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 FINAL

Alexandre Henrique Soares Dias: 20180001100

José Tobias Souza dos Santos: 20180154969

Gabriel Medeiros Coelho: 20180009538

Natal-RN

2019

1. Introdução

Neste relatório final, além de contextualizar a aplicação que fora escolhida para ser trabalhada ao longo do curso, também apresentamos as melhorias e problemas encontrados com relação à análise e otimização do algoritmo LU-CB.

1.1 Decomposição LU e aplicação do splash2

A nossa aplicação trata-se da decomposição LU ou fatorização Lower-Upper (LU) que, por sua vez, é comumente aplicada na computação para resolução de sistemas lineares quadrados, e ainda, é utilizada 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. 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.

2. Análise de perfilamento

A análise de perfilamento foi executada no código original, portado e otimizado. A ferramenta utilizada nesta tarefa foi o Gprof que faz parte do conjunto de ferramentas binárias GNU Binutils. Com o Gprof, obtemos as seguintes tabelas de perfilamento (*flat profile*) para os códigos original e portado para OpenMP, respectivamente:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
82.27	84.03	84.03	178486503	0.00	0.00	daxpy
15.92	100.29	16.26	862179	0.00	0.00	bmod
0.79	101.10	0.81	1	0.81	0.81	InitA
0.52	101.63	0.54	8	0.07	12.66	lu
0.27	101.91	0.28	11774456	0.00	0.00	BlockOwner
0.14	102.05	0.14	14752	0.00	0.00	bmodd
0.08	102.13	0.08	5	0.02	0.02	TouchA
0.05	102.18	0.05	7571	0.00	0.00	bdiv
0.00	102.18	0.00	129	0.00	0.00	lu0
0.00	102.18	0.00	8	0.00	12.67	OneSolve
0.00	102.18	0.00	1	0.00	101.37	SlaveStart

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
82.03	85.74	85.74	220665923	0.00	0.00	daxpy
16.13	102.61	16.86	1248661	0.00	0.00	bmod
0.83	103.48	0.87	1	0.87	0.87	InitA
0.68	104.19	0.71	1	0.71	103.73	lu
0.23	104.43	0.24	13241725	0.00	0.00	BlockOwner
0.12	104.56	0.13	11509	0.00	0.00	bmodd
0.05	104.61	0.05	5011	0.00	0.00	bdiv
0.03	104.64	0.03	1	0.03	0.03	TouchA
0.00	104.64	0.00	99	0.00	0.00	lu0
0.00	104.64	0.00	1	0.00	103.76	OneSolve
0.00	104.64	0.00	1	0.00	103.76	SlaveStart
0.00	104.64	0.00	1	0.00	103.76	printerr

Como pode ser visto, o resultado do perfilamento dos códigos original e portado é basicamente o mesmo. Isto é esperado, já que não houve alteração da estrutura como o algoritmo é executado.

Como foi comentado nas apresentações a partir do resultado do perfilamento, as *funções alvo* para otimização são a **daxpy** e **bmod**, pois estas são as que consomem mais tempo de execução do algoritmo e, portanto, foram os alvos principais no processo de otimização do algoritmo.

3. Otimização

3.1 Autovetorização

Com o foco da otimização nas funções **bmod** e **daxpy**, utilizando o código portado para OpenMP, indicamos por linha de comando para o compilador tentar realizar a otimização do algoritmo com o seguinte comando:

$ gcc -g -fopenmp -O3 -funsafe-math-optimizations -ftree-vectorizer-verbose=8 -lm -o lu.o lu.c

As *flags* mais relevantes no tocante a otimização do algoritmo são:

1. -O3: Habilitar um conjunto amplo de diretivas de otimização
2. -funsafe-math-optimizations: Otimização para operações aritméticas em ponto flutuante
3. -ftree-vectorizer-verbose=8: Mostrar um relatório detalhado do resultado da auto-vetorização

3.2 Autovetorização do loop das função daxpy e bmod

O corpo da função *daxpy* é composto por apenas um loop. O compilador conseguiu vetorizar o loop da função daxpy com sucesso. Por outro lado, para a função bmod os resultados obtidos não foram tão diretos. Abaixo, o corpo da função pode ser visualizado:

```
421  void bmod(a, b, c, dimi, dimj, dimk, stridea, strideb, stridec)
422
423  double *a;
424  double *b;
425  double *c;
426  int dimi;
427  int dimj;
428  int dimk;
429  int stridea;
430  int strideb;
431  int stridec;
432
433  {
434    int j;
435    int k;
436    double alpha;
437
438    for (k = 0; k < dimk; k++)
439    {
440      for (j = 0; j < dimj; j++)
441      {
442        alpha = -b[k + j * strideb];
443        daxpy(&c[j * stridec], &a[k * stridea], dimi, alpha);
444      }
445    }
446  }
```

Desta vez, temos 2 laços aninhados, e o resultado obtido para a tentativa de vetorização desses laços foi:

lu.c:438:3 = not vectorized:

too many BBs

lu.c:440:5 = not vectorized:

too many BBs

Primeiramente tentamos otimizar essa função pesquisando sobre possíveis soluções para esse problema. Basicamente, “too many BBs” ditam o fato de que os loops nessa função possuem vários blocos básicos de códigos (blocos inerentemente seriais que possuem uma entrada e uma saída). Após uma pesquisa sistemática, vimos que para mudar essa parte do código, deveríamos mudar a lógica de várias funções e o jeito que as mesmas realizavam atribuições a variáveis. Porém, observando com um pouco mais detalhe, vimos que a função bmod simplesmente realiza algumas atribuições e depois chama a função daxpy, que é a principal função do código e é responsável pela matemática pesada em si. Logo, vimos que não era necessário nos preocupar muito com a bmod, visto que o importante mesmo era garantir que a daxpy fosse otimizada, o que de fato ocorreu.

