Universidade de Brasília

Programação Paralela

Relatório do Exercício 3

Autor:
Alexandre Lucchesi Alencar

Professor:
George Luiz Medeiros Teodoro

19 de outubro de 2014

1 Introdução

Este relatório tem como objetivo apresentar os resultados obtidos a partir da execução do terceiro exercício de programação paralela [1], que consiste na implementação de uma árvore de redução de soma (*sum tree*) utilizando Message Passing Interface (MPI) [1]. Primeiramente, os aspectos principais do algoritmo desenvolvidos são apresentados. Em seguida, é realizada uma análise de desempenho comparando os tempos de execução do algoritmo em diversas configurações, isto é, variando-se o número de processos e a quantidade de números de ponto-flutuante a serem somados. O código-fonte completo deste trabalho, incluindo os arquivos LaTeX que compõem este relatório, estão publicamente disponíveis no GitHub [2].

1.1 *Hardware* Utilizado

- Processador: AMD FX(tm)-8350 Eight-Core Processor
- Velocidade por *core*: 1406.2 MHz
- Número de processadores: 1
- Número de *cores*: 8 (máx. 8)
- Número de *threads*: 8 (máx. 8)
- L1 Data cache: 8 x 16 KBytes, 4-way set associative, 64-byte line size
- L1 Instruction cache: 4 x 64 KBytes, 2-way set associative, 64-byte line size
- L2 cache: 4 x 2048 KBytes, 16-way set associative, 64-byte line size
- L3 cache: 8 MBytes, 64-way set associative, 64-byte line size

2 O Algoritmo

2.1 Artefatos Desenvolvidos

O programa desenvolvido possui duas funcionalidades principais: (i) geração de um arquivo de dados contendo um número arbitrário de valores de ponto-flutuante; (ii) e processamento de um arquivo de dados retornando a soma

[1] Neste trabalho, utilizou-se a implementação OpenMPI para Mac OS X, instalada a partir do utilitário Homebrew.

[2] `https://github.com/alexandrelucchesi/parallel-programming-ex03`

dos elementos e o tempo de execução do algoritmo. Além do programa principal, foram desenvolvidos dois *scripts bash*: um para facilitar a execução do programa principal, encapsulando a chamada ao `mpiexec` (ou `mpirun`), e outro para automatizar os testes da aplicação. Esses artefatos são descritos a seguir.

- `main.c`: programa em C contendo o código-fonte da aplicação. Após compilado com o `mpicc` (vide Makefile), pode ser executado chamando-se o *script* `run.sh` passando-se o número de processos e o arquivo de dados. O `run.sh` executará o programa usando o `mpiexec` e passando esses dois argumentos, que são recebidos via `scanf()`. A saída do programa é uma linha contendo dois números: o primeiro representa o resultado da soma dos números de ponto-flutuante e o segundo, o tempo de execução do algoritmo de redução em milisegundos (desconsiderando o tempo de entrada de dados).

- `test.sh`: *script* desenvolvido para automatizar os testes da aplicação. Recebe como entrada 2 argumentos, em ordem:

  - `max_numbers`: número máximo de processos. O *script* varia o número de processos de 2^{20} até $2^{max\_numbers}$.
  - `max_runs`: número máximo de vezes em que o programa deve ser executado em uma mesma configuração.

2.2 Geração do Arquivo de Dados

Para a geração de quantidades configuráveis de números de ponto-flutante em um formato apropriado para servir de entrada para o programa, pode-se executar o binário proveniente do processo de compilação diretamente. Por exemplo, para gerar um arquivo de dados, `numbers.dat`, contendo, por exemplo, 64 elementos, basta executar o binário passando-se a *flag* `-gen`, conforme descrito a seguir:

```
$ make      # Gera o binário com nome: 'sumtree'.
$ ./sumtree -gen numbers.dat 64
```

Uma outra opção disponível é a `--help`, que exibe informações de uso da aplicação.

2.3 Função de "Espalhamento"

Para distribuir os dados entre os diferentes processos, foram criadas duas funções:

```
void scather(int my_rank, int comm_sz,
    unsigned int *my_count, float **my_nums);
void scather_intercalate(int my_rank, int comm_sz,
    unsigned int *my_count, float **my_nums);
```

Uma sempre pode ser utilizada no lugar da outra sem alterar o resultado final (note que a assinatura é a mesma). A única diferença está na política de atribuição dos números aos processos. Internamente, o processo com `my_rank` igual a zero é sempre o responsável por ler o arquivo de dados e dividir os números entre os `comm_sz` processos, retornando em `my_count` e `my_nums` a quantidade de elementos e os números, respectivamente.

