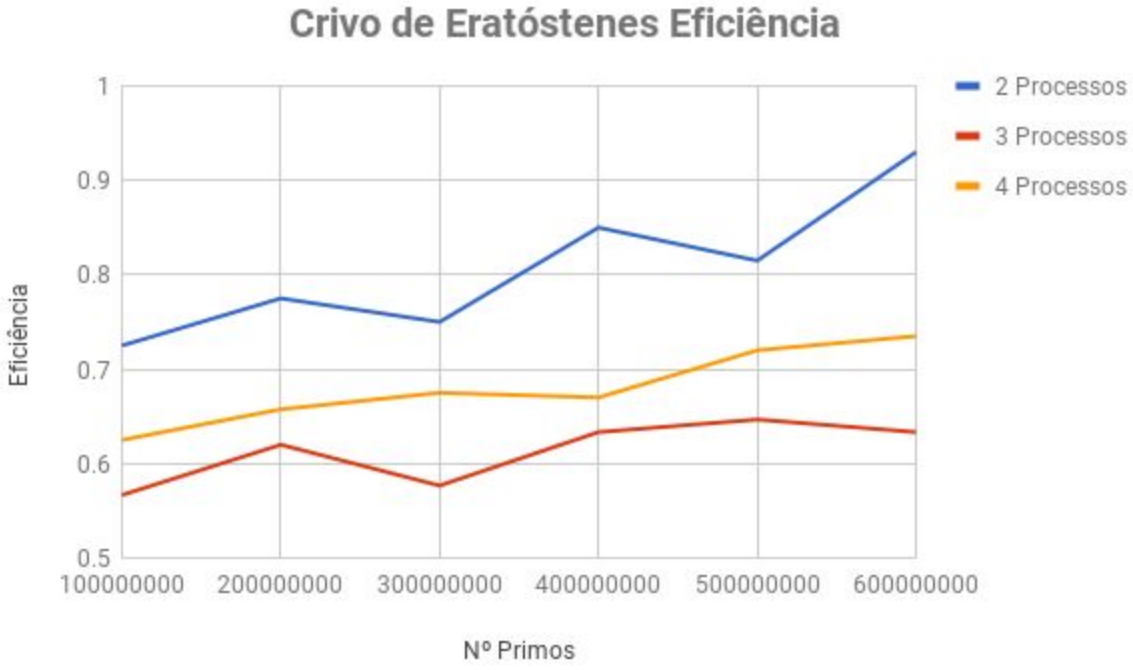

Figura 11: Eficiência do algoritmo do Crivo de Eratóstenes utilizando a biblioteca OpenMPI para diferentes quantidades de número primos e processos

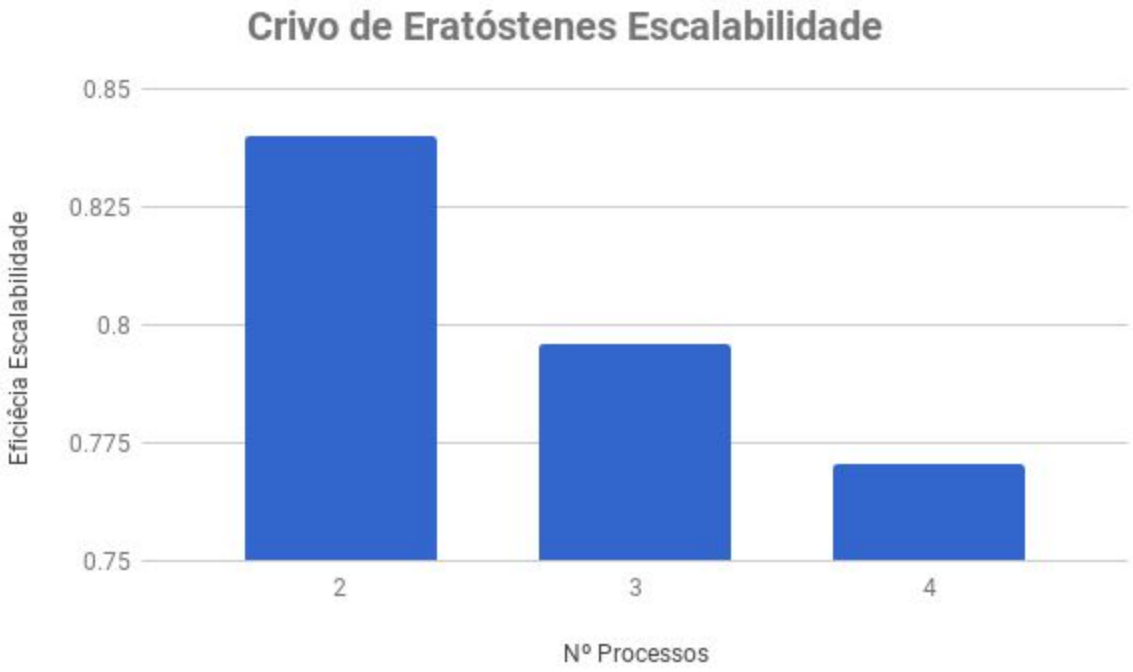

Figura 12: Eficiência da escalabilidade do algoritmo de decomposição LU utilizando a biblioteca OpenMPI para diferentes quantidades de processos

6 Conclusão

O presente trabalho propôs a análise de desempenho de dois algoritmos diferentes para a solução da multiplicação de matrizes em diferentes linguagens, analisando fatores como tempo de execução, acesso a memória e paralelismo. Concluiu-se que a linguagem C++ teve maior eficiência do que Java, e em todos os cenários o algoritmo 2 apresenta vantagens devido se aproveitar de conceitos como o “Princípio de Localidade” e “row-major order” da alocação de memória cache, além de que os dois algoritmos tiveram seus desempenhos melhorados em até 4 vezes após a utilização de threads.

Referências

[1] Bokhari, S. "Multiprocessing the sieve of Eratosthenes." (1986).

[2] Bartels, Richard H., and Gene H. Golub. "The simplex method of linear programming using LU decomposition." *Communications of the ACM* 12.5 (1969): 266-268.

[3] *Cse.Buffalo.Edu*,2018,https://www.cse.buffalo.edu//faculty/miller/Courses/CSE6_33/Tummala-Spring-2014-CSE633.pdf. Accessed 4 May 2018.