Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
SSC0143 - Programação Concorrente

Atividade: Implementação do Método de Gauss-Jordan

Profº: Paulo Sérgio Lopes de Souza

Alunos:

Bernardo Simões Lage Gomes Duarte (8598861)

Giovani Ortolani Barbosa (8936648)

Jorge Luiz da Silva Vilaça (9066491)

Luiz Augusto Vieira Manoel (8937308)

Novembro de 2017

Introdução

Neste projeto, implementamos o método de Gauss - Jordan utilizando as ferramentas de programação concorrente OpenMP e MPI. Este foi feito com base no primeiro projeto da disciplina de Programação Concorrente, considerando também os comentários dos corretores.

Na primeira entrega, utilizamos o método PCAM para modelar uma solução paralela para o método de Gauss-Jordan. Este método é utilizado para escalonar matrizes, aplicando operações elementares à matriz aumentada de um sistema de equações lineares. Os sistemas com os quais trabalhamos no projeto possuem o mesmo número de equações e variáveis e têm solução possível e única. O algoritmo transforma a matriz entrada em uma uma matriz identidade, e como resíduo no array de entrada se obtém a solução.

Nessa etapa do projeto, explicamos nas sessões abaixo, primeiro como a nossa implementação foi feita, baseada na modelagem PCAM apresentada no trabalho anterior. Falamos em seguida das dificuldades encontradas pelo grupo ao implementar o método. Mostramos os resultados obtidos com a nossa implementação sendo executada no *cluster* do LaSDPC. Por fim, apresentamos as conclusões do trabalho.

Método e Organização

De maneira resumida, na primeira etapa do trabalho, mapeamos o problema da eliminação de Gauss-Jordan da seguinte maneira: propomos a distribuição de colunas da matriz aumentada entre os *n* processos.

Para implementação do código usamos as linguagens C, openMP e openMPI. Assumimos que como entrada receberemos uma matriz quadrada e um vetor que tenham dimensão divisível pela quantidade de processos. Essa última condição é considerada para que a rotina de comunicação coletiva MPI_Scatter funcione da maneira esperada.

Sendo p_0, p_1, …, p_n os *n* processos criados pelo MPI, a organização foi feita da seguinte maneira:

- O p_0 é o único processo que conhece a matriz quadrada inteira;
- O p_0 é o único processo que conhece que conhece o vetor de entrada;
- As colunas da matriz quadrada foram divididas igualmente entre os *n* processos usando a rotina MPI_Scatter a partir de p_0;
- A solução é calculada dentro de um loop principal percorrendo os pivô;
- O processo p_x que for responsável por um pivô se encarrega de:
    - Verificar a necessidade da troca entre 2 linhas (acontece quando o pivô for igual zero) e descobrir onde está o pivô;
    - Envia um broadcast aos demais processos anunciando a posição do pivô;

    - Envia um broadcast da coluna inteira que possui o pivô para os outros processos.
- A partir disso, todos processos:
    - Utilizam a informação da posição do pivô para realizar a troca entre linhas quando necessário;
    - Conseguem descobrir descobrir qual o pivô com a coluna dele, para então realizar a operação de pivoteamento das suas colunas;
    - Realizam escalonamento utilizando informação da coluna do pivô para calcular.
- O p_0, também realiza todas as operações com o vetor de entrada, que ao fim vai possuir a solução.

Ao ao fim da solução da eliminação de Gauss-Jordan não é necessário executar a rotina MPI_Gather, uma vez que saber o resultante da matriz usada no cálculo não é parte da solução. Além disso, é conhecido que o resultado dessa matriz é a identidade.

OpenMP foi usado na implementação para paralelizar em threads as operações de pivoteamento e escalonamento, mesmo o grupo acreditando que o pivoteamento é uma operação simples suficiente para que haja queda de desempenho com uso de openMP.

Compilação e Execução

A compilação do programa é feita na pasta do projeto, basta inserir o comando `make`.

A execução do programa pode ser feita de duas maneiras:

1. Utilizando dois arquivos .txt nomeados `matrix.txt` e `vetor.txt` que são respectivamente, a matriz A e o vetor coluna B do sistema. Os arquivos devem seguir o modelo descrito na especificação do Projeto e devem estar inseridos no diretório `in/`. Dentro da pasta do projeto inserir o seguinte comando:

```
mpirun -np <N> ./bin/gaussjordan <OPENMP>
```

2. Através de matrizes criadas com valores aleatórios entre 0 e 49. Deve-se estar na pasta do projeto e inserir o seguinte comando:

```
mpirun -np <N> ./bin/gaussjordan <OPENMP> <DIM>
```

O resultado de ambos os modos de execução irá ser salvo em um arquivo .txt chamado `resultado.txt` e estará contido no diretório `out/`. Para verificar apenas os tempos de execução sugere-se comentar a linha 113 do arquivo `main.c`.

Em que:

`<DIM> = dimensão da matriz, <N> = número de processos, <OPENMP> = número de threads (se OPENMP == 0, significa não usar OPENMP)`

Para executar o programa no *cluster* basta executar o comando `make` no diretório `home` e em seguida executar o comando `./test.sh`.

Atenção: a dimensão da matriz deve ser divisível pelo número de processos.

Dificuldades

Durante a realização do trabalho o grupo enfrentou algumas dificuldades para o entendimento e implementação da solução. A organização da matriz em 1 dimensão fez com que a utilização de índices deixasse de ser trivial, além de ter exigido um esforço maior do grupo para entender a comunicação coletiva enviando colunas. Além disso, a presença do vetor na matriz aumentada faria com que a divisão das colunas por scatter não fosse ideal, levando a decisão de que o processo 0 seja responsável pelo vetor.

Resultados

Através dos testes realizados no *clusters* obtivemos os resultados apresentados na Imagem 1. Não consideramos as abordagens com matrizes de dimensões 5.000 e 10.000, pois seu tempo de execucação era alto. Além disso, utilizamos a segunda maneira apresentada ao executar o programa.

```
NPes = 1, Dim = 1000, Time = 18.531391, NThreads = 0
NPes = 2, Dim = 1000, Time = 11.861906, NThreads = 0
NPes = 4, Dim = 1000, Time = 10.879064, NThreads = 0
NPes = 8, Dim = 1000, Time = 6.650791, NThreads = 0
NPes = 2, Dim = 1000, Time = 13.134857, NThreads = 4
NPes = 4, Dim = 1000, Time = 14.211064, NThreads = 4
NPes = 8, Dim = 1000, Time = 10.793445, NThreads = 4
NPes = 2, Dim = 1000, Time = 15.065595, NThreads = 8
NPes = 4, Dim = 1000, Time = 17.257299, NThreads = 8
NPes = 8, Dim = 1000, Time = 12.634759, NThreads = 8
```

Imagem 1 - comparação dos tempos de execuação para diferentes parâmetros (NPes = nº de processos; dim = dimensão; time = tempo decorrido; NThreads = nº de threads)

É notável o ganho de desempenho ocorrido ao utilizarmos mais processos. Com o aumento do número de *threads* também observa-se um maior tempo de execuação devido ao *overhead* ocorrido.

Conclusão

Com os resultados adquiridos com esse trabalho pudemos perceber que o desempenho melhorou com o aumento de processos executando o código.

Por outro lado, o uso de openMP, tanto com 4, quanto com 8 *threads*, e mesmo com com uma quantidade maior de dados fez com que o desempenho piorasse. Acreditamos que esse resultado é fruto do *overhead* da comunicação é maior do que o ganho com a paralelização das threads.

Em geral, o ganho de desempenho é maior para uma quantidade maior de dados.