Universidade de Brasília

Programação Paralela

Relatório do Exercício 3

Autor:
Alexandre Lucchesi Alencar

Professor:
George Luiz Medeiros Teodoro

14 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: Intel Core i7
- Velocidade: 2 GHz
- Número de processadores: 1
- Número de *cores* reais: 2
- Número de *cores* virtuais: 4 (HyperThreading)
- L1 cache: 32KB
- L2 cache (per core): 256KB
- L3 cache: 4MB

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 dos elementos e o tempo de execução do algoritmo. Além do programa principal, foram desenvolvidos dois *scripts bash*: um para facilitar a execução

[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`

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 4 argumentos, em ordem:

  - `max_threads`: número máximo de *threads*. O *script* varia o número de *threads* de 1 até `max_threads`.
  - `max_tosses`: ordem máxima do número de lançamentos, isto é, o *script* executa a aplicação variando o número de lançamentos, começando em $10^2, 10^3$ até 10^{max_tosses}.
  - `max_runs`: número máximo de vezes em que o programa deve ser executado em uma mesma configuração.
  - `max_time`: *timeout* de execução, ou seja, se o programa não encerrar a execução nesse tempo, ele é finalizado.

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,
13         MPI_STATUS_IGNORE);
14 } else {
15     int dst = my_rank - 1;
16
17     // Copy first half of ‘nums[]‘ to ‘my_nums‘.
18     memcpy(my_nums, nums, qty * sizeof(float));
19
20     // Receive first half of his ‘nums[]‘ into ‘his_nums‘.
21     MPI_Recv(his_nums, qty, MPI_FLOAT, dst, 2, MPI_COMM_WORLD,
22         MPI_STATUS_IGNORE);
23
24     // Send second half of ‘nums[]‘ to ‘dst‘.
25     MPI_Send(nums + qty, qty, MPI_FLOAT, dst, 2, MPI_COMM_WORLD);
26 }
27 ...
```

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 Resultados

Executou-se o *script* de testes (`test.sh`) passando-se como argumentos: 6 *threads*, 10^{12} lançamentos, 5 execuções por configuração e *timeout* de 24 minutos. Além disso, utilizou-se o utilitário de linha de comando `time` para calcular o tempo total de execução dos testes. O resultado é apresentado a seguir.

