Universidade de Brasília

Programação Paralela

Relatório do Exercício 2

Autor:
Alexandre Lucchesi Alencar

Professor:
George Luiz Medeiros Teodoro

24 de setembro de 2014

1 Introdução

Este relatório tem como objetivo apresentar os resultados obtidos a partir da execução do segundo exercício de programação paralela [2], que consiste na paralelização do algoritmo *Pi de Monte Carlo* utilizando a biblioteca `pthreads`. Primeiramente, os aspectos principais do algoritmo desenvolvido e a estratégia de paralelização utilizada é apresentada. 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 lançamentos (pontos) e o número de *threads*. O código-fonte completo deste trabalho (incluindo os arquivos LaTeX que compõem este relatório) estão publicamente disponíveis no GitHub [1].

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

Além do programa principal, foi desenvolvido um *script bash* 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 as respectivas diretivas (`-lpthread -lmath` — vide Makefile) recebe via `scanf()` dois argumentos: o número de *threads* a serem criadas e a quantidade de lançamentos a serem realizados. A saída do programa tem duas linhas: a primeira contém o valor aproximado de Pi e a segunda, o tempo de execução do algoritmo em microsegundos.

[1] `https://github.com/alexandrelucchesi/parallel-programming-ex02`

- `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_count`: 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.

2.1 Geração de números aleatórios

Como o método de aproximação do valor de pi é baseado no Método de Monte Carlo, é necessária a geração de números aleatórios de ponto-flutuante. Além disso, a função que gera esses números tem que ser *thread safe*, uma vez que o algoritmo sera paralelizado e o código chamado por várias *threads* ao mesmo tempo. Por isso, utilizou-se a função `rand_r(unsigned int *)`.

A função inicializadora, `srand(unsigned int seed)`, foi executada uma vez por *thread*. Inicialmente, utilizou-se como semente para `srand` o resultado de uma função temporal (`time(NULL)`), porém, os resultados de pi em múltiplas execuções do programa quando executado a partir do *script* estava igual. Acredita-se que isso se deve ao fato das várias execuções serem realizadas em um intervalo de tempo muito pequeno, pois, acrescentando-se um `sleep(1)` entre as chamadas, os resultados começaram a variar.

Apesar de correta, a solução com `sleep()` não é ideal, pois deixa a execução dos testes muito lenta, sobretudo se o conjunto de testes for abrangente (vide Seção X). Dessa forma, somou-se ao resultado de *time(NULL)* um fator para introduzir aleatoridade que é único entre várias execuções do programa: o valor de `getpid()`.

2.2 Política de escalonamento

Procurou-se similar a política de escalonamento estática encontrada na biblioteca OpenMP. Dessa forma, tenta-se dividir igualmente o trabalho entre as *threads* antes da criação ou execução das mesmas. Por exemplo, se a entrada do programa for 2 *threads* e 1000 lançamentos, atribui-se previamente a cada *thread* o cálculo de 500 lançamentos. Caso a divisão não seja exata, ou

seja, se a entrada do programa for 2 *threads* e 1005 lançamentos, o excedente (resto da divisão) é atribuído à primeira *thread* criada.

2.3 Parâmetros de entrada

Se o número de *threads* especificado for menor ou igual a 1, nenhuma *thread* é criada e o programa é executado de forma sequencial. É válido ressaltar que isso não implicou em redundância de código-fonte, ou seja, no *modo sequencial* é executada a mesma função cujo ponteiro seria passado para `pthread_create()`, porém, sem o *overhead* de criação e finalização de *threads*.

2.4 Tipos das variáveis

Com o objetivo de se preservar ao máximo os dados provenientes das computações e mitigar a perda de precisão por truncamento ou resultados errôneos por *overflow*, usou-se em toda a aplicação os tipos `unsigned long long int` e `long long int` para valores inteiros, e `long double` para valores de ponto flutuante. Utilizou-se `typedef`s para tornar o código menos verboso e mais legível:

```
typedef unsigned long long int ulli;
```

2.5 *Benchmark*

O *benchmarking* da aplicação foi feito a partir de uma série de funções, cujas assinaturas são apresentadas a seguir:

```
int bench(const char* filename);
bench_res* bench_sched_chunk(const char *filename, int power_of_2);
bench_res* bench_sched_thread(const char *filename, int chunk_size,
    int max_threads);
double calculate_mean(const double *vec, int size);
void write_csv(bench_res *res, benchType type, int size, FILE *fp);
void write_latex_tables(bench_res *res, benchType type, int size,
    FILE *fp);
```

