Computação Paralela 23/24
Fase 2

João Castro - pg53929
Vasco Rito - a98728

***Key Words*—Paralelismo, Escalabilidade, OpenMP, Memória Partilhada, Threads, Secções Críticas, Data Races, Profiling**

I. Introdução

Para a realização da segunda fase do projeto de Computação Paralela foi-nos proposto explorar o paralelismo em memória partilhada recorrendo ao OpenMP de forma a optimizar a execução de tempo da aplicação de simulação de partículas.

II. Versão Sequencial

Em relação à versão sequencial apenas foi retirado um if statement dentro da função **potAccWork** que estava a afetar a legibilidade do nosso código e já não era necessário uma vez que já tínhamos fundido a função Potential na **potAccWork**.

III. Profiling

Em primeiro lugar, procedeu-se ao aumento do **N** (aumento do número de partículas) para 5000, e realizou-se o *profiling* do código, já tendo em conta as várias otimizações que se colocaram em prática na fase 1. Recorrendo à ferramente **gprof** fez-se uma análise da *performance* da versão sequencial do nosso programa.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 99.99     30.34    30.34      201   150.92   150.92  potAccWork()
  0.10     30.37     0.03                             VelocityVerlet(double, int, _IO_FILE*)
  0.00     30.37     0.00     7500     0.00     0.00  frame_dummy
  0.00     30.37     0.00        1     0.00     0.00  _GLOBAL__sub_I_N
```

Fig. 1. Flat profile

Como podemos observar a maior parte da computação é realizada na função **potAccWork()**, responsável por 99.99% do tempo de execução do código(30.34 segundos apenas na sua execução- *self seconds*). Desta forma, procedeu-se à paralelização desta função de modo a aumentar a performance do código.

IV. Paralelização

O passo seguinte consistiu na identificação das secções críticas que poderiam, mais tarde, originar *data races* furtuito de acessos múltiplos a variáveis partilhadas por diferentes *threads*. Após essa análise, verificámos que o incremento da variável Pot, assim como as operações realizadas no *array* das acelerações **a** teriam de ser protegidas devido a possíveis escritas de *threads* ao mesmo tempo, ou escritas antes de leituras atualizadas, etc.

A opção pela diretiva *reduction* foi feita visando melhorar o desempenho. Isto ocorre porque garantimos que cada *thread* na região paralela irá ter sua própria cópia privada da variável. Em seguida, as operações são realizadas independentemente, e no final, os resultados parciais são combinados. Este método mostra-se mais eficiente em termos de *performance*. Desta forma, podemos deixar para o OpenMp a responsabilidade das agregações serem feitas de forma correta e consistente. Em versões mais recentes do OpenMp é possível utilizarmos a diretiva *reduction* sob valores que não são escalares, o que será uma mais valia, já que outras diretivas como o *critical*, ou *atomic* geram um *overhead* de sincronização o que acaba por afetar o tempo de execução e a escalabilidade do nosso programa. Assim, em primeiro lugar foi criada uma região paralela de forma a criar um conjunto de *threads* que irão executar o nosso bloco de código: **#pragma omp parallel reduction(+:Pot) reduction(+:a[:N*3])**. Testámos a *reduction* quer numa versão de duas dimensões, quer numa versão de um array de uma dimensão de **a** e, verificou-se que na versão de um array de uma dimensão, o CPI era ligeiramente inferior devido a um possível menor *overhead* associado ao *indexing*, ou até pela localidade já que, nessa versão, ocorreram menos *cache misses* (1 dimensão: CPI-1.0 e cerca de menos 200k *cache misses* nível L1; 2 dimensões: CPI-1.1). De seguida, introduzimos a diretiva **#pragma omp for schedule(dynamic,1) collapse(2)**, uma vez que pretendemos paralelizar o *loop* e dividir da forma mais justa possível o *workload* das iterações do *loop* dinamicamente. Ao fazermos *dynamic 1*, garantimos que quando as *threads* que acabarem a sua execução são escalonadas para executar uma nova iteração. Se o número fosse superior a 1, a diferença de trabalho seria superior podendo aumentar o *IDLE time* de outras *threads*. Por fim, o *collapse(2)* permite combinar *nested loops* num único *loop* o que ajuda a melhorar a eficiência da paralelização.
A nossa estratégia de paralelização viza em atribuir para cada *thread* um bloco de posições de forma a balancear a carga e a minimizar o *overhead* de sincronização. Logo, para esse efeito decidiu-se aumentar um pouco o tamanho do bloco para não gerar *overhead* de paralelismo *fine-grained* o que leva a maiores tempos de execução devido à elevada sincronização e coordenação entre *threads*.

V. Análise da Escalabilidade

