Implementação e análise de performance de algoritmos

Relatório 2º Projeto

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Computação Paralela

4º ano – 2º semestre

Trabalho efetuado por:

- Cláudia Margarida da Rocha Marinho – up201404493@fe.up.pt

9 de março de 2018

1. Descrição do problema

No âmbito da unidade curricular de Computação Paralela, foi proposta a realização de um estudo da paralelização dos algoritmos *LU Decomposition* e *The Sieve of Eratosthenes* através do uso de ferramentas que facilitam o desenvolvimento de programas com memória partilhada e memória distribuída.

Para concretizar este estudo, foi necessário implementar três versões diferentes dos dois algoritmos mencionados anteriormente: uma versão sequencial em C/C++, uma versão com memória partilhada usando OpenMP e uma versão com memória distribuída usando MPI.

Neste relatório, vai-se então efetuar uma análise e comparação dos resultados obtidos, de forma a explicar estes resultados e tirar conclusões apropriadas. No entanto, antes de proceder a esta análise é importante explicar os dois algoritmos implementados, como se vai fazer de seguida.

2. Soluções sequenciais e medidas de performance

2.1. Algoritmo LU Decomposition

Em álgebra linear, a decomposição LU é uma forma de fatorização de uma matriz quadrada (matriz com formato NxN, sendo N o tamanho da matriz) como o produto de uma matriz triangular inferior (lower – L) e uma matriz triangular superior (upper – U), de tal forma que se verifique a igualdade **A = LU**, sendo **A** a matriz original. Este método é usado para resolver sistemas de equações lineares, como um passo fundamental na inversão de matrizes ou no cálculo do determinante de uma matriz. Para implementar este método, usei um algoritmo simples baseado na eliminação de Gauss, tal como é apresentado a seguir:

```
for k = 1:n-1
  for i = k+1:n
      A(i,k) = A(i,k)/A(k,k)
  end for
  for i = k+1:n
      for j = k+1:n
        A(i,j) = A(i,j) - A(i,k)* A(k,j)
      end for
  end for
end for
```

Figura 1. Algoritmo utilizado na implementação da decomposição em LU

Este algoritmo permite transformar a própria matriz inicial (A) nas matrizes triangular inferior (L) e superior (U), tirando partido do método de eliminação de Gauss. Para obter estas duas matrizes basta só percorrer a matriz A e cumprir os seguintes requisitos:

- Quando o índice correspondente ao número da linha for maior que o índice correspondente ao número da coluna na matriz A então o elemento correspondente na matriz triangular inferior (L) tem o mesmo valor.
- Quando o índice correspondente ao número da coluna for maior ou igual ao índice correspondente ao número da linha na matriz A então o elemento correspondente na matriz triangular superior (U) tem o mesmo valor.
- A diagonal principal da matriz triangular inferior é toda preenchida com o valor 1.

No algoritmo de decomposição em LU via eliminação de Gauss, percorre-se todos os elementos da diagonal principal da matriz (variável **k** na figura 1) que vão determinar a linha pivot e a coluna pivot. Para cada uma das linhas pivot percorre-se as linhas depois dessa (variável **i** na **figura 1**) de forma a determinar o fator pelo qual multiplicar a linha pivot de forma a que os elementos correspondentes na coluna pivot em cada uma das linhas **i** se tornem zero. É importante notar no entanto que cada um dos fatores calculados vai ser atribuído aos elementos mencionados anteriormente para determinar a matriz triangular inferior. Depois de calculado o factor referido, vai-se percorrer os elementos de cada a linha **i** (variável **j** na **figura 1**) que se encontram depois da coluna pivot e vai-se a atualizar os valores desses elementos subtraindo a linha pivot pelo factor multiplicado anteriormente. Uma forma muito mais fácil de entender este algoritmo é com um exemplo com matrizes, mas como não há espaço neste relatório para explicar dessa forma, deixo apenas um link para uma página que explica-o bem.

2.2. Algoritmo Sieve of Erathosthenes

Sieve of Erathosthenes é um algoritmo simples que permite encontrar todos os números primos até um dado número. O algoritmo baseia-na seguinte ideia: lista-se todos os números de 2 a *n* (sendo *n* o último número); posteriormente, percorre-se os números dessa lista de tal forma que todos os múltiplos desse número sejam “descartados” como não-primos (note-se que os números só podem ser descartados uma vez) e só se pára este ciclo quando se atinge a raiz quadrada de *n*.

