Análise de Paralelismo
Fatoração LU

Paulo Alexandre Piornedo Panucci[1]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)

ra88380@uem.br

Abstract. *A sequential code can be written to be parallel executed in a set of threads or process, with the purpose of taking advantage of the core quantity in a machine, or more than one machine, making the program execution faster. It is expected that making a program parallel, it has a significantly better execution time than the sequential program, although there are cases in which the parallelized code has a similar or worst performance compared to the sequential code. In this work will be analyzed the parallel version of LU decomposition algorithm to matrix decomposition.*

Resumo. *Um código sequencial pode ser escrito para ser executado paralelamente em um conjunto de threads ou processos, com finalidade de tirar proveito da quantidade de núcleos presentes em uma máquina, ou de mais de uma máquina, e assim tornar a execução do programa mais rápida. Espera-se que ao paralelizar um determinado programa, ele possua um tempo significantemente melhor que o código sequencial, mas também existem casos em que um código paralelizado tem um desempenho próximo, ou pior, que o sequencial. Neste trabalho será analisada a paralelização do algoritmo Fatoração LU para decomposição de matrizes.*

1. Introdução

Como observado por [Aubanel 2016], antes da revolução dos processadores *multi-core*, o programador dependia das melhoras de performance apresentadas por processadores conforme o lançamento de novas gerações. Com a estagnação do aumento da taxa de *clock*, poder utilizar múltiplos *cores* para conseguir rodar um programa tende a tornar a execução mais eficiente dependendo como este problema é modelado.

Hoje conta-se com diretivas de compilação e APIs para paralelizar um código. Um exemplo é o *OpenMP* para a linguagem C/C++. Apesar de existirem alternativas de deixar o trabalho de paralelização para o compilador, aprender a escrever códigos paralelos distribuídos tem suas vantagens. [Aubanel 2016] ainda diz que os compiladores tentam otimizar o código que é dado a eles, porém eles não reescrevem esse código para que ele realmente seja paralelo. Desta forma pode-se dizer que existe uma diferença entre paralelizar um código em C com o OpenMP e escrever um código paralelo utilizando a biblioteca Pthread do C.

As ferramentas citadas acima tem por objetivo auxiliar na escrita de códigos paralelos em memória compartilhada. Para a memória distribuída, pode-se utilizar ferramentas como o MPI (*Message Passing Interface*).

2. Fundamentação teórica

Este trabalho abordará a análise de paralelismo do algoritmo da fatoração LU. Desta forma, é importante se conhecer o objetivo desta decomposição, assim como alguns conceitos básicos de paralelismo para se entender esta análise.

2.1. fatoração LU

A fatoração LU consiste em: dada uma matriz A não singular, existe uma matriz triangular inferior L (*lower*) tal que multiplicada pela matriz triangular superior U (*upper*) o resultado seja A. Logo, esta decomposição pode ser definida pela seguinte fórmula:

$$A = \mathrm{L} \times \mathrm{U}.$$

Uma matriz singular consiste em uma matriz que não admite inversa. Logo a matriz não singular admite a sua inversa.

2.2. Paralelismo

Serão abordados agora alguns conceitos básicos importantes para se entender o paralelismo a nível de threads e processos e sua análise.

2.2.1. *Speedup*

Ao se escrever um código paralelo a partir de um sequencial, espera-se que seu código paralelo tenha um desempenho de tempo que se aproxima de N vezes melhor que o tempo do código sequencial, sendo este N o número de threads ou processos que serão executadas em paralelo, e este valor é dado pelo *speedup*.

Quando dividimos o tempo de execução do algoritmo sequencial pelo tempo de execução do algoritmo paralelo, tem-se então o seu *speedup*:

$$S_p = \frac{T_s}{T_p}.$$

É importante distinguir o tempo decorrido da execução do código, do tempo total levado pela soma dos tempos de execução de cada *thread* ou processo. Para fim de análise de speedup precisa-se do *elapsed time*, que é o tempo decorrido do começo ao fim do programa (ou de uma função paralela específica).

