Relatório TP1 Computação Paralela

1st Hugo dos Santos Martins
Universidade do Minho
a95125
Barcelos, Portugal
a95125@alunos.uminho.pt

2nd João Bernardo Teixeira Escudeiro
Universidade do Minho
a96075
Valpaços, Portugal
a96075@alunos.uminho.pt

***Abstract*—Este documento contém uma breve descrição do trabalho realizado na primeira fase do trabalho prático da unidade curricular de COmputação Paralela do 1º ano do Mestrado em Egenharia Informática.**

I. Introdução

Este documento contém um resumo acerca de todo o processo seguido pelo grupo, desde o produto inicial até ao produto final, bem como são explicadas as metodologias para superar cada uma das dificuldades que foram surgindo ao longo do desenvolvimento do trabalho prático.

II. Sinopse

Após identificar os objetivos da primeira etapa do trabalho prático, o grupo deu prioridade à conclusão das fichas que não tinham sido finalizadas durante as aulas, uma vez que o conteúdo das fichas e os objetivos de aprendizagem eram semelhantes com o que nos era requerido nesta fase inicial do projeto. Os tópicos predominantes na matéria que constituía o cerne do trabalho envolviam a Hierarquia de Memória, paralelização e Vetorização.

Concluída a elaboração das fichas, o grupo decidiu dedicar-se à análise do código disponibilizado, procurando compreender o propósito do mesmo e examinando minuciosamente as funcionalidades de cada função, bem como o resultado gerado pelo programa. O código em questão consistia num conjunto de fórmulas físicas que simulavam o movimento de partículas à medida que o tempo avançava.

Após a leitura atenta da documentação associada a cada função e a identificação de algumas inconsistências no código, o grupo direcionou o seu trabalho para a simulação, fazendo uso dos dados fornecidos no ficheiro denominado *input-data.txt*. O conteúdo desse arquivo especificava o tipo de gás a ser utilizado na simulação, a temperatura inicial e a pressão. Constatamos prontamente que o programa demandaria um alto período de tempo para a sua execução.

Nesse ponto, o grupo procedeu para a fase de análise de desempenho, fundamentando-se na seguinte equação, reconhecendo que a redução do tempo de execução dependia crucialmente da minimização do número de instruções:

$$Texec = I * CPI * Tcc \quad (1)$$

III. Procedimentos

Após correr pela primeira vez na máquina nativa e observar o ficheiro de outputs, o grupo decidiu correr novamente o programa, desta vez utilizando o Search, local onde seria avaliado o projeto. No Search, o grupo recorreu à ferramenta **Perf**, que permite analisar o desempenho do programa e fornece um leque de informações acerca da execução do programa, entre elas o número de tempo utilizado, o número de Ciclos e o de Instruções.Na tabela abaixo, bem como na figura 1 podemos observar os valores de desempenho obtidos inicialmente.

TABLE I
Resultados de Performance Iniciais

Tempo(s)	***Intruções***	***Ins/Ciclo***	***Ciclos***
333,28	$1 * 10^12$	1.23	$1 * 10^12$

Dado isto, o grupo decidiu começar por tentar descobrir o que causaria um número tão grande de instruções. Com recurso ao GPROF, detetamos que as funções que estariam a consumir mais tempo de exucução eram a *Potencial()* e a *ComputeAccelaration()*. Detetamos que na função Potencial estavam a ser realizados cálculos com potências de grau 12, bem como raízes que poderiam ser simplificadas. Após simplificar as equações no papel, o grupo passou as novas equações para o código e tentou perceber se estas alterações trariam benefícios no desempenho do programa.

TABLE II
Resultados de Performance Após Simplificar Cálculos

Tempo(s)	***Intruções***	***Ins/Ciclo***	***Ciclos***
119.52	$620 * 10^9$	1.68	$369 * 10^9$

Observámos bastantes melhorias, principalmente no número de instruções e no tempo de execução, sendo que este último passou para metade e o número de intruções reduziu cerca de 100x. O passo seguinte consistiu na simplificação ao máximo das operações de multiplicações, tentar evitar contas que fossem repetidas e dar enfâse à hierarquia de memória, tendo especial atenção ao número de cache misses e tentar perceber a melhor forma para ”percorrer” as matrizes minimizando o número de cache misses. Após as alterações no código, prin-

cipalmente na função *Potencia()* o grupo obteve os resultados em baixo.