```
$ time sh test.sh 6 12 5 1440

real    633m8.800s
user    1503m45.812s
sys     1m9.481s
```

O valor `real` é o tempo real ("relógio de parede") transcorrido desde o início da execução do programa até seu término (incluindo períodos de bloqueio, etc). Por outro lado, os valores de `user` e `sys` se referem ao tempo de CPU usado pelo processo em modo usuário e modo *kernel*, respectivamente. Dessa forma, a execução dos testes responsáveis por gerar as tabelas de *benchmark* levou quase 11 horas para ser concluída.

O *script* `test.sh` gera como saída $2 \times max\_threads$ arquivos no formato CSV (no exemplo acima, 12 arquivos). Cada tabela é indexada pelo número de lançamentos (no exemplo acima, de 10^2 à 10^{12}) e o número da execução (no exemplo acima, de 1 a 5). Metade possui o prefixo `pi_n`, onde `n` é o número de *threads* utilizado, representando os valores aproximados de pi calculados. A outra metade possui prefixo `time_n`, e contém os valores dos tempos de execução. O valor $*$ representa um *timeout*. Em poucas palavras, tem-se 2 arquivos por quantidade de *threads* contendo as duas saídas do programa: o valor estimado de pi e o tempo de execução do algoritmo.

As tabelas a seguir apresentam os resultados dos testes. Observa-se que para uma quantidade de lançamentos pequena (10^2 e 10^3), o tempo de execução do algoritmo sequencial é melhor do que nas demais configurações. Isto ocorre porque os ganhos obtidos da paralelização do algoritmo não são suficientes para cobrir o custo de se criar as *threads*. Ao aumentar o número de lançamentos para 10^4, observa-se que o desempenho do algoritmo sequencial já perde para algumas configurações, sendo inferior ao obtido com 3 e 4 *threads*. A partir de 10^5 lançamentos, o desempenho do algoritmo sequencial se torna inferior a todas as demais configurações.

É importante notar que para o algoritmo sequencial e para as configurações 2 e 3 *threads*, o programa excedeu o tempo máximo de execução pré-estabelecido (24 minutos). No entanto, para as configurações 4, 5 e 6 *threads*,

esse tempo não foi atingido, evidenciando que o algoritmo paralelizado é escalável. Além disso, observando-se os tempos de execução quando o número de lançamentos é alto (10^{10} e 10^{11}), conclui-se que o maior desempenho é alcançado utilizando-se 4 *threads*. Isso está relacionado com o fato de que o processador utilizado possui 2 *cores* em HyperThreading 1.1.

Em linhas gerais, para 10^6 lançamentos e 4 *threads*, foi possível obter o *speedup* máximo: $\approx 252\%$. Em contrapartida, para 10^2 lançamentos e 6 *threads*, ocorreu o pior *speedup*: $\approx 4.5\%$. Esse resultado era esperado, uma vez que o custo de se gerenciar 6 *threads* para calcular apenas 100 números é muito alto.

Benchmark

Sequencial

Valor de Pi

	1	2	3	4	5
1.00E+02	2.8	2.8	3.28	3.2	3.32
1.00E+03	3.212	3.168	3.092	3.172	3.104
1.00E+04	3.1512	3.1272	3.1172	3.1932	3.15
1.00E+05	3.13296	3.13948	3.14028	3.14504	3.13304
1.00E+06	3.141212	3.141324	3.140128	3.143152	3.140292
1.00E+07	3.14176	3.14145	3.141892	3.142508	3.141282
1.00E+08	3.141704	3.141639	3.141605	3.141381	3.141768
1.00E+09	3.141599	3.141566	3.141591	3.141578	3.141573
1.00E+10	3.141594	3.141592	3.141595	3.141592	3.141594
1.00E+11	*	*	*	*	*
1.00E+12	*	*	*	*	*

Tempo de Execução

	1	2	3	4	5
1.00E+02	10	9	10	9	10
1.00E+03	35	35	37	36	35
1.00E+04	287	288	288	285	287
1.00E+05	3063	2791	2792	2818	2857
1.00E+06	30795	28307	28184	28053	30821
1.00E+07	282886	282494	280869	290568	282843
1.00E+08	2830318	2914815	2834505	2843038	2837852
1.00E+09	28117168	28033622	28034648	28048268	28025265
1.00E+10	280500346	280228907	280138891	279964573	280303605
1.00E+11	*	*	*	*	*
1.00E+12	*	*	*	*	*

2 threads

Valor de Pi

	1	2	3	4	5
1.00E+02	3.28	2.96	3.04	3.12	3.28