No caso da `scather`, a quantidade total de números (lida do arquivo de entrada) é dividida pelo número de processos (`comm_sz`) e o resultado (`res`) obtido é utilizado para atribuir sequencialmente os números aos processos, ou seja, o processo 0 recebe os números indexados pelo intervalo $[0, res - 1]$, o processo 1 recebe $[res, 2 \times res - 1]$, e assim por diante.

Por outro lado, a `scather_intercalate` intercala os números entre os processos, varrendo o vetor e atribuindo o elemento no índice i ao processo $i \bmod comm\_sz$.

2.4 Evitando *Deadlocks*

Na implementação do MPI utilizada, ambas as primitivas `Send()` e `Recv()` são "blocantes". Isso significa que cuidado adicional deve ser tomado para que não ocorram *deadlocks*. O trecho de código 1 apresenta como a função `reduce_sumtree()` foi projetada para evitar a ocorrência de *deadlocks*.

Código 1: Ordem das primitivas MPI_Send() e MPI_Recv() na função `reduce_sumtree()`.

```
1  ...
2  if (my_rank % 2 == 0) {
3      int dst = my_rank + 1;
4
5      // Copy second half of ‘nums[]‘ to ‘my_nums‘.
6      memcpy(my_nums, nums + qty, qty * sizeof(float));
7
8      // Send first half of ‘nums[]‘ to ‘dst‘.
9      MPI_Send(nums, qty, MPI_FLOAT, dst, 2, MPI_COMM_WORLD);
10
11     // Receive second half of his ‘nums[]‘ into ‘his_nums‘.
12     MPI_Recv(his_nums, qty, MPI_FLOAT, dst, 2, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
13 } else {
14     int dst = my_rank - 1;
15
16     // Copy first half of ‘nums[]‘ to ‘my_nums‘.
17     memcpy(my_nums, nums, qty * sizeof(float));
18
19     // Receive first half of his ‘nums[]‘ into ‘his_nums‘.
20     MPI_Recv(his_nums, qty, MPI_FLOAT, dst, 2, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
21
22     // Send second half of ‘nums[]‘ to ‘dst‘.
23     MPI_Send(nums + qty, qty, MPI_FLOAT, dst, 2, MPI_COMM_WORLD);
24 }
25 ...
```

Se as primitivas `MPI_Recv()` e `MPI_Send()` aparecerem na mesma ordem no `if` e no `else`, os processos entrarão em *deadlock*. Ao colocá-los de forma alternada, garante-se que para cada `MPI_Send()` existirá um `MPI_Recv()` e vice-versa.

3 Medida de Tempo de Execução

Para medir o tempo máximo de execução da aplicação de forma precisa, utilizou-se *barreiras*. Com uma chamada à função `MPI_Barrier()` antes de `reduce_sumtree()`, garante-se que todos os processos começam a executar o algoritmo de redução "ao mesmo tempo". Com outra chamada à função `MPI_Barrier()` após a chamada à `reduce_sumtree()` é possível sincronizar todos os processos no ponto de término da execução do algoritmo. Dessa forma, coletando os tempos ("relógio de parede") no processo 0 imediatamente após as barreiras, é possível calcular o tempo total de execução da aplicação.

Utilizou-se a função `gettimeofday()` (disponível em "`sys/time.h`") para se obter os tempos de início e término e aritmética simples para se obter o intervalo de execução em milisegundos.

4 Resultados

Executou-se o *script* de testes (`test.sh`) passando-se como argumentos: 25 `max_numbers`, para executar testes variando-se a quantidade de elementos de 2^{20} à 2^{25}; e 5 `max_runs`, para se realizar 5 execuções em cada configuração.

O *script* `test.sh` gerou como saída arquivos de dados contendo os números de ponto-flutuante para serem usados nos testes (extensão `.dat`) e arquivos no formato CSV contendo as tabelas que compõem este relatório. Cada tabela é indexada pela quantidade de elementos (no exemplo acima, de 10^{20} à 10^{25}) e o número da execução (no exemplo acima, de 1 a 5). Metade possui o prefixo `sum_n`, onde `n` é o número de processos utilizado, representando os valores aproximados da soma calculados. A outra metade possui prefixo `time_n`, e contém os valores dos tempos de execução. Em poucas palavras, tem-se 2 arquivos para cada número de processos contendo as duas saídas do programa: o valor estimado da soma e o tempo de execução do algoritmo.