Quando o *speedup* de um determinado programa paralelo possui um valor maior que o número de *threads* ou processos que executa este código, tem-se então um *speedup* dito superescalar.

2.2.2. Condição de corrida

Ao se escrever um código paralelo, quando *threads* ou processos simultâneas podem competir por algum recurso diz-se que existe condição de corrida neste algoritmo. Pode-se definir como recursos acesso em uma determinada região da memória, e até mesmo o próprio hardware em que o algoritmo está sendo executado. Como um exemplo de disputa de recursos, pode se dar o exemplo que uma determinada *thread* ou processo queira escrever ao mesmo tempo que outra em uma região da memória, ou também a primeira quer ler um resultado que está ao mesmo tempo sendo escrita pela segunda. Como exemplo

de disputa de recurso de *hardware*, uma quantidade x de *threads* ou processos que estão prontos para serem executados, porém a máquina contém $\frac{x}{2}$ núcleos de processamento.

Para se resolver problemas de condição de corrida, principalmente de software, deve se usar recursos como *locks* e barreiras ou mensagens síncronas.

Se existe alguma região crítica em que somente uma *thread* pode estar executando essa região, pode-se utilizar *lock* para não deixar que outras *threads* acessem essa área e *unlock* para liberar para as outras.

Ao se procurar por sincronia em um algoritmo, opta-se pelo uso de barreiras. Quando uma barreira é colocada no algoritmo, isso implica que todas as *threads* devem chegar na mesma região para assim poder dar continuidade na execução.

As mensagens síncronas em programas escritos para memória distribuída acaba fazendo o papel do *lock* e da barreira, apesar de tais estruturas poderem ser utilizadas.

Ter condições de corrida no algoritmo que deseja paralelizar implica em diminuir a possibilidade de ter um *speedup* próximo a quantidade de *threads* ou processos que estão executando. Isto se dá pelo fato de impedir que estes dependam deles mesmos para executar o trabalho que a eles foi dado.

2.2.3. Balanceamento de carga

Independente de como é feita a implementação de um código paralelo, é interessante que os processos ou *threads* tenham um balanceamento execução, ou seja, façam quantidades de trabalho parecidas, para assim alcançar uma melhor performance.

2.3. Ferramentas

Serão apresentadas algumas ferramentas utilizadas para o desenvolvimento do algoritmo paralelo e sua análise.

2.3.1. Polybench

O Polybench é uma coleção de *benchmarks* contendo partes de controle estático, com o propósito de uniformizar a execução e o monitoramento de kernels.

Desta coleção de *benchmarks* que se teve como base de analise o algoritmo de fatoração LU.

2.3.2. Pthread

Pthread é uma biblioteca para a linguagem C em sistemas *UNIX* que padroniza a programação em *threads* nesse ambiente.

Ao se tratar de uma biblioteca de criação de *threads*, fica a cargo do programador em ditar a maneira como a paralelização funcionará. Desta forma deve-se criar *threads* e passar uma função como argumento para ser executada. Deve-se também fazer a junção dessas *threads* e se precisar, matá-las.

2.3.3. Open MPI

O Open MPI é uma implementação *open source* de interface de passagem de mensagem, MPI, desenvolvido por um conjunto de acadêmicos, pesquisadores e indústrias.

3. Proposta

3.1. Primeira parte do trabalho

A primeira parte deste trabalho consiste em desenvolver um algoritmo que realize a fatoração LU em uma arquitetura de memória distribuída utilizando a ferramenta Open MPI. Assim, comparar seu desempenho com o desempenho do algoritmo sequencial e com o algoritmo paralelo para memória compartilhada utilizando a biblioteca Pthread.

