Work Assignment - Phase 1

1st Maria Inês Matos
Mestrado em Engenharia Informática
PG61533

2nd Hugo Simões Marques
Mestrado em Engenharia Informática
PG60266

I. Introdução

Neste relatório vamos descrever o processo de transformação de um código sequencial para a sua versão paralela utilizando, principalmente, o OpenMP. O objetivo principal deste trabalho é acelarar a execução do programa tendo sempre em atenção para evitar o acontecimento de *data races*.

II. Identificação de *Hot-Spots*

Para iniciar o processo de otimização, executámos a versão sequencial original do código para estabelecer um tempo de referência. Como mostra a Fig. 1, vimos que a duração total é de 76.30s, 46.77s deles para a função do *spec.advance* e 12.64s para o *emf. advance*.Também é possível observar que o valor final da energia é *3.001722e+02*.

```
[uminhocp044@cna0257 zpic_CompParalela]$ ./zpic
Starting simulation ...

n = 0, t = 0.0
Energy (fields | particles | total) = 3.000000e+02 0.000000e+00 3.000000e+02
n = 42106, t = 40.000702

Simulation ended.

Energy (fields | particles | total) = 2.985474e+02 1.624786e+00 3.001722e+02
Initial energy: 3.000000e+02, Final energy: 3.001722e+02

Final energy different from Initial Energy. Change in total energy is: 0.06 %
Time for spec. advance = 46.777178 s
Time for emf   advance = 12.637492 s
Total simulation time  = 76.303086 s

Particle advance [nsec/part] = 246.826067
Particle advance [Mpart/sec] = 4.051436
```

Fig. 1. *Output* do código original.

Posteriormente, utilizámos as ferramentas Score-P e Cube para podermos visualizar com precisão as funções e regiões do código que representavam os maiores *hot-spots*.

É importante referir que, como mostra a Fig.2, os tempos de execução registados pelo Score-P são superiores aos observados na execução normal do código (Fig.1) devido às operações adicionais de monitorização e recolha de dados de desempenho. Apesar desta distorção, a análise mantém a sua validade para identificar as regiões críticas.

A análise identificou a função *spec_advance* (150.50 s) como o principal *hot-spot*. Dentro desta, destacam-se as funções *interpolate_fld* (46.47 s), *ltrim* (42.79 s) e *dep_current_zamb* (46.44 s). A função *spec_set_u* (5.54 s) também apresentou tempo relevante, todas no ficheiro *particles.c*.

No ficheiro *current.c*, a função *kernel_x* registou 16.72 s. No ficheiro *emf.c*, as funções *yee_e* (7.22 s), *yee_b* (3.57 s) e *emf_move_window* (1.92 s) completam as regiões críticas identificadas.

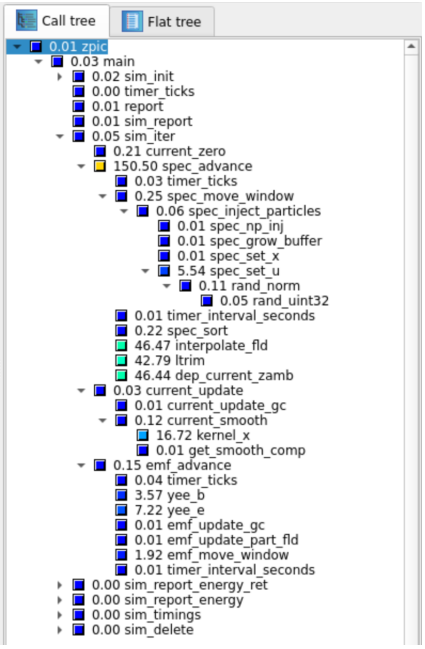

Fig. 2. *Hot-Spots* demonstrados no *Cube*.

III. Análise e Implementação

A. *spec_advance*

Ao analisar o *spec_advance*, identificamos que ele consiste num *loop* principal seguido de outras estruturas de repetição. Paralelizamos facilmente o *loop* principal, adicionando a *flag reduction(+:energy)* para garantir que a variável *energy* mantivesse o valor correto no final.

Para as outras secções, convertemos o *loop while* num *for*, ajustando o contador i antes de cada *continue* para manter o comportamento original. O outro for também foi paralelisado.

Dentro do *loop* principal do *spec_advance*, três funções são criticalmente importantes:

ltrim: executa um cálculo matemático simples

interpolate_fld: realiza operações matemáticas mais complexas

dep_current_zamb: identificada como a principal candidata a optimizações

B. dep_current_zamb

A função *dep_current_zamb* foi identificada como principal candidata para optimização, uma vez que sua implementação original utilizava um *loop* que iterava 1 ou 2 vezes com *overhead* significativo, além de operações matemáticas redundantes e acesso subóptimo à memória.

Foram aplicadas optimizações matemáticas, substituindo divisões por multiplicações e eliminando cálculos repetidos. O acesso à memória foi melhorado para ser mais eficiente.