1.00E+03	3.144	3.144	3.216	3.24	3.048
1.00E+04	3.136	3.176	3.1144	3.1608	3.1
1.00E+05	3.14896	3.12816	3.12904	3.13792	3.14424
1.00E+06	3.144544	3.140584	3.144288	3.143832	3.146248
1.00E+07	3.142114	3.140789	3.140964	3.141358	3.141208
1.00E+08	3.141468	3.141638	3.141131	3.141347	3.14154
1.00E+09	3.14161	3.141593	3.141596	3.141576	3.141581
1.00E+10	3.141592	3.141596	3.141589	3.141591	3.141595
1.00E+11	*	*	*	*	*
1.00E+12	*	*	*	*	*

Tempo de Execução

	1	2	3	4	5
1.00E+02	89	92	89	105	103
1.00E+03	117	109	109	119	127
1.00E+04	385	325	330	324	328
1.00E+05	1835	2343	2369	2339	2334
1.00E+06	15078	17014	16996	17034	17586
1.00E+07	200142	179457	151244	225138	168587
1.00E+08	1703396	1947705	2097090	1858184	1592866
1.00E+09	19731505	19570984	19239716	19756756	19084827
1.00E+10	192179479	193882544	190497287	192874675	195533761
1.00E+11	*	*	*	*	*
1.00E+12	*	*	*	*	*

Speedup

	1	2	3	4	5
1.00E+02	11.24%	9.78%	11.24%	8.57%	9.71%
1.00E+03	29.91%	32.11%	33.94%	30.25%	27.56%
1.00E+04	74.55%	88.62%	87.27%	87.96%	87.50%
1.00E+05	166.92%	119.12%	117.86%	120.48%	122.41%
1.00E+06	204.24%	166.37%	165.83%	164.69%	175.26%
1.00E+07	141.34%	157.42%	185.71%	129.06%	167.77%
1.00E+08	166.16%	149.65%	135.16%	153.00%	178.16%
1.00E+09	142.50%	143.24%	145.71%	141.97%	146.85%
1.00E+10	145.96%	144.54%	147.06%	145.15%	143.35%
1.00E+11	*	*	*	*	*
1.00E+12	*	*	*	*	*

3 threads

Valor de Pi

	1	2	3	4	5
1.00E+02	2.8	3.4	3.16	3.28	3.04
1.00E+03	3.228	3.208	3.208	3.204	3.16
1.00E+04	3.1528	3.1588	3.1456	3.1288	3.1068
1.00E+05	3.14632	3.14116	3.136	3.13324	3.14244
1.00E+06	3.143404	3.138688	3.140476	3.14208	3.140092
1.00E+07	3.141964	3.141119	3.140475	3.141258	3.141078
1.00E+08	3.141778	3.141545	3.141671	3.141179	3.141445
1.00E+09	3.141507	3.141701	3.14153	3.141589	3.141617
1.00E+10	3.141596	3.14159	3.1416	3.141593	3.141588
1.00E+11	*	*	*	*	*
1.00E+12	*	*	*	*	*

Tempo de Execução

	1	2	3	4	5
1.00E+02	136	127	141	131	126
1.00E+03	167	143	147	140	146
1.00E+04	254	252	253	253	255
1.00E+05	1656	1669	1673	1702	1671
1.00E+06	14378	15007	15017	14977	15738
1.00E+07	135963	150853	135801	143265	153833
1.00E+08	1515653	1363380	1409022	1397496	1430213
1.00E+09	13568218	13361771	13530002	13495157	13688096
1.00E+10	133652304	134022003	133904506	133340383	133718315
1.00E+11	*	*	*	*	*
1.00E+12	*	*	*	*	*

Speedup

	1	2	3	4	5
1.00E+02	7.35%	7.09%	7.09%	6.87%	7.94%
1.00E+03	20.96%	24.48%	25.17%	25.71%	23.97%
1.00E+04	112.99%	114.29%	113.83%	112.65%	112.55%
1.00E+05	184.96%	167.23%	166.89%	165.57%	170.98%
1.00E+06	214.18%	188.63%	187.68%	187.31%	195.84%
1.00E+07	208.06%	187.26%	206.82%	202.82%	183.86%
1.00E+08	186.74%	213.79%	201.17%	203.44%	198.42%
1.00E+09	207.23%	209.80%	207.20%	207.84%	204.74%
1.00E+10	209.87%	209.09%	209.21%	209.96%	209.62%
1.00E+11	*	*	*	*	*
1.00E+12	*	*	*	*	*

Benchmark

4 threads

Valor de Pi

	1	2	3	4	5
1.00E+02	3.52	2.72	3.68	3.04	3.52
1.00E+03	3.296	3.264	3.072	3.04	3.2
1.00E+04	3.1648	3.12	3.1424	3.152	3.1408
1.00E+05	3.13536	3.14768	3.1472	3.1456	3.13792
1.00E+06	3.1464	3.141104	3.137808	3.14456	3.136144
1.00E+07	3.14375	3.141883	3.142301	3.142443	3.14173
1.00E+08	3.141597	3.141739	3.142046	3.14216	3.142156
1.00E+09	3.141526	3.141511	3.141622	3.14161	3.141606
1.00E+10	3.14159	3.141602	3.141587	3.141599	3.141597
1.00E+11	3.141593	3.141592	3.141593	3.141592	3.141593
1.00E+12	*	*	*	*	*

Tempo de Execução

	1	2	3	4	5
1.00E+02	162	150	151	151	153
1.00E+03	203	144	161	151	151
1.00E+04	361	275	247	259	256
1.00E+05	1480	1338	1339	1334	1336
1.00E+06	12216	12376	12368	12713	12341
1.00E+07	120271	120090	120436	120003	120377
1.00E+08	1195526	1211219	1224043	1198886	1196745
1.00E+09	11995083	11952330	11950297	11965284	11952279
1.00E+10	119524543	119656347	119993341	119593283	119774364
1.00E+11	1201008146	1202551849	1203193342	1200559938	1203574797