A comparação entre estes algoritmos será mostrada através da análise de *speedup* explicada na subseção 2.2

O *kernel* do algoritmo sequencial tem a seguinte estrutura:

```
static void kernel_lu(){
  int i, j, k;

  for (k = 0; k < size; k++){
    for (j = k+1; j < size; j++){
      I[k][j] = I[k][j] / I[k][k];
    }
    for(i = k+1; i < size; i++){
      for (j = k + 1; j < size; j++){
        I[i][j] -= I[i][k] * I[k][j];
      }
    }
  }
}
```

O *kernel* do algoritmo Pthread tem a seguinte estrutura:

```
static void *kernel_lu(void *arg){
  int i, j, k;

  int id = *((int *)arg);
  int start = id * cut;
  int end = minimum(start + cut, size);

  for (k = 0; k < size; k++){
    int gap = maximum(k + 1, start);
    for (i = gap; i < end; i++){
      I[i][k] /= I[k][k];
    }
    pthread_barrier_wait(&barrier);
    for(i = gap; i < end; i++){
      for (j = k + 1; j < size; j++){
        I[i][j] -= I[i][k] * I[k][j];
      }
    }
    pthread_barrier_wait(&barrier);
  }

```

```
  return NULL;
}
```

O *kernel* do algoritmo MPI tem a seguinte estrutura:

```
void ludec_mpi(){

  MPI_Status status;


  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  /*Distribute tasks*/
  if(world_rank == 0){
    populate2Dmatrix(I, size);

    for(int i = 1; i < world_size; i++){
      for(int j = 0; j < size; j++){
        for(int k = 0; k < size; k++){
          MPI_Send(&I[j][k], 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
        }
      }
    }

    // printf("world_rank(%d) distributing the matrix: \n", world_rank)
    ;
    // printMatrix(I, size);
  }
  /*Receiving tasks*/
  else{
    for(int i = 0; i < size; i++){
      for(int j = 0; j < size; j++){
        MPI_Recv(&I[i][j], 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
      }
    }
    // printf("world_rank(%d) receiving the matrix: \n", world_rank);
    // printMatrix(I, size);
  }
  start = MPI_Wtime();
  /*Executing Kernel*/
  for(int k = 0; k < size; k++) {
    MPI_Bcast(I[k], size, MPI_DOUBLE, k % world_size, MPI_COMM_WORLD);
    for(int i = k + 1; i < size; i++) {
      if(i % world_size == world_rank){
        double l = I[i][k] / I[k][k];
        for (int j = k; j < size; j++){
          I[i][j] -= l * I[k][j];
        }
      }
    }
  }
  end = MPI_Wtime();
  /*Distributing Results*/
```

```
50    int param = (size % world_size - 1 + world_size ) % world_size;
51    if(param != 0) {
52      if(world_rank == param) {
53        for (int i = 0; i < size; i++){
54          MPI_Send(I[i], size, MPI_DOUBLE, 0, size + 1, MPI_COMM_WORLD);
55        }
56      }
57      if(world_rank == 0) {
58        for(int i = 0; i < size; i++){
59          MPI_Recv(I[i], size, MPI_DOUBLE, param, size + 1,
     MPI_COMM_WORLD, &status);
60        }
61      }
62    }
63  }
```

3.2. Segunda parte do trabalho

A segunda parte do trabalho consiste em organizar uma arquitetura em *grid* utilizando 13 computadores. a figura 1 mostra a organização proposta onde os números nos vértices representam os *world ranks* já definidos para exercer cada função. As informações das arestas definem os comunicadores entre cada par de vértice. Cada par de vértice tem seu próprio comunicador.

A maquina com *world rank* 9 será a requisitora. Ela será responsável por requisitar a fatoração LU de matrizes de variadas dimensões. Assim, este computador enviará as matrizes para resolução ao *grid*, sendo o nó de acesso o computador com *world rank* 0.