A função `bench` é responsável por orquestrar as funções `bench_sched_chunk` e `bench_sched_thread` na geração dos relatórios contendo os tempos de execução do algoritmo em diferentes configurações. Essas funções realizam dois tipos diferentes de *benchmarking*: a primeira varia o tamanho do *chunk* e

as políticas de escalonamento enquanto mantém o número de *threads* fixo [2]; já a segunda mantém o tamanho do *chunk* fixo [3] enquanto varia as políticas de escalonamento e o número de *threads*. Ambas retornam uma lista de `bench_res`, que é uma estrutura de dados que encapsula as informações necessárias para a geração do relatório, definida como:

```
typedef struct bench_res {
    int chunk_size;
    double static_sorting_time;
    double static_input_sorting_time;
    double dynamic_sorting_time;
    double dynamic_input_sorting_time;
    double guided_sorting_time;
    double guided_input_sorting_time;
} bench_res;
```

Essas funções utilizam internamente 3 funções auxiliares: `calculate_mean`, `write_csv` e `write_latex_tables`. A primeira é utilizada para se calcular o tempo médio de execução em cada configuração de entrada da função `count_sort`, recebendo um vetor de tempos de execução (representados como `double`) e calculando a média aritmética desses valores [4]. As duas últimas são mecanismos de exportação dos resultados, representando-os no formato CSV ou como um conjunto de tabelas prontas para serem importadas em um arquivo `.tex` (vide Tabelas 1 e 2).

3 Resultados

Conforme explicado na seção anterior, foram gerados dois tipos de *benchmark*: um para avaliar o desempenho variando o tamanho do *chunk* e outro para avaliar o desempenho variando o número de *threads*. Como entrada para o algoritmo, utilizou-se um arquivo contendo 32768 números de ponto-flutante gerados a partir do método descrito na Seção **??**. Além disso, para cada conjunto de entradas, executou-se o algoritmo 3 vezes e calculou-se a média aritmética dos tempos de execução, a fim de se obter medidas mais

[2]Esse valor é o valor padrão atribuído pelo OpenMP de acordo com a máquina que está executando o algoritmo. No *hardware* utilizado neste trabalho, este valor padrão é 4, uma vez que o processador possui 2 *cores* em HyperThreading.

[3]Adotou-se o valor do *chunk* como sendo 256, que foi o valor ótimo obtido a partir do primeiro experimento (vide Tabela 1).

[4]A quantidade de vezes que `count_sort` deve ser executada por configuração é configurada a partir da diretiva `#define BENCH_EXEC_TIMES N`, onde N é o número de execuções

precisas. O teste foi realizado executando a aplicação com a *flag* `-bench`, que demorou 2961.464582s para executar e forneceu como saída dois arquivos: `bench_sched_chunk.tex` e `bench_sched_thread.tex` — contendo os resultados dos testes.

A Tabela 1 apresenta os resultados da primeira análise. Nota-se que em todos os casos, o tamanho de *chunk* ótimo foi 1. Isso contrariou as expectativas, pois esperava-se obter como tamanho ótimo para o *chunk* um valor que se aproximasse do tamanho da *cache*, para se beneficiar do *alinhamento de cache* [3]. Além disso, observa-se que o escalonamento estático apresenta desempenho médio superior ao dinâmico e ao guiado, e que o impacto de desempenho provocado pela leitura dos dados é irrisório, sendo em média inferior à 0.1s.

Tabela 1: Tamanho de *chunk* variável e número de *threads* fixo (em 4).