Após termos executado o nosso programa com diferentes números de *threads* e de ter feito um *profiling* do código, acreditamos poder estar na presença de alguns problemas de escalabilidade. Eventualmente, este código paralelizado poderia estar a ser impactado por *Memory Wall* já que existe um aumento do número de *cache misses*, assim como um aumento do número do CPI. Isto poder-se-ia apreender pelo facto de existir um grande número de acessos à memoria derivados de operações com *arrays* e matrizes, ou de uma redundância de cálculos, ou até de falta de localidade espacial. No entanto, o facto de termos implementado hierarquia de memória, recorrendo a *loop block optimization*, não acreditamos que este seja um problema que tenha tido um maior impacto na escalabilidade.
Quanto à *task granularity* foram testados diferentes valores para o *blockSize* e ajustámo-lo de forma a que não fosse um valor pequeno, visto que lançar uma *thread* por cada um desses blocos iria conduzir em um *overhead* de sincronização e de mudanças de contexto, já que como cada thread possui informações de registos e de estado essenciais para a retoma da exeçução da nova thread isso iria apresentar um custo acrescido, pelo que aumentámos o valor para 96. Apesar de termos obtido um melhor desempenho não podemos assegurar que esse tamanho do bloco para a execução desta tarefa seja o ideal/óptimo. Verificámos que essa melhoria é mais notória até o cenário de paralelismo de até 16 *threads* com uma diferença de tempo de execução de aproximadamente 1 segundo a menos.

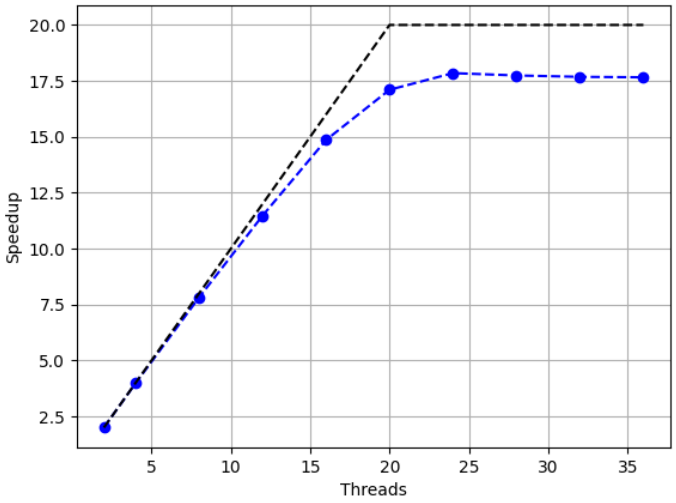

Fig. 2. Gráfico de análise de escalabilidade

No gráfico representado acima podemos averiguar que o *speedup* ideal é atingido sensívelmente entre as 2 e 12 *threads*, sendo que com 12 *threads* o desempenho já não é o ideal, uma vez que, o expectável é obtermos um *speedup* proporcinal ao número de PU's atribuídos. Como o *cluster* onde foram obtidos os *outputs* apresentam 1 *thread* por *core*, logo seria expectável obter um speedup=12. Como o presente sistema possui 20 PU's físicos o *speedup* nunca irá ultrapassar os 20. Por outro lado, podemos identificar que o nosso programa irá ter um maior *speedup* com 24 threads, ainda que seria necessário testar efetivamente com outro número de threads que não estão representados no gráfico. A partir das 24 threads há uma diminuição do *speedup*, ainda que os valores se mantenham constantes.

VI. Problemas de Escalabilidade

A. Serial work (Amdahl's law)

Na função **PotAccWork**, a região paralela apresenta reduções da variável **Pot** e do *array* **a** o que introduz serialização. Segundo a lei de *Amdahl*, a existência de frações de código serializadas limitam a escalabilidade. Ora, dentro dos *loops* podemos averiguar que a maior parte do trabalho é realizado com operações efetuadas de leitura e escrita da memória (no *array* das acelerações **a**) o que poderá conduzir numa possível limitação do nosso código.

B. Memory wall

Existe um aumento do CPI com o aumento do número de threads, pelo que poderia ser indicativo de um problema de *Memory Wall*, porém foi explorada a localidade espacial com a introdução de otimizações de *block loop*. Este aumento poderá advir da redundância de cálculos, na soma do quadrado das diferenças e Pot.

C. Parallelism/task granularity

Uma das razões que pode conduzir a este problema é a existência de tarefas pequenas que introduzem um maior overhead. Assim, como já foi referido optámos por aumentar o tamanho do blockSize, porém seria necessário testar outros valores pois podemos estar perante um paralelismo fine-grained.

D. Synchronisation overhead

Ao paralelizar o nosso código evitámos usar diretivas do tipo *atomic*, ou *critical*, pois estas têm uma maior carga de sincronização e posterior pior desempenho, pelo que optámos pela redução. O facto de *threads* acederem aos seus valores locais na mesma *cache line*, pode levar à invalidação da linha provocando maior número de *stalls* e consequentemente maior CPI. Para combater o *false sharing* teríamos de alinhar estas variáveis locais e averiguar se de facto este se tratava de um problema de escalabilidade no problema que temos em mãos.

E. Load imbalance

No que toca ao balanceamento de carga optámos por atribuir um bloco a cada *thread* e recorrendo a um escalonamento **dynamic,1**, de forma a reduzir ao máximo o *IDLE time* de outras *threads* que não estejam a efetuar trabalho. No entanto, para fazer um diagnóstico mais preciso seria necessário medir o tempo de computação de cada *thread*, de forma a verificar se existe um desfazamento grande de trabalho entre as várias *threads* ou não.

VII. Outras técnicas de análise de performance

Através da técnia de *profiling*, **perf record**, é possível fazer uma recolha de amostras para cada função ou sinais, com a respetiva estimativa de desempenho para cada uma delas. Como podemos verificar 1.08% de 1251 amostras são atribuídas à **gomp_team_barrier_wait_end**, que está relacionado com um *sinal wait* para a sincronização de *threads*, a barreira no fundo não irá permitir que as *threads* avancem sem que todas as outras tenham feito o seu trabalho. Ora 1% no contexto de desempenho não será muito significativo, porém na função **potAccWork** é onde a maior parte da execução de tempo do nosso programa é gasto.