Do *world rank* 0 ao 8, os computadores serão responsáveis por fazer o transporte da mensagem até os nós de processamento. A troca de mensagens de um nó ao outro será de forma randômica, com o intuito de analisar os caminhos realizados entre o *grid* até as máquinas processadoras. Como o objetivo é alcançar às máquinas que resolvem o algoritmo, definiu-se que quando um nó nesta malha recebe uma matriz, ele deve mandar para outro nó diferente daquele que enviou os dados pra ele.

Os *world ranks* 10, 11 e 12 são as máquinas que realizam a computação da fatoração LU. Após a computação, esses nós devolvem a matriz fatorada para seus nós adjacentes, e esta matriz fará o caminho mostrado pela figura 2 até o nó requisitor. Após atender todas as requisições, o *world rank* 9 envia uma mensagem, que se espalhará pela malha conforme a figura 3, para terminar o algoritmo.

Para análise do comportamento do *grid* será gerado um *log* de uma execução com 4 requisições, com dimensões de 64, 128, 256 e 512. Para exemplificação na subseção 5.2 será mostrado um *log* reduzido de 3 requisições, com dimensões de 64, 128, e 256.

Quando uma matriz chega a um nó ”de borda de processador”(nós 2, 6 e 8), este vértice verifica se o processador está livre. Caso esteja ocupado, a matriz continua andando na malha de forma randômica. Caso contrário, a matriz é mandada para o processador adjacente.

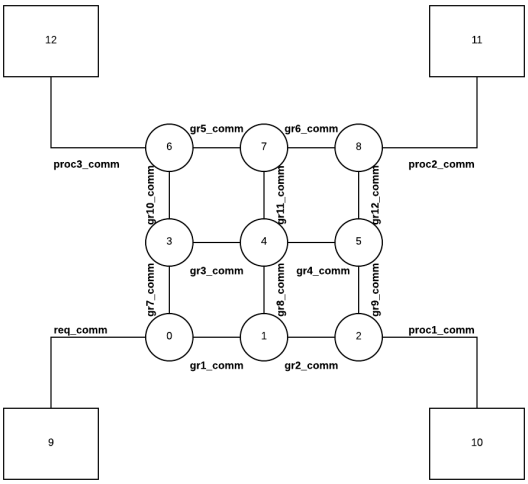

Figura 1. Formação do *grid*.

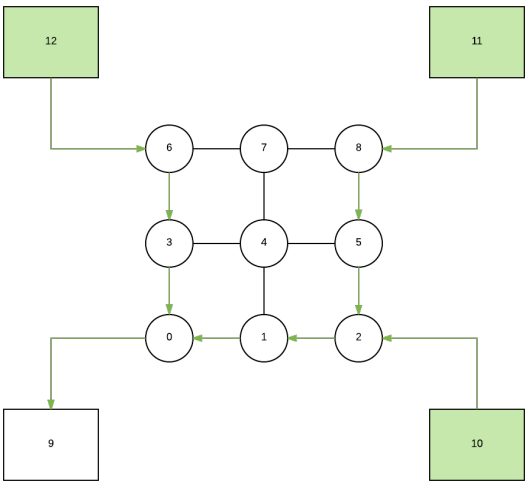

Figura 2. Percurso da mensagem após processamento até o requisitor.

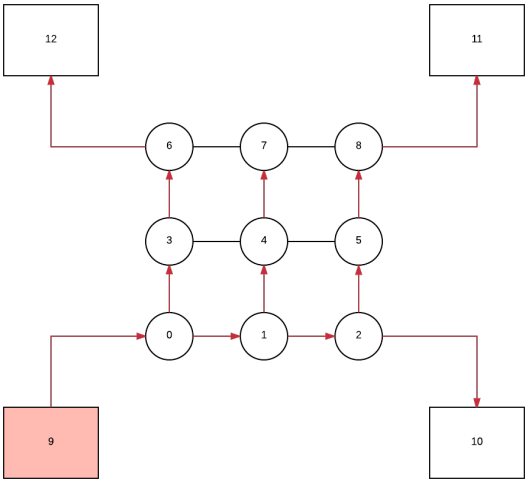

Figura 3. Percurso da mensagem de finalização do algoritmo.