A versão que eu implementei trata-se de uma versão optimizada em que descarta logo de início todos os números pares (já que o único número par que é também primo é 2). Assim inicializei o vector com metade dos números e depois percorreu-se o vector dando saltos de 2 em 2 números. Note-se que um booleano a *true* tem o significado que o número associado a essa posição foi filtrado, ou seja, não é primo.

Também é muito relevante mencionar que o uso do vector<bool> resulta num aumento significativo de performance, visto que internamente é usado um dynamic_bitset. Tal permite encaixar mais valores numa *cache line* resultando assim numa diminuição do tempo de execução.

2.3. Medidas de performance e resultados

Com o fim de comparar a performance das diferentes implementações de ambos os algoritmos, foram efetuados vários testes em que se recolheu o tempo real de execução dos programas. No caso do algoritmo de decomposição em LU, foram realizados testes com matrizes de dimensões 1000x1000 a 6000x6000 (com um passo de 1000) e no caso do algoritmo Sieve of Erathosthenes, foram realizados testes com números desde 100,000,000 até 600,000,000 (com um passo de 100,000,000), tendo sido os resultados dos testes os apresentados na figura 2 e 3 que se encontram nos anexos deste relatório.

3. Soluções paralelas e suas caracterizações

De forma a aumentar a performance da versão sequencial foram codificadas diferentes versões deste usando técnicas de paralelização. Estas implementações visam dividir o problema em várias partes concorrentes de forma a melhor utilizar os recursos disponíveis. Tal foi feito recorrendo a threads com memória partilhada (OpenMP) e recorrendo a processos em diversas máquinas com memória distribuída (MPI).

3.1 LU Decomposition usando OpenMP

Na implementação da versão de memória partilhada deste algoritmo usando o mesmo algoritmo que na versão sequencial, deparei-me com um sério problema: o *loop* mais exterior não pode ser paralelizado pois cada operação num bloco depende do resultado da iteração anterior, o que resultava em matrizes com valores errados no output do programa. Assim, só foi possível paralelizar o segundo loop, com a instrução (no_threads corresponde ao número de threads dado como input do programa):

```
#pragma omp parallel for num_threads(no_threads) schedule(dynamic)
```

Note-se que apesar de não ter conseguido um aumento significativo de performance com este algoritmo, tal não acontecia com o algoritmo que tinha implementado anteriormente de Decomposição em LU com o nome de Dolittle. Por exemplo, para uma matrix de dimensão 2000x2000, obti 18 segundos para a versão sequencial e 13 segundos para a versão OpenMP. No

entanto, esta versão era no geral muito mais lenta que a versão atual, tendo sido essa a razão pelo qual foi descartada.

Os resultados obtidos usando a versão OpenMP estão apresentados na figura 4, tendo sido testado a execução do programa com vários tamanhos de matrizes e a correr o algoritmo principal com 1 a 6 threads.

3.2. LU Decomposition usando MPI

O algoritmo implementado nesta versão de memória distribuída deste algoritmo é exatamente igual aos usados nas versões anteriores, no entanto o código teve de ser remodelado de forma a permitir que o trabalho fosse dividido por todos os processos durante a execução do programa.

A maior diferença em relação às versões anteriores é que agora no loop em que são percorridas as linhas subsequentes à linha pivot (o segundo loop), o trabalho é dividido pelos vários processos, sendo que o processo com o id **rank** só executa as iterações desse loop em que se verifica que **i % size == rank**, sendo i o índice da iteração atual do ciclo e size o número total de processos que estão a ser executados.

Concluído o ciclo anterior, percorre-se novamente o mesmo loop percorrido anteriormente de forma a enviar e receber de todos os outros processos as linhas calculadas no ciclo anterior, através do uso de **MPI_Bcast**. Note-se que este método permite mandar a todos os outros processos a linha atual caso o processo em execução trata-se do processo raiz mencionado no **MPI_Bcast** (neste caso determinado por **i % size** que corresponde às linhas calculadas pelo processo atual) e, caso tal não se verifique, este método permite receber essas linhas dos outros processos. Desta forma o **MPI_Bcast** funciona simultaneamente como um recetor e emitente de dados, permitindo manter a matriz sincronizada em todos os processos.