TABLE III
RESULTADOS DE PERFORMANCE APÓS SIMPLIFICAÇÃO DE CÁLCULOS, E PARALELIZAÇÃO DE INSTRUÇÕES.

Tempo(s)	***Intruções***	***Ins/Ciclo***	***Ciclos***
52,43	$300 * 10^9$	1.84	$162 * 10^9$

As alterações realizadas até este momento garantiam que já existiam melhorias na ordem do 50% ao nível de desempenho,no entanto ainda havia bastantes aspetos passíveis de melhoria. O grupo decidiu então, incluir flags de otimização. Optamos pela flag **O3** que ajuda a otimizar o desempenho do código durante a compilação. Com esta inclusão seria de prever que existiriam melhorias no desempenho.

TABLE IV
RESULTADOS DE PERFORMANCE APÓS UTILIZAR A FLAG 03.

Tempo(s)	***Intruções***	***Ins/Ciclo***	***Ciclos***
12,44	$55 * 10^9$	1.66	$33 * 10^9$

A inclusão da flag no processo de compilação do código traz melhorias a todos os níveis e prevê-se que com a utilização de mais flags o tempo de execção tenda a diminuit. O grupo notou que existiam ciclos que executavam apenas 3 vezes e que podiam ser simplificados o que poderia ajudar na redução das operações. Também incluímos duas flags adicionais que foram abordadas nas aulas práticas **-ftree-vectorize** e **-msse4**.Os resultados obtidos estão na tabela 5.

TABLE V
RESULTADOS DE PERFORMANCE APÓS REDUÇÃO DE COMPLEXIDADE E UTILIZAÇÃO DE NOVAS FLAGS.

Tempo(s)	***Intruções***	***Ins/Ciclo***	***Ciclos***
11.05	$45 * 10^9$	1.41	$32 * 10^9$

Dado isto e como foi falado nas aulas, o grupo decidiu utilizar vectorização, ao invés do uso de matrizes. Estas matrizes de ordem 5000 por 3, seriam substituidos por arrays de 15000 posições que seriam acedidos de forma próxima e que aumentaria o desempenho nos acessos à memória, visto que estes seriam realizados de forma local.Na tabela 6 temos os resultados de performance obtidos no Search.

TABLE VI
RESULTADOS DE PERFORMANCE APÓS VETORIZAR O CÓDIGO .

Tempo(s)	***Intruções***	***Ins/Ciclo***	***Ciclos***
18	$56 * 10^9$	1.04	$54 * 10^9$

Ao realizar a vetorização reparamos que o tempo tinha aumentado. Uma possível melhoria era juntar as funções *Potencial()* e ComputeAccelarations(), visto que a função potencial poderia ser incluída nos ciclos da função compute.Isto deve-se ao facto de serem realizadas contas semelhantes, bem como os ciclos serem semelhantes. Seria de prever que o número de operações diminuísse. Após testar no cluster os resultados obtidos foram :

TABLE VII
RESULTADOS DE PERFORMANCE APÓS JUNTAR AS DUAS FUNÇÕES QUE MAIS TEMPO DISPENDIAM .

Tempo(s)	***Intruções***	***Ins/Ciclo***	***Ciclos***
9	$31 * 10^9$	1.17	$26 * 10^9$

Tendo em conta os resultados fornecidos, facilmente percebemos que desde o início conseguimos obter melhorias significativas.No entanto achamos que ainda seria possível otimizar mais o programa para tentar baixar ainda mais o tempo e o número de instruções, que era o maior causador do atraso(Figura 3). Identificamos que os cálculos utilizados para calcular os valores de potencial poderiam ainda ser mais simplificados para remover ao máximo instruções desnecessárias.Os resultados que obtivemos foram :

TABLE VIII
RESULTADOS DE PERFORMANCE APÓS OTIMIZAR OS CALCULOS DE POTENCIAL .