C. spec_set_u

No *spec_set_u*, implementámos uma estratégia de paralelização que reutiliza *threads* entre os três loops sequenciais, melhorando a eficiência geral.

D. kernel_x

O *kernel_x* era um ponto particularmente lento e enganador. Apesar do loop principal parecer sequencial, conseguimos melhorar drasticamente o seu desempenho através de:

- Reorganização de variáveis
- Uso de um array temporário para evitar *data races*
- Paralelização dos loops subsequentes

E. Emf Advance

O *emf_advance* é o último ponto crítico relevante no código. Esta função funciona principalmente como um organizador, chamando outras funções sequencialmente. Das várias funções chamadas, focámos em optimizar três:

yee_b e *yee_e*: paralelizadas diretamente nos seus loops

emf_move_window: requerindo uma abordagem mais elaborada

F. emf_move_window

A primeira versão que tentamos da função *emf_move_window* utilizava uma paralelização direta, o que resultava em *data races*. Estes ocorriam porque a operação de deslocamento *E[i] = E[i+1]* cria uma cadeia de dependências onde múltiplas threads acediam concorrentemente às mesmas posições de memória.

Para resolver este problema, desenvolvemos uma abordagem baseada em fases sincronizadas com partições não-sobrepostas. A solução divide a operação em três fases distintas:

Fase de Leitura Paralela: Cada *thread* lê os dados da sua partição exclusiva para armazenamento temporário

Barreira de Sincronização: Garante que todas as leituras são completadas antes de iniciar escritas

Fase de Escrita Paralela: Cada *thread* escreve os dados deslocados para as suas posições finais

Esta solução elimina conflitos usando uma dupla estratégia: sincronização entre fases e divisão exclusiva de memória. Assim, obtemos sempre o mesmo resultado correto, independentemente de como as threads são distribuídas pelo sistema.

IV. Resultados do Benchmark

O *benchmark* foi realizado através de *perf stat -r 5*, que nos deu a média de cinco corridas do código para cada número de threads.

Os resultados do *benchmark* mostram o desempenho da implementação paralela em relação ao número de threads utilizadas. A Tabela I apresenta os dados completos do *benchmark*, mostrando o tempo total de execução e os speedups obtidos para diferentes números de *threads*. O speedup relativo é calculado em relação ao tempo base original de 76.30 s.

TABLE I
Resultados do Benchmark

Threads	Tempo Total (s)	Speedup (Relativo)	Speedup (Base 76.30 s)
1	69.830	1.00	1.09
2	37.145	1.87	2.05
3	27.448	2.54	2.78
4	22.828	3.05	3.34
5	19.618	3.55	3.89
6	18.075	3.86	4.22
7	16.922	4.12	4.51
8	16.155	4.32	4.72
9	15.525	4.49	4.91
10	15.266	4.57	5.00
11	14.706	4.74	5.19
12	14.428	4.83	5.29
13	14.165	4.92	5.39
14	14.283	4.88	5.34
15	14.416	4.84	5.29
16	14.445	4.83	5.28
17	14.104	4.95	5.41
18	14.435	4.83	5.29
19	14.445	4.83	5.28
20	14.925	4.67	5.11

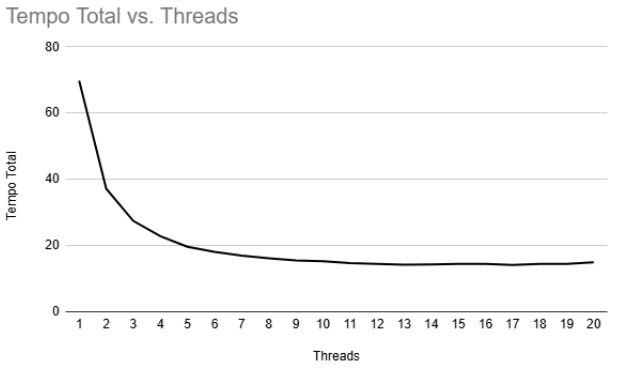

Fig. 3. Gráfico de tempo em relação a Threads.

Como podemos observar, o menor tempo de execução foi alcançado com 17 *threads*. No entanto, o ganho de desempenho em relação à configuração com 13 *threads* é de apenas 0.065 segundos, uma diferença insignificante. Considerando o custo computacional de utilizar quatro *threads* adicionais para um benefício tão ínfimo, conclui-se que 13 *threads* representam o ponto de equilíbrio ideal para esta aplicação.

CONCLUSÃO

Este trabalho permitiu aprofundar significativamente os conhecimentos do OpenMP e de progamação paralela. O maior desafio foi adicionar paralelismo a alguns *hot-spots* em que era necessário garantir a ausência de *data races*, particularmente em funções complexas como a *emf_move_window*. Este trabalho ajudou bastante a compreender que nem todas as partes do código beneficiam de paralelização. Os resultados finais demonstram uma aceleração significativa, confirmando o sucesso da estratégia implementada.