As seções a seguir apresentam os resultados dos testes, apresentando métricas de *speedup*, eficiência e escalabilidade. Os resultados são apresentados sob a forma de gráficos. As tabelas com os dados exatos que deram origem a esses gráficos estão anexadas ao final do documento.

4.1 *Speedup*

A Figura 1 apresenta um gráfico em que cada curva relaciona o *speedup* obtido, o número de processos criados e a quantidade de elementos somados na redução. De uma forma geral, observa-se que o *speedup aumenta* conforme

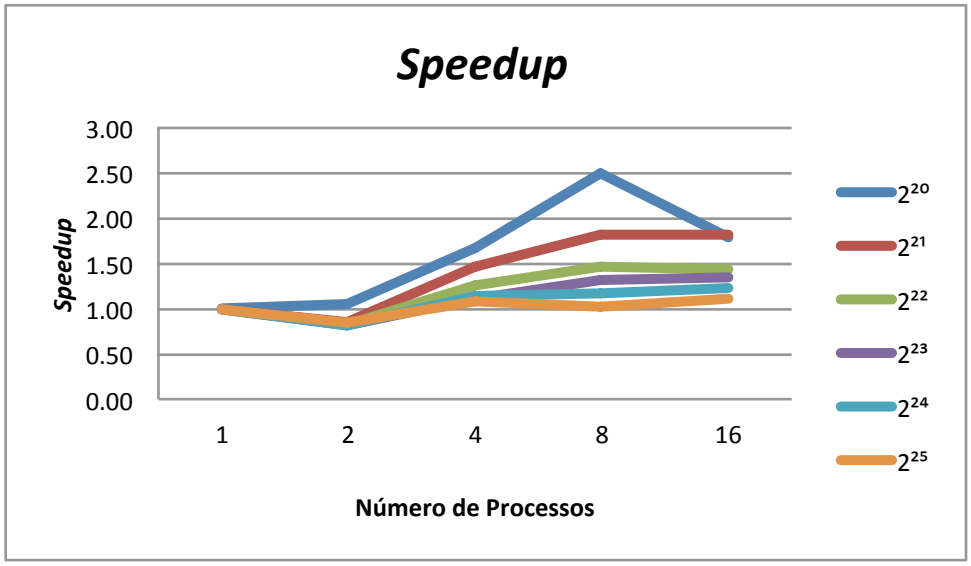

Figura 1: Gráfico apresentando a relação entre o *speedup*, o número de processos e a quantidade de elementos somados na redução.

a quantidade de processos cresce. A excessão ocorre apenas quando se utiliza dois processos, onde perde-se desempenho para quase todos os tamanhos de entrada [3].

Isso ocorre porque o *overhead* envolvido na criação de dois processos e, principalmente, na comunicação entre os processos é maior do que os possíveis ganhos advindos da concorrência entre os mesmos.

4.2 Eficiência

A Figura 2 apresenta um gráfico em que cada curva relaciona a *eficiência* obtida, o número de processos criados e a quantidade de elementos somados na redução. De uma forma geral, observa-se que a *eficiência diminui* conforme a quantidade de processos cresce.

4.3 Escalabilidade

Conforme ilustrado na Figura 2 e observando as tabelas de tempo de execução ao final deste documento, nota-se que a eficiência cai em uma taxa muito maior que os tempos de execução. Dessa forma, conclui-se que a aplicação não possui uma boa *escalabilidade forte*.

[3] Com 2^{20} elementos houve um ganho de desempenho mínimo, com o *speedup* no valor de 1.04.