```
# Total Lost Samples: 5989
#
# Samples: 116K of event 'cycles:uppp'
# Event count (approx.): 68770187738
#
# Overhead       Samples  Command    Shared Object     Symbol
# ........  ............  .........  ................  ........................................
#
    96.42%        111860  MDpar.exe  MDpar.exe         [.] potAccWork
     1.08%          1251  MDpar.exe  libgomp.so.1.0.0  [.] gomp_team_barrier_wait_end
     0.99%          1171  MDpar.exe  libgomp.so.1.0.0  [.] gomp_barrier_wait_end
     0.67%           780  MDpar.exe  libgomp.so.1.0.0  [.] gomp_mutex_lock_slow
     0.58%           767  MDpar.exe  [unknown]         [k] 0xffffffffab38c4ef
     0.15%           168  MDpar.exe  libgomp.so.1.0.0  [.] gomp_iter_dynamic_next
     0.06%            69  MDpar.exe  MDpar.exe         [.] VelocityVerlet
     0.01%            19  MDpar.exe  [unknown]         [k] 0xffffffffab396098
     0.01%             9  MDpar.exe  MDpar.exe         [.] MeanSquaredVelocity
     0.01%             7  MDpar.exe  MDpar.exe         [.] potAccWork
     0.01%             7  MDpar.exe  libgomp.so.1.0.0  [.] gomp_barrier_wait
```

Fig. 3. perf report

VIII. Compilador

Optámos também por introduzir algumas otimizações a nivel do compilador, visto que o parâmetro *cycles per instruction* está a aumentar, nomeadamente com a introdução da flag **-fpredictive-commoning** em ambas as versões sequencial e paralela de forma a que a medições do *speedup* fossem consistentes. A flag com a opção **-fpredictive-commoning** possibilita a otimização da reutilização de cálculos previamente realizados, como carregamentos ou leituras de memória e operações de escrita que ocorreram em iterações anteriores de *loops*. Como sabemos este tipo de operações em memória envolvem mais ciclos *clock* pelo que a sua utilização iria ajudar a diminuir o CPI. No entanto, a melhoria não foi muito significativa, sendo que apenas na execução com 20 *threads* se registou um decréscimo do CPI.

IX. Performance

Posteriormente, testámos a execução do programa com um número de *threads* compreendido entre 20 a 24, pois é o cenário em que o *speedup* do nosso programa é superior. Por fim, obtivemos uma maximização da *performance* com a execução de 21 *threads*, adquirindo um tempo médio de execução de aproximadamente 1.69 segundos com o comando sbatch.