



Universidade de Brasília

PROGRAMAÇÃO PARALELA

Relatório do Exercício 1

Autor:

Alexandre Lucchesi Alencar
09/0104471

Professor:

George Luiz Medeiros Teodoro

12 de setembro de 2014

1 Introdução

Este relatório tem como objetivo apresentar os resultados obtidos a partir da execução do primeiro exercício de programação paralela, que consiste na paralelização do algoritmo *count sort* utilizando a biblioteca OpenMP. Primeiramente, o algoritmo desenvolvido e a estratégia de paralelização utilizada é apresentada. Em seguida, é feita uma análise de desempenho comparando os tempos de execução do algoritmo em diversas configurações, isto é, variando-se a estratégia de escalonamento, o tamanho dos *chunks* e o número de *threads*. O código-fonte completo deste trabalho juntamente com os `.text` deste 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

O programa desenvolvido contempla três funcionalidades principais: geração de um arquivo de dados contendo um número arbitrário de valores de ponto-flutuante, processamento de um único arquivo de dados e a geração de um *benchmark*. Essas três funcionalidades são detalhadas nesta seção.

¹<https://github.com/alexandreLucchesi/parallel-programming-ex01>

2.1 Geração do Arquivo de Dados

O algoritmo *count sort* é um algoritmo de ordenamento que foi utilizado neste trabalho para ordenar um vetor de números de ponto-flutuante. Dessa forma, foi necessária a criação de funções para a geração de quantidades configuráveis de números de ponto-flutuante e que os gravasse em um formato apropriado para servir de entrada para o programa, possibilitando a fácil experimentação do *count sort* com diferentes entradas de dados.

A interface dessas funções é apresentada a seguir:

```
void generate_floats(FILE* out, int count);
int read_input(FILE* fp, float** vector, int* size);
float* parse_floats(char* values_str, int count);
```

A função `generate_floats` recebe um arquivo de destino e a quantidade de números que se deseja gerar. A função `read_input` recebe um arquivo (no formato gerado por `generate_floats`) e retorna em `vector` e `size` o vetor e seu tamanho, respectivamente. Por fim, a função `parse_floats` é utilizada internamente por `read_input` para realizar o *parsing* de *strings* para *floats*.

Para gerar um arquivo de dados, `vec.dat`, contendo, por exemplo, 1000 elementos, basta executar o binário passando-se a *flag* `-gen`, conforme descrito a seguir:

```
$ ./a.out -gen vec.dat 1000
```

2.2 Processamento

O processamento de um arquivo de dados foi encapsulado na função `process`, cuja assinatura é apresentada a seguir:

```
int process(float **vec, int *vec_size, double *time_sorting_only,
            double*time_with_input, const char *filename, omp_sched_t kind,
            int chunk_size, int thread_num);
void count_sort(float a[], int n, omp_sched_t kind, int chunk_size);
```

Os quatro primeiros parâmetros: `vec`, `vec_size`, `time_sorting_only` e `time_with_input` são, na verdade, retornos da função `process`, e retornam respectivamente, o vetor de *floats* ordenado, o tamanho do vetor, o tempo transcorrido considerando apenas a execução do *count sort* e o tempo transcorrido desde a entrada de dados até o término da execução da função de ordenação. Os demais parâmetros são utilizados para parametrizar a execução da função `count_sort` e equivalem, respectivamente, ao nome do arquivo

de dados, à política de escalonamento do OpenMP (*static*, *dynamic*, *guided* ou *auto*), ao tamanho do *chunk* e ao número de *threads* a serem criadas.

Para processar um único arquivo, basta executar o binário passando-se o arquivo de dados e *flags* opcionais de seleção da política de escalonamento, tamanho do *chunk* ou quantidade de *threads*. Por padrão, utiliza-se **static**, tamanho do *chunk* 1 e o número padrão de *threads* do OpenMP (depende da máquina sendo executada). O trecho abaixo ilustra o processamento do arquivo `vec.dat`, com escalonamento dinâmico e tamanho de *chunk* 4:

```
$ ./a.out vec.dat -k dynamic -c 4
```

Encapsular a lógica de processamento na função *process* permitiu a automatização do *benchmarking* da aplicação, por meio do desenvolvimento da função **bench** (descrita a seguir), evitando trabalho manual.