C. Size	ST	ST w/ input	DYN	DYN w/ input	GD	GD w/ input
2^0	3.588906	3.590270	3.624919	3.626081	3.754292	3.755485
2^1	3.825235	3.826580	3.898424	3.899586	3.952240	3.953837
2^2	3.749607	3.751171	3.797465	3.799064	3.883701	3.884942
2^3	3.798255	3.799627	3.720954	3.722612	3.908239	3.909431
2^4	3.756875	3.758218	3.722065	3.723301	3.825053	3.826316
2^5	3.670378	3.671616	3.719752	3.720995	3.842732	3.844046
2^6	3.724537	3.725801	3.726843	3.728427	3.873497	3.874915
2^7	3.703976	3.705490	3.735572	3.737224	3.808382	3.809649
2^8	3.700626	3.702020	3.719415	3.721091	3.887111	3.888764
2^9	3.775881	3.777658	3.697703	3.698939	3.823143	3.824359
2^{10}	3.682649	3.683839	3.712101	3.713821	3.963459	3.965141
2^{11}	3.683639	3.684850	3.675376	3.676596	3.882637	3.883888
2^{12}	3.773273	3.774547	3.784975	3.786518	3.898208	3.899412
2^{13}	3.922494	3.923783	3.929915	3.931134	3.880116	3.881456
2^{14}	4.812951	4.814442	4.832530	4.833810	4.691696	4.692826
2^{15}	6.536151	6.537288	6.547434	6.548724	6.576480	6.577817
2^{16}	6.500869	6.502155	6.557550	6.558768	6.535602	6.536718

A Tabela 2 apresenta os resultados da segunda análise. Observa-se dois valores aparentemente absurdos de tempo de resposta com escalonamento dinâmico e uma *thread*: 680.678471s e 680.679593s. No entanto, esses valores

apareceram nos resultados porque o *laptop* "dormiu" devido à inatividade enquanto executava o algoritmo.

Tabela 2: Tamanho de *chunk* fixo (em 256) e número de *threads* variável.

Threads	ST	ST w/ input	DYN	DYN w/ input	GD	GD w/ input
1	6.395153	6.396270	680.678471	680.679593	6.185817	6.187074
2	4.589843	4.591248	4.531762	4.533200	4.671461	4.672986
3	3.509808	3.511261	3.476184	3.477305	3.591881	3.593058
4	3.608319	3.610359	3.622033	3.623194	3.766382	3.767998
5	3.665674	3.667167	3.675254	3.676431	3.837303	3.838563
6	3.711966	3.713215	3.699011	3.700181	3.764481	3.766109
7	3.626323	3.627496	3.663744	3.664902	3.774158	3.775587
8	3.720559	3.721782	3.696651	3.698206	3.734909	3.736129

O *speedup* de um programa paralelo é definido como [1]:

$$S = \frac{T_{serial}}{T_{parallel}}$$

Por motivos de simplificação, a Tabela 3 apresenta a relação entre o *speedup* obtido e o número de *threads* executadas apenas para o escalonamento estático, uma vez que as outras políticas apresentaram ganhos similares. Observa-se que ao se utilizar duas *threads* foi possível obter um ganho significativo de desempenho ($\approx 72\%$). A partir de três *threads* o ganho sofreu atenuação e manteve uma certa uniformidade. Acredita-se que esse comportamento ocorreu devido as características do processador utilizado (vide Seção 1.1), que só possui 2 *cores*.

Tabela 3: *Speedup* x *threads*.

Threads	2	3	4	5	6	7	8
Speedup	0.717707	0.548823	0.564227	0.573196	0.580434	0.567042	0.581778

4 Conclusão

Este trabalho possibilitou uma maior compreensão acerca da biblioteca OpenMP e sobre algumas das dificuldades encontradas no contexto de programação paralela. O algoritmo *count sort* foi otimizado a partir da aplicação de técnicas

de programação para um maior aproveitamento dos recursos computacionais que levaram a ganhos de desempenho. Uma análise dos tempos de execução do algoritmo evidenciou ganhos de desempenho (*speedup*) de até 72% em relação à versão sequencial.

Por fim, é válido ressaltar que o programa está todo parametrizado via diretivas de pré-processamento (`#define`) e aceita parâmetros de configuração em tempo de execução, possibilitando a experimentação com diferentes entradas para o algoritmo. Além disso, o *design* da aplicação permite variar de forma fácil a política de escalonamento, o tamanho dos *chunks* e a quantidade de *threads* a serem executadas de forma não intrusiva, e favorece a inclusão de novas funções de *benchmarking* de forma modular.

Referências

[1] Peter Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[2] G. L. M. Teodoro. Programação paralela, exercício de programação 01, parallel count sort, September 2014.

[3] G. L. M. Teodoro. Programação paralela, notas de aula, September 2014.