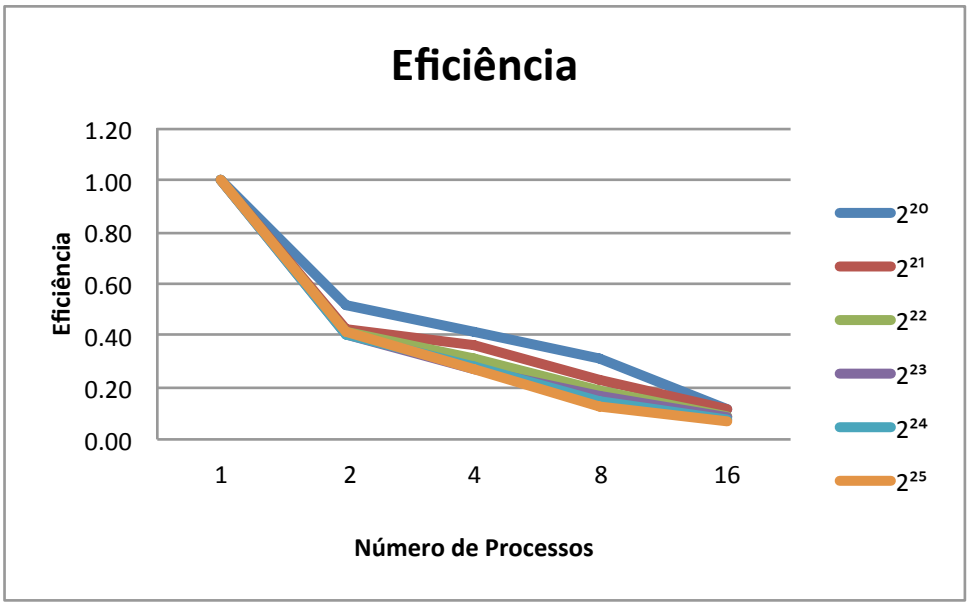

Figura 2: Gráfico apresentando a relação entre o *speedup*, o número de processos e a quantidade de elementos somados na redução.

De forma similar, não se tem também uma boa *escalabilidade fraca*, conforme apresentado na Figura 3, que relaciona a eficiência, o número de processos, e a quantidade de elementos somados na redução. Não está explícito no gráfico, mas para cada número de processos obteve-se o valor da eficiência de acordo incrementando-se proporcionalmente o tamanho do problema (ou número de elementos a serem somados). Sendo assim, para 1 processo, utilizou-se 2^20 elementos; para 2 processos, 2^21 elementos; para 4 processos, 2^22 elementos; e assim por diante.

5 Conclusão

Este trabalho possibilitou uma maior compreensão acerca de modelos de programação baseados em trocas de mensagem, em particular, que utilizam primitivas `Send()` e `Recv()`, através da implementação de uma árvore de redução de soma usando MPI.

O algoritmo desenvolvido possibilita a simulação de reduções usando um número grande de processos, utilizando as primitivas citadas anteriormente para realizar somas intermediárias até chegar na redução a partir de uma *Sum Tree*. Uma análise dos tempos de execução do algoritmo evidenciou ganhos de desempenho (*speedup*) de até 252% em relação à versão sequencial.

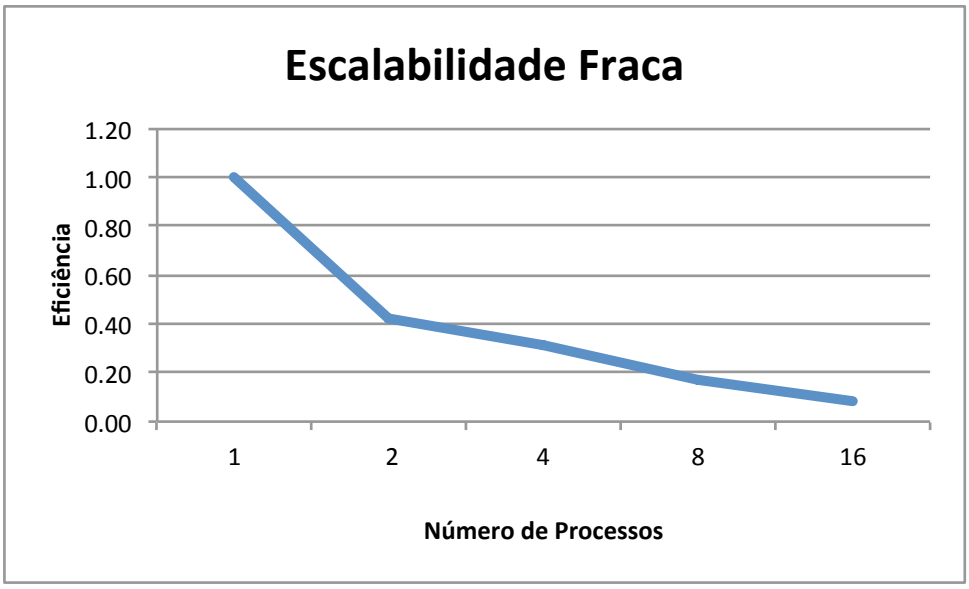

Figura 3: Gráfico apresentando a relação entre a eficiência e o número de processos, aumentando-se proporcionalmente o tamanho do problema.