1.00E+12	*	*	*	*	*

Speedup

	1	2	3	4	5
1.00E+02	6.17%	6.00%	6.62%	5.96%	6.54%
1.00E+03	17.24%	24.31%	22.98%	23.84%	23.18%
1.00E+04	79.50%	104.73%	116.60%	110.04%	112.11%
1.00E+05	206.96%	208.59%	208.51%	211.24%	213.85%
1.00E+06	252.09%	228.72%	227.88%	220.66%	249.74%
1.00E+07	235.21%	235.24%	233.21%	242.13%	234.96%
1.00E+08	236.74%	240.65%	231.57%	237.14%	237.13%
1.00E+09	234.41%	234.55%	234.59%	234.41%	234.48%
1.00E+10	234.68%	234.19%	233.46%	234.10%	234.03%
1.00E+11	*	*	*	*	*
1.00E+12	*	*	*	*	*

5 threads

Valor de Pi

	1	2	3	4	5
1.00E+02	3.4	3.2	3.2	2.8	3.2
1.00E+03	3.24	3.24	3.04	3.16	3.2
1.00E+04	3.058	3.148	3.152	3.132	3.148
1.00E+05	3.1394	3.1404	3.1452	3.1426	3.1122
1.00E+06	3.14678	3.14566	3.1371	3.13962	3.1383
1.00E+07	3.1404	3.141944	3.14135	3.142294	3.14128
1.00E+08	3.141935	3.141615	3.141298	3.141935	3.141739
1.00E+09	3.141581	3.141612	3.141557	3.141499	3.141579
1.00E+10	3.141582	3.141599	3.141592	3.141597	3.141594
1.00E+11	3.141593	3.141593	3.141593	3.141593	3.141592
1.00E+12	*	*	*	*	*

Tempo de Execução

	1	2	3	4	5
1.00E+02	153	161	167	149	171
1.00E+03	233	200	173	168	166
1.00E+04	322	293	294	293	295
1.00E+05	1774	1640	1667	1644	1658
1.00E+06	15318	15302	15402	15533	15347
1.00E+07	130398	136801	143131	130657	137871
1.00E+08	1468643	1319933	1356675	1346917	1331484
1.00E+09	13623952	13597283	13567071	13568261	13604632
1.00E+10	135174170	135995421	135535350	135857350	135590523
1.00E+11	1357282993	1359368352	1357791326	1354537295	1357221225
1.00E+12	*	*	*	*	*

Speedup

	1	2	3	4	5
1.00E+02	6.54%	5.59%	5.99%	6.04%	5.85%
1.00E+03	15.02%	17.50%	21.39%	21.43%	21.08%
1.00E+04	89.13%	98.29%	97.96%	97.27%	97.29%
1.00E+05	172.66%	170.18%	167.49%	171.41%	172.32%
1.00E+06	201.04%	184.99%	182.99%	180.60%	200.83%
1.00E+07	216.94%	206.50%	196.23%	222.39%	205.15%
1.00E+08	192.72%	220.83%	208.93%	211.08%	213.13%
1.00E+09	206.38%	206.17%	206.64%	206.72%	206.00%
1.00E+10	207.51%	206.06%	206.69%	206.07%	206.73%
1.00E+11	*	*	*	*	*
1.00E+12	*	*	*	*	*

6 threads

Valor de Pi

	1	2	3	4	5
1.00E+02	3.24	3.2	3.24	2.92	2.76
1.00E+03	3.136	3.224	3.228	2.944	2.968
1.00E+04	3.1168	3.0612	3.174	3.1432	3.0944
1.00E+05	3.15088	3.16216	3.1554	3.11896	3.14768
1.00E+06	3.136232	3.14406	3.13914	3.141488	3.13926
1.00E+07	3.142504	3.142235	3.141805	3.142652	3.140067
1.00E+08	3.141644	3.14115	3.141227	3.141702	3.141513
1.00E+09	3.141678	3.141699	3.141484	3.141662	3.141555
1.00E+10	3.141597	3.141583	3.141591	3.141603	3.141603
1.00E+11	3.141593	3.141593	3.141593	3.141594	3.141591
1.00E+12	*	*	*	*	*

Tempo de Execução

	1	2	3	4	5
1.00E+02	170	172	184	200	182
1.00E+03	224	193	195	237	185
1.00E+04	349	293	306	291	310
1.00E+05	1770	1700	1700	1704	1480
1.00E+06	13495	15599	13248	13735	13220
1.00E+07	129743	136448	130435	134385	136975
1.00E+08	1331668	1278803	1325363	1352629	1328453
1.00E+09	13509233	13417546	13322456	13235895	13363284
1.00E+10	133742811	134047003	134292618	134440907	133438725
1.00E+11	1337651947	1336530016	1341097789	1335527054	1340819032
1.00E+12	*	*	*	*	*

Speedup

	1	2	3	4	5
1.00E+02	5.88%	5.23%	5.43%	4.50%	5.49%
1.00E+03	15.63%	18.13%	18.97%	15.19%	18.92%
1.00E+04	82.23%	98.29%	94.12%	97.94%	92.58%
1.00E+05	173.05%	164.18%	164.24%	165.38%	193.04%
1.00E+06	228.20%	181.47%	212.74%	204.24%	233.14%
1.00E+07	218.04%	207.03%	215.33%	216.22%	206.49%
1.00E+08	212.54%	227.93%	213.87%	210.19%	213.62%
1.00E+09	208.13%	208.93%	210.43%	211.91%	209.72%
1.00E+10	209.73%	209.05%	208.60%	208.24%	210.06%
1.00E+11	*	*	*	*	*
1.00E+12	*	*	*	*	*

4 Conclusão

Este trabalho possibilitou uma maior compreensão sobre modelos de programação baseados em troca de mensagens, em particular, que utilizam basicamente 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.

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.