Tempo(s)	***Intruções***	***Ins/Ciclo***	***Ciclos***
5,69	$21 * 10^9$	1.32	$16 * 10^9$

Basendo-se nestes resultados, o grupo acredita que os mesmos são bastante aceitáveis, pelo que ainda existiriam alguns aspetos passíveis de melhoria caso existisse mais tempo disponível.No entanto achamos que o trabalho foi cumprido na íntegra, onde tentamos ao maximo otimizar o programa,obtendo bons resultados comparativamente aos resultados iniciais. O grupo focou-se no fim na organização e na legibilidade do código para que fosse percetível, para quem lê, o que cada função faz.

Dado como concluída esta fase o grupo autoavalia o seu trabalho de forma positiva, tendo em conta o resultado inicial e o resultado final. As maiores dificuldades foram a adaptação à máquina da escola *Search*, e também a identificação de aspetos passíveis de melhoria,com os resultados já otimizados, visto que é atingido um ponto em que fica dificíl encontrar pontos-chave.

IV. ESTRUTURAÇÃO FINAL.

Será entregue um zip, cujo conteúdo será este relatório, um *Makefile* capaz de compilar o código utilizando as flags fornecidas de uma forma rápida.Para além disto existe um ficheiro de texto *inputdata* que contém o input para o programa e uma subdirectoria *src*, que contém o ficheiro de código final, com as alterações todas. Para exexcutar o programa basta realizar o comando *make* e para correr baste realizar o comando *make run*. Caso pretenda correr e observar a performance,o grupo criou uma alternativa, basta correr o programa utilizando *make run1*.

V. Anexos

```
Performance counter stats for './MD.exe':

        333284,02 msec task-clock                #    1,000 CPUs utilized
              118      context-switches          #    0,000 K/sec
                0      cpu-migrations            #    0,000 K/sec
              939      page-faults               #    0,003 K/sec
     1015438060944      cycles                    #    3,047 GHz
      672796846808      stalled-cycles-frontend   #   66,26% frontend cycles idle
     1244240340813      instructions              #    1,23  insn per cycle
                                                  #    0,54  stalled cycles per insn
      129398933001      branches                  #  388,254 M/sec
         432557608      branch-misses             #    0,33% of all branches

     333,280997297 seconds time elapsed

     333,282615000 seconds user
       0,002000000 seconds sys


[a96075@search7edu2 code_inicial]$
```

Fig. 1. Tempo de execução inicial.

```
Performance counter stats for './MD.exe':

         11047,45 msec task-clock                #    1,000 CPUs utilized
               22      context-switches          #    0,002 K/sec
                0      cpu-migrations            #    0,000 K/sec
              451      page-faults               #    0,041 K/sec
      32036755269      cycles                    #    2,900 GHz
      19785190240      stalled-cycles-frontend   #   61,76% frontend cycles idle
      45194653019      instructions              #    1,41  insn per cycle
                                                 #    0,44  stalled cycles per insn
       2363798251      branches                  #  213,968 M/sec
          1539666      branch-misses             #    0,07% of all branches

     11,052480556 seconds time elapsed

     11,045911000 seconds user
      0,002000000 seconds sys


[a96075@search7edu2 code_escudas]$
```

Fig. 2. Tempo de execução antes de fazer a vetorização.

```
rnorm=sqrt(r2);
quot=sigma/rnorm;
term1 = pow(quot,12.);
term2 = pow(quot,6.);

Pot += 4*epsilon*(term1 - term2);
```

```
aux2=rSqd*rSqd*rSqd;
term2=1. /aux2;
Pot += 8*term2*(term2-1.);
f=(48.-24.*aux2)/(aux2*aux2*rSqd);
```

Fig. 3. Simplificações que foram feitas.

```
Performance counter stats for './MD.exe':

          5694,90 msec task-clock                #    0,999 CPUs utilized
                21      context-switches          #    0,004 K/sec
                 0      cpu-migrations            #    0,000 K/sec
               394      page-faults               #    0,069 K/sec
      16514717021      cycles                    #    2,900 GHz
       9903949539      stalled-cycles-frontend   #   59,97% frontend cycles idle
      21741130549      instructions              #    1,32  insn per cycle
                                                 #    0,46  stalled cycles per insn
        482605495      branches                  #   84,743 M/sec
           567682      branch-misses             #    0,12% of all branches

     5,699681295 seconds time elapsed

     5,694318000 seconds user
     0,000999000 seconds sys


[a96075@search7edu2 code_escudas]$
```

Fig. 4. Resultado Final.