4. Ambiente experimental

Foram utilizadas as máquinas presentes no laboratório LIN 02 (Laboratório de Informática 02) do Departamento de Informática da Universidade Estadual de Maringá. Nelas foram rodadas as execuções sequenciais, Pthread e MPI para uma análise justa.

para cada algoritmo foram realizadas 11 execuções, em que o tempo de execução da primeira é descartado, e entre as outras 10, é retirado o menor e o maior tempo, calculando a média do tempo das execuções restantes.

O algoritmo Pthread foi executado para 2, 4, 8 e 16 *threads*. O algoritmo MPI foi executado para 2, 4, 8 e 16 processos. Por problemas de infraestrutura não foi possível executar cada processo do MPI em máquinas diferentes. o LIN 02 só permitia rodar o códio MPI em até 3 máquinas diferentes, impossibilitando a execução para 4 ou mais máquinas. Assim foi utilizado mais processos por máquina, sendo utilizado somente 3 máquinas para todas as execuções.

5. Resultados

5.1. Primeira parte do trabalho

O gráfico apresentado pela figura 4 mostra o *speedup* do algoritmo Pthread e MPI. O resultado do Pthread se dá pelo fato de existirem dois pontos de sincronização com barreira dentro laço de repetição principal do *kernel* apresentado na subseção 3.1. Para a fatoração ocorrer corretamente, Antes do algoritmo realizar a etapa, chamada de etapa de eliminação, apresentada na linha 16, todas as *threads* devem ter terminado a primeira etapa. Isso acontece pois esta fase escreve resultados em posições utilizadas para o calculo da primeira parte. A ultima barreira na linha 19 sincroniza as *threads* para iniciarem juntas uma nova iteração do laço de repetição principal.

Dada a dimensão de uma matriz para a fatoração LU, existirá um número de sincronização duas vezes maior. Como citado na subseção 2.2.2, muita condição de corrida, que necessita de sincronização, atrapalha no desempenho do código sequencial.

Era esperado um melhor resultado de *speedup* do código MPI, pois a necessidade de sincronização constante com barreira é substituída pelas mensagens síncronas. Nota-se um bom *speedup* para dois processos. Isto acontece pois conseguiu-se utilizar dois computadores distintos para executar esses processos. Como explicado na seção 4 por problemas de infraestrutura só se conseguia executar o algoritmo MPI para até 3 computadores distintos. A interface de passagem de mensagem foi desenvolvida para sanar o problema de troca de mensagens em arquitetura de memória distribuída. Apesar de se conseguir executar para vários processos em uma única máquina, haverá a troca de mensagens entre processos que tem acesso à mesma memória, gerando informações redundantes, assim encarecendo desnecessariamente a execução do algoritmo.

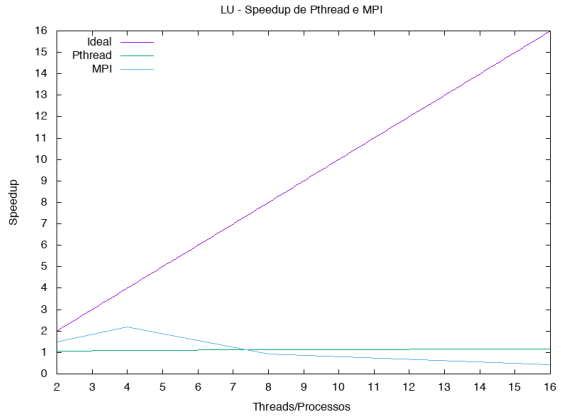

Figura 4. *Speedup* do algoritmo Pthread e MPI.

