UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO

RELATÓRIO
Perfilamento

Alexandre Henrique Soares Dias: 20180001100

José Tobias Souza dos Santos: 20180154969

Gabriel Medeiros Coelho: 20180009538

Natal-RN

2019

1. Introdução

1.1 Decomposição LU

A Decomposição ou fatorização Lower-Upper (LU) é comumente aplicada na computação para resolução de sistemas lineares quadrados, e ainda, é utilizado como um dos passos intermediários para obtenção da inversa de uma matriz e cálculo de determinantes. A decomposição LU fatora uma matriz *A* em um produto de duas submatrizes, uma triangular inferior e outra triangular superior.

1.2 Aplicação do splash2 - *LU Contiguous-Blocking Decomposition* (lu_cb)

Na aplicação LU do splash2, a matriz densa *A* (*n x n)* é dividida em *arrays N x N* de blocos *B x B* (*n = NB*) para explorar localidade temporal nos elementos das submatrizes. O tamanho do bloco deve ser grande o suficiente para manter a taxa de *miss* de cache baixa, e pequena o suficiente para manter uma bom balanceamento de carga. Elementos em um bloco são alocados contiguamente para aumentar os benefícios da localidade espacial, e os blocos são alocados localmente para os processadores que os possuem.

2. Perfilamento

Para realizar o perfilamento da aplicação lu_cb foi utilizado a ferramenta Gprof que faz parte do conjunto de ferramentas binárias GNU Binutils. O comando utilizado para executar o perfilamento foi o seguinte:

$ gprof -b -p lu gmon.out

Com ele, obtemos a tabela de perfilamento *flat profile*, como por exemplo:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 85.76      0.18     0.18  2856832     0.00     0.00  daxpy
  9.53      0.20     0.02    10416     0.00     0.02  bmod
  4.76      0.21     0.01        1    10.01    10.01  InitA
  0.00      0.21     0.00    15008     0.00     0.00  BlockOwner
  0.00      0.21     0.00      496     0.00     0.01  bdiv
  0.00      0.21     0.00      496     0.00     0.02  bmodd
  0.00      0.21     0.00       32     0.00     0.01  lu0
  0.00      0.21     0.00        1     0.00   200.11  OneSolve
  0.00      0.21     0.00        1     0.00   200.11  SlaveStart
  0.00      0.21     0.00        1     0.00     0.00  TouchA
  0.00      0.21     0.00        1     0.00   200.11  lu
```

Foram geradas tabelas para tamanhos de problema diferentes e número de processadores diferentes. Considerando a coluna *self seconds*, que representa o número de segundos decorridos na função, coletamos os dados de todas as tabelas e concluímos que

1. A função *daxpy* é responsável por, em média, 85% do tempo de execução do algoritmo, e ainda, ela é disparadamente a função mais chamada durante a execução do algoritmo.
2. A função *bmod* consome cerca de 9% a 13% do tempo de execução total do algoritmo.
3. As funções *daxpy* e *bmod* juntas, consomem cerca de 95% a 98% do tempo de execução total do algoritmo.

A partir destes resultados, concluímos que as funções *daxpy* e *bmod* serão os alvos principais para aplicar paralelização.

2. Escalabilidade

A análise da escalabilidade das funções *daxpy* e *bmod* foi realizada executando o algoritmo variando o tamanho do problema (dimensão da matriz) e número de processadores. Para cada uma das funções, foi uma tabela de SpeedUp e outra e Eficiência com valores da dimensão da matriz e o número de processadores.

2.1 Resultados para a função *daxpy*

Abaixo, são mostradas as tabelas do *SpeedUp* e Eficiência para diferentes tamanhos do problema e número de processadores para a função *daxpy*.

SpeedUp				
p \ n	512	1024	2048	4096
2	1,125	0,957	1,095	1,078
4	0,857	0,918	1,085	1,040
8	1,125	1,031	1,037	1,018
16	0,857	0,509	0,528	1,018
32	0,450	0,477	0,679	1,018
64	0,750	0,655	1,101	1,018

Eficiência				
p \ n	512	1024	2048	4096
2	0,563	0,479	0,548	0,539
4	0,214	0,230	0,271	0,260
8	0,141	0,129	0,130	0,127
16	0,054	0,032	0,033	0,064
32	0,014	0,015	0,021	0,032
64	0,012	0,010	0,017	0,016

2.2 Resultados para a função *bmod*

De modo semelhante, tabelas foram construídas contendo os resultados para a função *bmod*:

SpeedUp				
p \ n	512	1024	2048	4096
2	0,000	1,000	0,746	0,835
4	0,000	0,833	0,700	0,771
8	0,000	0,952	0,704	0,739
16	0,000	0,909	0,733	0,739
32	0,000	0,714	0,947	0,739
64	0,000	0,952	1,326	0,739

Eficiência				
p \ n	512	1024	2048	4096
2	0,000	0,500	0,373	0,417
4	0,000	0,208	0,175	0,193
8	0,000	0,119	0,088	0,092
16	0,000	0,057	0,046	0,046
32	0,000	0,022	0,030	0,023
64	0,000	0,015	0,021	0,012

2.3 Conclusões sobre escalabilidade

Finalmente, uma vez que analisamos os dados, conclusões claras sobre escalabilidade não puderam ser elaboradas, uma vez que o algoritmo estranhamente executa mais rápido quando executado com um processador. Isto pode ser observado pelos valores de SpeedUp tanto da função *daxpy* quanto da função *bmod*.

De acordo com a referência consultada, o problema é pobremente escalável, e em específico, a aplicação LU tem muitas regiões de barreira quando comparado às demais aplicações da suíte splash2. Talvez isto justifique os resultados obtidos, entretanto, uma investigação mais profunda do problema é necessário para tomar conclusões mais sólidas a respeito.

Referência

WOO, Steven Cameron et al. **The SPLASH-2 programs: Characterization and methodological considerations**. ACM SIGARCH computer architecture news, v. 23, n. 2, p. 24-36, 1995.