Os resultados obtidos usando a versão MPI estão apresentados na figura 6. Os resultados como se pode verificar facilmente foram desastrosos e tentará-se explicar mais à frente algumas das razões pelas quais isto pode ter acontecido.

3.3. Sieve of Erathosthenes usando OpenMP

Ao implementar a versão OpenMP deste algoritmo usando um vector<bool> tal como na versão sequencial, deparei-me com um problema: não conseguia obter resultados corretos quando tentava aplicar qualquer tipo de paralelização ao algoritmo com as diretivas de OpenMP. Após uma pesquisa, encontrei que isto era de facto o comportamento “normal” do vector<bool> e por isso

troquei o vector de booleanos por um array de booleanos alocado dinamicamente para a versão OpenMP.

Em relação à versão OpenMP deste algoritmo, esta foi implementada usando exatamente o mesmo algoritmo que na versão sequencial (sendo a única diferença o facto do vector ter sido trocado por um array) e introduziu-se esta diretiva no ciclo mais exterior do algoritmo de forma a paralelizá-lo:

```
#pragma omp parallel for num_threads(no_threads) schedule(dynamic)
```

Os resultados obtidos usando a versão OpenMP estão apresentados na figura 5 e foram efetuados testes com 1 a 6 threads.

3.4. Sieve of Erathosthenes usando MPI

O algoritmo usado nesta versão é exatamente o mesmo que foi usado nas versões anteriores neste algoritmo, no entanto foi necessário remodelar o programa de forma a dividir o trabalho entre os diferentes processos.

Para tal, dividiu-se o vector pelo número total de processos e definiu-se os limites inferior e superior de cada vector de tal forma que cada processo apenas percorresse uma parte de todo o vector e determinasse os primos apenas para essa parte. Esta divisão de tarefas vai permitir otimizar muito a execução do loop interior do algoritmo já que para cada processo apenas é executado o bloco que lhe calhou nesse loop. Note-se que foi usado **MPI_Bcast** outra vez para sincronizar a execução do ciclo em todos os processos.

Também é importante mencionar que na impressão dos números primos no ficheiro, embora os números imprimidos sejam todos primos (ou seja, corretos), a ordem em que estes são imprimidos pode não estar correta. Isto porque não está definida a ordem em que os processos acabam.

Os resultados obtidos usando a versão MPI para este algoritmo estão apresentados na figura 7 e foram efetuados testes com 1 a 3 processos.

4. Avaliação de performance e análise de escalabilidade

Em relação às versões sequenciais implementadas para ambos os algoritmos, diria que as medidas de tempo de execução obtidas foram muito boas, especialmente para o algoritmo Sieve of Erathosthenes, em que foi aproveitada a implementação específica do vector<bool> para obter a melhor performance possível.

Sendo assim, considerando que as versões sequenciais já eram muito eficientes, foi difícil melhorar ainda mais com as versões paralelas dos algoritmos. Apesar disso, penso que foi possível obter bons resultados para todas as implementação menos a versão MPI de LU.

Em relação à versão OpenMP do algoritmo LU, é importante notar que embora o aumento de performance tenha sido muito ligeiro (tal como já tinha sido referido) no geral para qualquer número de threads, o número de threads que resultou num maior aumento de performance foi a do programa com 2 threads.

Em relação à versão OpenMP do algoritmo Sieve, os tempos de execução com qualquer número de threads teve resultados semelhantes. Note-se que apesar de não ter-se tido um aumento de performance com OpenMP, os tempos obtidos nesta versão são bastante razoáveis (no máximo ~10s mesmo para um número colossal como 600,000,000).

Em relação às versões MPI, é importante mencionar desde já que no meu computador cada vez que eu tentava correr qualquer programa MPI com mais de 2 processos dava um erro a dizer que não havia *slots* suficientes para correr o programa com o número especificado de processos. Para correr o programa com mais de 2 processos seria então necessário fazer –**oversubscribe** que geralmente resultava numa significativa degradação da performance.