5.2. Segunda parte do trabalho

5.2.1. *Log* de troca de mensagens no *grid*

Requisitor 9:

REQUISITOR (WR: 9) - ENVIANDO REQUISIÇÃO [numero de requisição: 1 tamanho 256]

REQUISITOR (WR: 9) - ENVIANDO REQUISIÇÃO [numero de requisição: 2 tamanho 128]

REQUISITOR (WR: 9) - ENVIANDO REQUISIÇÃO [numero de requisição: 3 tamanho 64]

REQUISITOR (WR: 9) - REQUISIÇÃO ATENDIDA [numero de requisição: 1 tamanho 256]

REQUISITOR (WR: 9) - REQUISIÇÃO ATENDIDA [numero de requisição: 3 tamanho 64]

REQUISITOR (WR: 9) - REQUISIÇÃO ATENDIDA [numero de requisição: 2 tamanho 128]

REQUISITOR (WR: 9) - TODAS AS REQUISIÇÕES FORAM ATENDIDAS - ENVIANDO MENSAGEM DE FIM

REQUISITOR (WR: 9) - TODAS AS REQUISIÇÕES FORAM ATENDIDAS - PARANDO AS ATIVIDADES NO REQUISITOR

NÓ 0 DO GRID:

GRID (WR: 0) - RECEBENDO REQUISIÇÃO DE 9 [numero de requisição: 1 tamanho 256]

GRID (WR: 0) - RETRANSMITINDO REQUISIÇÃO PARA 3 [numero de requisição: 1 tamanho 256]

GRID (WR: 0) - RECEBENDO REQUISIÇÃO DE 9 [numero de requisição: 2 tamanho 128]

GRID (WR: 0) - RETRANSMITINDO REQUISIÇÃO PARA 3 [numero de requisição: 2 tamanho 128]

GRID (WR: 0) - RECEBENDO REQUISIÇÃO DE 9 [numero de requisição: 3 tamanho 64]

GRID (WR: 0) - RETRANSMITINDO REQUISIÇÃO PARA 1 [numero de requisição: 3 tamanho 64]
GRID (WR: 0) - RETORNANDO REQUISIÇÃO ATENDIDA [numero de requisição: 1 tamanho 256]
GRID (WR: 0) - RETORNANDO REQUISIÇÃO ATENDIDA [numero de requisição: 3 tamanho 64]
GRID (WR: 0) - RETORNANDO REQUISIÇÃO ATENDIDA [numero de requisição: 2 tamanho 128]
GRID (WR: 0) - PARANDO AS ATIVIDADES NO GRID

NÓ 1 DO GRID:
GRID (WR: 1) - RECEBENDO REQUISIÇÃO DE 0 [numero de requisição: 3 tamanho 64]
GRID (WR: 1) - RETRANSMITINDO REQUISIÇÃO PARA 2 [numero de requisição: 3 tamanho 64]
GRID (WR: 1) - RETORNANDO REQUISIÇÃO ATENDIDA [numero de requisição: 3 tamanho 64]
GRID (WR: 1) - PARANDO AS ATIVIDADES NO GRID

NÓ 2 DO GRID:
GRID (WR: 2) - RECEBENDO REQUISIÇÃO DE 1 [numero de requisição: 3 tamanho 64]
GRID (WR: 2) - PROCESSADOR 10 PRONTO PARA ATENDER A REQUISIÇÃO [numero de requisição: 3 tamanho 64]
GRID (WR: 2) - RETORNANDO REQUISIÇÃO ATENDIDA [numero de requisição: 3 tamanho 64]
GRID (WR: 2) - PARANDO AS ATIVIDADES NO GRID