Por fim, é válido ressaltar que o o *design* da aplicação (incluindo o *script* `test.sh`) permite variar de forma fácil as condições de teste, permitindo a reprodução dos experimentos apresentados neste trabalho e facilitando a experimentação com novas configurações.

Referências

[1] G. L. M. Teodoro. Programação paralela, exercício de programação 03, sum tree, September 2014.

Anexo I

Sequencial (Nro. de Execuções x Nro. de Elementos)

	Tempos de Execução							
	1	2	3	4	5	Média	D. Padrão	Coef. de Variação
2^{20}	5	5	5	5	5	5	0	0.00%
2^{21}	9	9	9	9	9	9	0	0.00%
2^{22}	18	18	18	17	17	17.6	0.4898979	2.78%
2^{23}	35	35	35	35	35	35	0	0.00%
2^{24}	69	69	70	69	70	69.4	0.4898979	0.71%
2^{25}	138	138	139	138	138	138.2	0.4	0.29%

2 Processos (Nro. de Execuções x Nro. de Elementos)

	Tempos de Execução							
	1	2	3	4	5	Média	D. Padrão	Coef. de Variação
2^{20}	5	5	4	5	5	4.8	0.4	8.33%
2^{21}	10	12	10	11	10	10.6	0.8	7.55%
2^{22}	21	21	21	22	22	21.4	0.4898979	2.29%
2^{23}	46	44	43	43	43	43.8	1.1661904	2.66%
2^{24}	86	86	86	85	85	85.6	0.4898979	0.57%
2^{25}	166	168	168	166	165	166.6	1.2	0.72%

4 Processos (Nro. de Execuções x Nro. de Elementos)

	Tempos de Execução							
	1	2	3	4	5	Média	D. Padrão	Coef. de Variação
2^{20}	3	3	3	3	3	3	0	0.00%
2^{21}	6	7	6	6	6	6.2	0.4	6.45%
2^{22}	13	15	15	15	13	14.2	0.9797959	6.90%
2^{23}	32	30	30	33	34	31.8	1.6	5.03%
2^{24}	61	63	60	61	61	61.2	0.9797959	1.60%
2^{25}	129	128	137	126	128	129.6	3.8262253	2.95%

8 Processos (Nro. de Execuções x Nro. de Elementos)

	Tempos de Execução							
	1	2	3	4	5	Média	D. Padrão	Coef. de Variação
2^{20}	2	2	2	2	2	2	0	0.00%
2^{21}	5	5	5	5	5	5	0	0.00%
2^{22}	12	12	12	12	12	12	0	0.00%
2^{23}	26	26	27	27	26	26.4	0.4898979	1.86%
2^{24}	59	59	57	58	62	59	1.6733201	2.84%
2^{25}	137	142	135	136	132	136.4	3.2619013	2.39%

16 Processos (Nro. de Execuções x Nro. de Elementos)

Tempos de Execução								
	1	2	3	4	5	Média	D. Padrão	Coef. de Variação
2^{20}	3	3	3	2	3	2.8	0.4	14.29%
2^{21}	5	5	5	5	5	5	0	0.00%
2^{22}	13	12	12	13	12	12.4	0.4898979	3.95%
2^{23}	26	26	26	26	27	26.2	0.4	1.53%
2^{24}	56	60	62	54	52	56.8	3.7094474	6.53%
2^{25}	124	127	123	124	123	124.2	1.4696938	1.18%

Speedup (Nro. de Processos x Nro. de Elementos)

	1	2	4	8	16
2^{20}	1.00	1.04	1.67	2.50	1.79
2^{21}	1.00	0.85	1.45	1.80	1.80
2^{22}	1.00	0.82	1.24	1.47	1.42
2^{23}	1.00	0.80	1.10	1.33	1.34
2^{24}	1.00	0.81	1.13	1.18	1.22
2^{25}	1.00	0.83	1.07	1.01	1.11

Eficiência (Nro. de Processos x Nro. de Elementos)

	1	2	4	8	16
2^{20}	1.00	0.52	0.42	0.31	0.11
2^{21}	1.00	0.42	0.36	0.23	0.11
2^{22}	1.00	0.41	0.31	0.18	0.09
2^{23}	1.00	0.40	0.28	0.17	0.08
2^{24}	1.00	0.41	0.28	0.15	0.08
2^{25}	1.00	0.41	0.27	0.13	0.07

Escalabilidade Fraca

1	2	4	8	16
1.00	0.42	0.31	0.17	0.08