Isto deve-se às especificações do meu computador em que se correu todos os testes, que já agora são: Intel® Core™ i7-4510U CPU @ 2.00GHz, 2 cores, 4 threads por core, CPU 2800MHz, L1d cache 32K, L1i cache 32K, L2 cache 256K e L3 cache 4096K.

O aumento/redução de performance através da utilização do MPI depende muito da forma em que é distribuído o trabalho pelos processos e até mesmo dos próprios algoritmos e tal é aparente comparando os resultados obtidos na versão MPI do algoritmo de decomposição em LU (pouco escaláveis, aumento significativo dos tempos de execução com um maior número de processos) com os resultados obtidos na versão MPI do algoritmo Sieve of Erathosthenes (muito escaláveis, o aumento dos tempos de execução com um maior número de processadores deve-se maioritariamente ao overhead introduzido pelo **–oversubscribe** e por aí fora).

Eu diria que a razão pela qual a minha versão MPI do algoritmo LU tem uma performance tão abismal é maioritariamente por causa do uso intensivo do **MPI_Bcast** que aumenta quanto maior for o número de processos.

Por último, em relação à escalabilidade dos vários programas implementados diria que todos os programas implementados (menos o LU MPI) são bastante escaláveis, principalmente a versão sequencial e MPI do algoritmo Sieve of Erathosthenes.

5. Referências

- http://www.personal.psu.edu/jhm/f90/lectures/lu.html
- http://www.cl.cam.ac.uk/teaching/1314/NumMethods/supporting/mcmaster-kiruba-ludecomp.pdf
- https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
- https://stackoverflow.com/questions/33617421/write-concurrently-vectorbool

6. Anexos

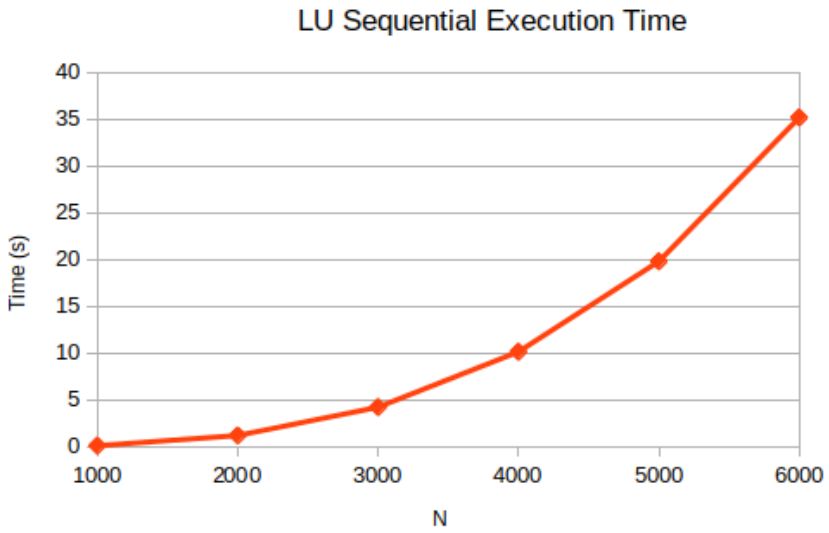

Figura 2. Tempos obtidos para a execução da versão sequencial do algoritmo LU Decomposition