NÓ 3 DO GRID:
GRID (WR: 3) - RECEBENDO REQUISIÇÃO DE 0 [numero de requisição: 1 tamanho 256]
GRID (WR: 3) - RETRANSMITINDO REQUISIÇÃO PARA 6 [numero de requisição: 1 tamanho 256]
GRID (WR: 3) - RECEBENDO REQUISIÇÃO DE 0 [numero de requisição: 2 tamanho 128]
GRID (WR: 3) - RETRANSMITINDO REQUISIÇÃO PARA 6 [numero de requisição: 2 tamanho 128]
GRID (WR: 3) - RETORNANDO REQUISIÇÃO ATENDIDA [numero de requisição: 1 tamanho 256]
GRID (WR: 3) - RETORNANDO REQUISIÇÃO ATENDIDA [numero de requisição: 2 tamanho 128]
GRID (WR: 3) - PARANDO AS ATIVIDADES NO GRID

NÓ 4 DO GRID:
GRID (WR: 4) - PARANDO AS ATIVIDADES NO GRID

NÓ 5 DO GRID:

GRID (WR: 5) - PARANDO AS ATIVIDADES NO GRID

NÓ 6 DO GRID:

GRID (WR: 6) - RECEBENDO REQUISIÇÃO DE 3 [numero de requisição: 1 tamanho 256]

GRID (WR: 6) - PROCESSADOR 11 PRONTO PARA ATENDER A REQUISIÇÃO [numero de requisição: 1 tamanho 256]

GRID (WR: 6) - RETORNANDO REQUISIÇÃO ATENDIDA [numero de requisição: 1 tamanho 256]

GRID (WR: 6) - RECEBENDO REQUISIÇÃO DE 3 [numero de requisição: 2 tamanho 128]

GRID (WR: 6) - PROCESSADOR 11 PRONTO PARA ATENDER A REQUISIÇÃO [numero de requisição: 2 tamanho 128]

GRID (WR: 6) - RETORNANDO REQUISIÇÃO ATENDIDA [numero de requisição: 2 tamanho 128]

GRID (WR: 6) - PARANDO AS ATIVIDADES NO GRID

NÓ 7 DO GRID:

GRID (WR: 7) - PARANDO AS ATIVIDADES NO GRID

NÓ 8 DO GRID:

GRID (WR: 8) - PARANDO AS ATIVIDADES NO GRID

PROCESSADOR 10:

PROCESSADOR (WR: 10) - PROCESSANDO REQUISIÇÃO [numero de requisição: 3 tamanho 64]

PROCESSADOR (WR: 10) - TODAS AS REQUISIÇÕES FORAM PROCESSADAS - PARANDO AS ATIVIDADES NO PROCESSADOR

PROCESSADOR 11:

PROCESSADOR (WR: 11) - TODAS AS REQUISIÇÕES FORAM PROCESSADAS - PARANDO AS ATIVIDADES NO PROCESSADOR

PROCESSADOR 12:

PROCESSADOR (WR: 12) - PROCESSANDO REQUISIÇÃO [numero de requisição: 1 tamanho 256]

PROCESSADOR (WR: 12) - PROCESSANDO REQUISIÇÃO [numero de requisição: 2 tamanho 128]

PROCESSADOR (WR: 12) - TODAS AS REQUISIÇÕES FORAM PROCESSADAS - PARANDO AS ATIVIDADES NO PROCESSADOR

6. Conclusão

A análise feita neste trabalho mostra um desempenho abaixo do esperado para o algoritmo de fatoração LU utilizando a biblioteca Pthread para memória compartilhada.

Esperava-se um bom desempenho de *speedup* para o algoritmo MPI, mas foi concluído que esta biblioteca foi desenvolvida para ter um melhor desempenho quando se tem um processo em cada máquina que faz parte da execução. Assim, é interessante o uso de várias máquinas e um algoritmo para memória distribuída para a execução desta fatoração, sendo um fator limitante para isso o custo de se ter várias máquinas independentes para este cálculo.

O *grid* modelado para a segunda parte deste trabalho teve o comportamento esperado de acordo com a proposta da subseção 3.2, provado pelo *log* apresentado na subseção 5.2.

Referências

Aubanel, E. (2016). *Elements of Parallel Computing*. Chapman and Hall/CRC.