Um dos objetivos deste trabalho consistiu na experimentação da biblioteca OpenMP. O trecho de código a seguir apresenta a funcionalidade central da função `count_sort` e a estratégia de paralelização adotada:

```
...
    // Set number of threads to be created.
    if (num_threads > 0) omp_set_num_threads(num_threads);
#pragma omp parallel
    {
        int count; // *Private* variable.

        // Set scheduling policy and chunk size at runtime.
        omp_set_schedule(kind, chunk_size);

#pragma omp for schedule(runtime)
        for (int i = 0; i < n; i++) {
            count = 0;
            for (int j = 0; j < n; j++)
                if (a[j] < a[i])
                    count++;
            else if (a[j] == a[i] && j < i)
                count++;
            temp[count] = a[i];
        }
    }
...
```

A variável `count` é declarada dentro de uma seção paralela e, portanto, é atribuída um escopo privado ². As funções `omp_set_num_threads` e `omp_set_schedule` são padrões do OpenMP e consistem em uma forma dinâmica de se configurar, respectivamente, a quantidade de *threads*, e a política de escalonamento e o tamanho do *chunk*. Esses parâmetros são utilizados em tempo de execução para atribuir tarefas às *threads* dentro do `for` (observe a clause `schedule(runtime)`). O vetor `temp` é uma variável compartilhada que conterá a sequência de números ordenada no término das iterações. É importante notar que não nenhum mecanismo de sincronização (como a utilização de *locks* ou declarar `temp` como *critical*) foi necessário para a criação de uma seção crítica em torno de `temp`, uma vez que `count` retorna um valor diferente (indexando uma posição de memória diferente) para cada *thread* e, portanto, não há condições de corrida.

2.3 *Benchmark*

O *benchmarking* da aplicação é 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, int size, FILE *fp);
void write_latex_tables(bench_res *res, 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 *benchmark*: a primeira varia o tamanho do *chunk* e as políticas de escalonamento enquanto mantém o número de *threads* fixo ³; já a segunda mantém o tamanho do *chunk* fixo ⁴ 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:

²Poderia-se declarar esse escopo via *clauses* dentro do `pragma`.

³Esse 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.

⁴Adotou-se o valor do *chunk* como sendo 256, que foi o valor ótimo obtido a partir do primeiro experimento (vide Tabela 3).

```
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 ⁵. 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` `reftable:bench-sched-chunk`, `table:bench-sched-thread`.

3 Resultados

Conforme explicado na seção anterior, foram realizados 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-flutuante gerados a partir do método descrito na Seção 2.1. 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 precisas. O teste foi realizado executando-se a aplicação com a *flag* `-bench`, que forneceu a seguinte saída:

```
$ ./a.out -bench vec.dat
=====
Benchmarking: varying chunk size and scheduling policy
=====
Looping...
Performing iteration 0, chunk size is 1.
Performing iteration 1, chunk size is 2.
```

⁵A 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

```

Performing iteration 2, chunk size is 4.
Performing iteration 3, chunk size is 8.
Performing iteration 4, chunk size is 16.
Performing iteration 5, chunk size is 32.
Performing iteration 6, chunk size is 64.
Performing iteration 7, chunk size is 128.
Performing iteration 8, chunk size is 256.
Performing iteration 9, chunk size is 512.
Performing iteration 10, chunk size is 1024.
Performing iteration 11, chunk size is 2048.
Performing iteration 12, chunk size is 4096.
Performing iteration 13, chunk size is 8192.
Performing iteration 14, chunk size is 16384.
Performing iteration 15, chunk size is 32768.
Performing iteration 16, chunk size is 65536.
Generating output latex tables...
File 'bench_sched_chunk.tex' successfully generated! :-)

```

```

=====
Benchmarking: varying thread count and scheduling policy
=====

```

```

Looping...
Performing iteration 0, thread count is 1.
Performing iteration 1, thread count is 2.
Performing iteration 2, thread count is 3.
Performing iteration 3, thread count is 4.
Performing iteration 4, thread count is 5.
Performing iteration 5, thread count is 6.
Performing iteration 6, thread count is 7.
Performing iteration 7, thread count is 8.
Generating output latex tables...
File 'bench_sched_thread.tex' successfully generated! :-)

```

Execution time: 2961.464582 seconds.

A Tabela 3 apresenta os resultados da primeira análise. Nota-se que em todos os casos, o tamanho de *chunk* ótimo foi 1. 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 nesse caso (em média, inferior à 0.1s).

A Tabela 3 apresenta os resultados da segunda análise.

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

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:

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

A Tabela 3 apresenta a relação entre o *speedup* obtido e o número de *threads* executadas.

Tabela 3: *Speedup* x *threads*.

<i>Threads</i>	1	2	3	4	5	6	7	8
<i>Speedup</i>	0.003256	0.003484	0.003181	0.003339	0.003351	0.003585		

4 Conclusão

Por fim, é válido ressaltar que o programa está todo parametrizado via diretivas de pré-processamento (**#define**) e aceita alguns parâmetros em tempo de execução (como o tamanho do *chunk* e a política de escalonamento), possibilitando a experimentação com diferentes configurações. O *design* escolhido permite variar de forma fácil a política de escalonamento, o tamanho dos *chunks* e a quantidade de *threads* de forma não intrusiva e modular. Entretanto, é importante frisar que o *design* da aplicação favorece a inclusão de novas funções de *benchmark* de forma fácil.

Referências