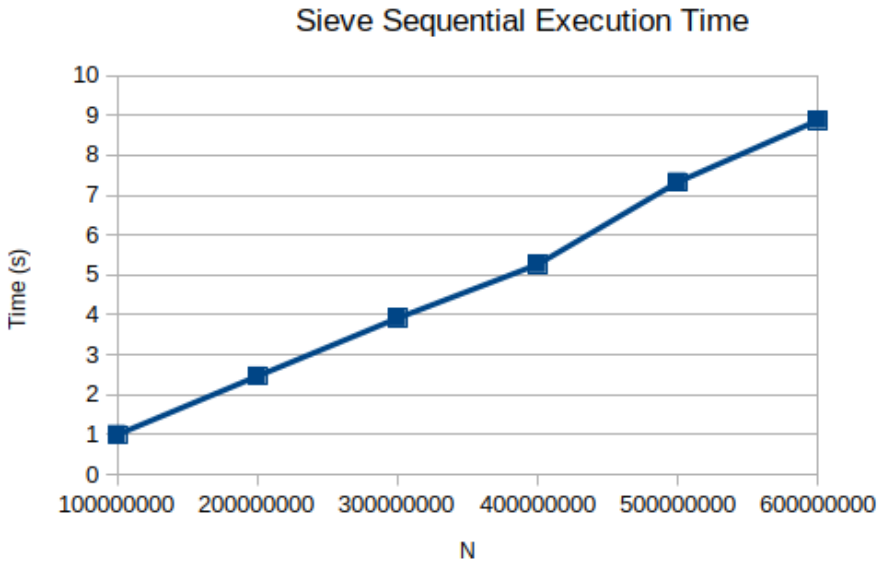

Figura 3. Tempos obtidos para a execução da versão sequencial do algoritmo Sieve of Erathosthenes

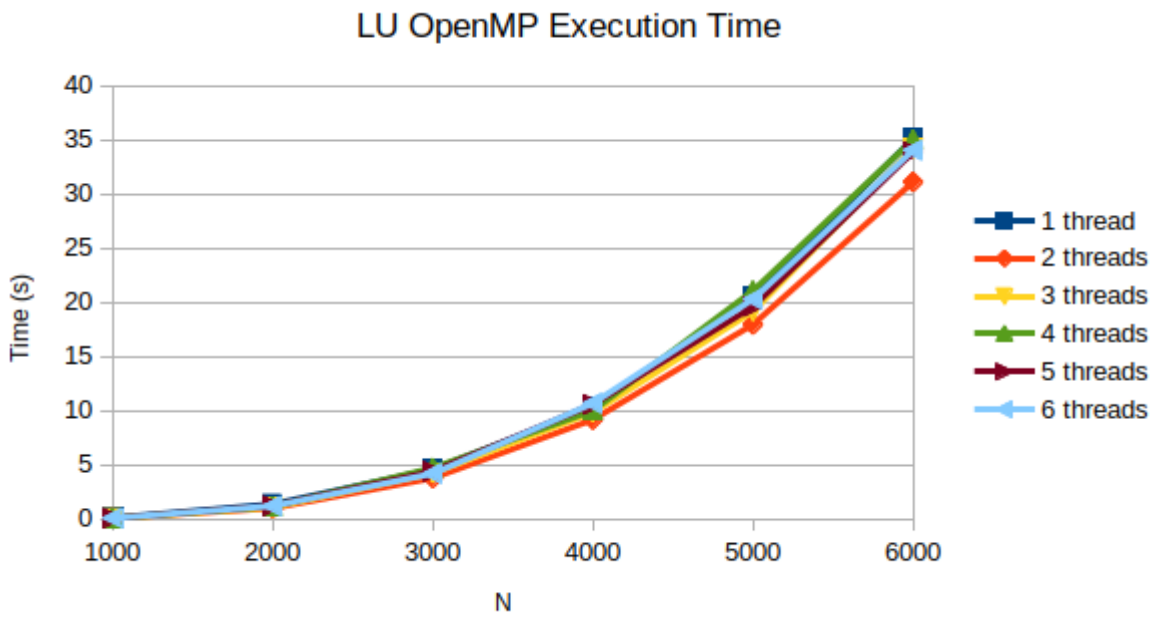

Figura 4. Tempos obtidos para a execução da versão OpenMP do algoritmo LU Decomposition