3.3 Diretivas de otimização

Uma vez que a função daxpy já estava em uma região paralela do algoritmo, a autovetorização do seu único loop interno foi considerada como o esgotamento das oportunidades de otimização nesta função. Sendo assim, não foi utilizada nenhuma diretiva nesta função para tentar otimizá-la ainda mais.

Por outro lado, como foi explicado na seção anterior, várias tentativas foram realizadas para otimizar a função **bmod**, como adaptações para atingir a autovetorização e o uso de algumas diretivas OpenMP.

3.4 Otimização por refatoração

Conseguimos otimizar a função de inicialização da matriz A (InitA) através de uma refatoração de código. Na função em questão, haviam dois loops aninhados mais um outro loop que executavam n (tamanho do problema) cada um, o que caracteriza a complexidade dessa função como $2n^2+n$. Após refatorar o código desta função, conseguimos reduzir a complexidade para n^2+n.

Como essa função é serial, essa melhoria acarreta em uma diminuição considerável do tempo de execução da função, principalmente para valores cada vez maiores de n.

4. Escalabilidade

Uma vez que o algoritmo foi otimizado, executamos cada um dos três algoritmos 5 vezes no super computador e coletamos a mediana entre os tempos. Dessa maneira, construímos as seguintes tabelas de tempos de execução:

Tempos de execução do programa (pthread)					
p \ n	512	1024	2048	4096	8192
1	0,195	1,548	12,357	99,494	794,053
2	0,105	0,800	6,289	49,985	398,137
4	0,055	0,414	3,191	25,136	200,095
8	0,036	0,223	1,646	12,871	101,169
16	0,024	0,136	0,879	6,543	51,335
32	0,013	0,068	0,460	3,499	27,897

Tempos de execução do programa (OpenMP)					
p \ n	512	1024	2048	4096	8192
1	0,196	1,553	12,335	99,128	793,343
2	0,102	0,787	6,189	49,284	394,732
4	0,053	0,402	3,114	24,989	197,155
8	0,028	0,207	1,583	12,464	99,046
16	0,015	0,108	0,817	6,324	49,961
32	0,012	0,062	0,436	3,246	25,445

Tempos de execução do programa (Otimizado)					
p \ n	512	1024	2048	4096	8192
1	0,037	0,294	2,354	19,214	154,001
2	0,021	0,158	1,241	9,931	79,814
4	0,011	0,080	0,625	4,972	40,026
8	0,006	0,042	0,321	2,545	20,292
16	0,004	0,022	0,165	1,299	10,292
32	0,002	0,012	0,088	0,685	5,608

Os tempos de execução do algoritmo original e portado são aproximadamente os mesmos, sendo que o código portado apresenta um desempenho ligeiramente melhor do que o original. Em contrapartida, com o algoritmo otimizado, obtivemos tempos em média 5 vezes mais rápidos, o que consideramos ser uma melhoria significativa. Por outro lado, analisamos a escalabilidade pelas tabelas de eficiência abaixo:

Eficiência (pthread)				
p \ n	512	1024	2048	4096
2	0,926	0,968	0,982	0,995
4	0,879	0,935	0,968	0,990
8	0,677	0,868	0,939	0,966
16	0,504	0,711	0,879	0,950
32	0,469	0,711	0,839	0,889

Eficiência (OpenMP)				
p \ n	512	1024	2048	4096
2	0,961	0,987	0,997	0,999
4	0,925	0,966	0,990	0,992
8	0,875	0,938	0,974	0,994
16	0,817	0,899	0,944	0,980
32	0,628	0,783	0,884	0,954

Eficiência (Otimizado)				
p \ n	512	1024	2048	4096
2	0,883	0,929	0,949	0,967
4	0,808	0,921	0,941	0,966
8	0,798	0,886	0,917	0,944
16	0,653	0,837	0,894	0,924
32	0,544	0,764	0,837	0,877

Analisando as diagonais das tabelas, percebe-se que a eficiência se mantem aproximadamente constante com um desvio de +- 5%, isso implica que o algoritmo, incluindo a versão otimizada, é **fracamente escalável**.

Um fato interessante que notamos é que as eficiências do algoritmo otimizado foram ligeiramene menor do que as eficiências dos outros algoritmos. Contudo, não conseguimos chegar a uma conclusão clara sobre esse fato.

5. Conclusão

Após tudo o que foi desenvolvido ao longo do semestre, concluímos que os resultados finais foram bastante satisfatórios pois, com o código portado de pthreads para OpenMP, foi possível capturar a conveniência da biblioteca OpenMP através do que foi aprendido em sala e nos exercícios propostos.

E ainda, no que tange a performance do algoritmo, obtivemos tempos de execução 5 vezes menores do que os tempos do código em pthreads, sendo este um ponto extremamente positivo do nosso trabalho. Um ponto que não consideramos tão positivo foi o fato de não conseguirmos otimizar mais ainda o algoritmo através do uso de direticas do OpenMP e tasks.

Finalemente, também foi interessante analisar a relação de escalabilidade entre os algoritmos, e ver ainda que todos são fracamente escaláveis.