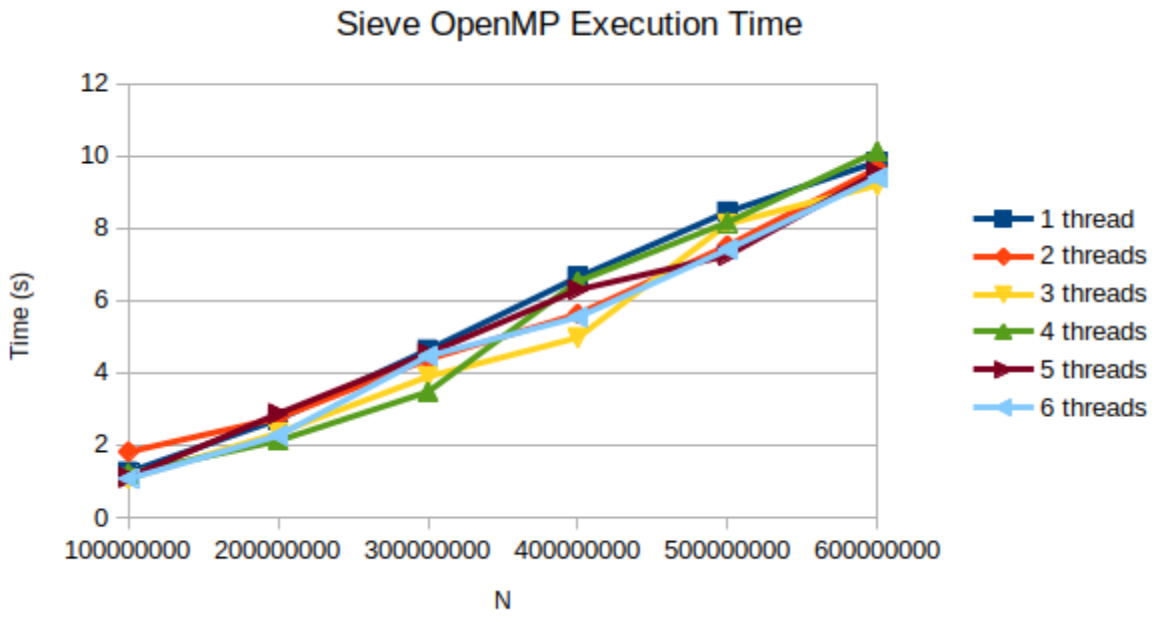

Figura 5. Tempos obtidos para a execução da versão OpenMP do algoritmo Sieve of Erathosthenes

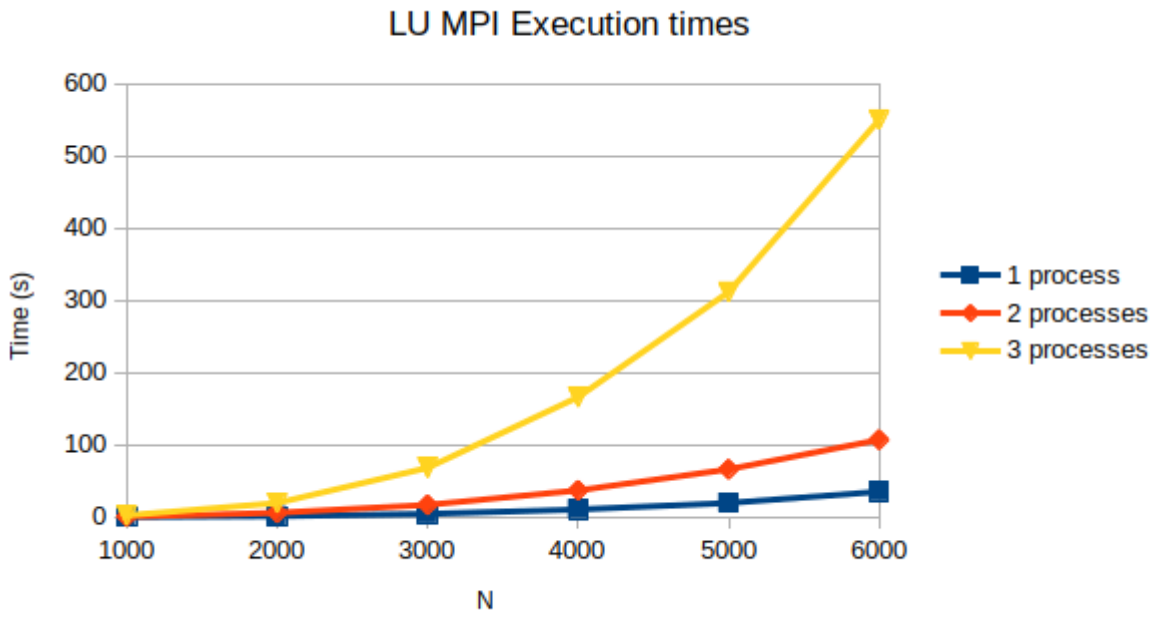

Figura 6. Tempos obtidos para a execução da versão MPI do algoritmo LU Decomposition

Figura 7. Tempos obtidos para a execução da versão MPI do algoritmo Sieve of Erathosthenes