Work Assignment - Phase 1

Parallel Computing, Master's Degree in Computer Engineering, University of Minho

Afonso Miguel Matos Bessa
MEI
University of Minho
pg53597

Francisco Luís Rodrigues Claudino
MEI
University of Minho
pg50380

***Abstract*—Este relatório documenta a nossa exploração do paralelismo de memória partilhada utilizando diretivas OpenMP para melhorar o tempo total de execução do código desenvolvido na Fase 1. A tarefa requer a otimização do tempo de computação ao identificar áreas críticas na aplicação, analisar e apresentar alternativas de paralelismo nessas áreas críticas, selecionar uma abordagem justificada com base numa análise de escalabilidade, implementar e otimizar esta abordagem no cluster SeARCH e, por fim, medir e discutir o desempenho da solução proposta. Este relatório detalha cada passo da metodologia utilizada para diminuir o Tempo de Execução através do paralelismo do código, fornecendo perceções sobre os desafios e sucessos encontrados durante este processo.**

FASE 2

A. Identificação de Hot-Spots

Primeiramente, definimos o número de partículas para **5000**.

Após isso, identificamos, com recurso ao **GProf**, *hot-spots*, ou seja, blocos de código cujo tempo de computação era elevado o suficiente para justificar o paralelismo do código. No nosso código, existia apenas uma função-`ComputeAccelarionsPotencial`-função essa que executava **99.57%** do código. Logo, a primeira conclusão que foi possível retirar face a esta análise foi a seguinte: apenas seria benéfico utilizar paralelização do código nesta função, pois o *overhead* resultante da criação, controlo e sincronia de fios de execução seria contraproducente e acabaria por aumentar o tempo de execução do programa.

time	seconds	seconds	calls	ms/call	ms/call	name
99.57	11.65	11.65	201	57.96	57.96	computeAccelerationsPotential()
0.69	11.73	0.08				VelocityVerlet(double, _IO_FILE*)
0.00	11.73	0.00	6480	0.00	0.00	gaussdist()
0.00	11.73	0.00	1	0.00	0.00	_GLOBAL__sub_I_N

Fig. 1: GProf Phase 1

B. Análise e Apresentação de alternativas para explorar o Paralelismo

Após uma análise detalhada da função `ComputeAccelarionsPotencial`, foi possível identificar três momentos onde paralelizar o código poderia ser benéfico, sendo estes os seguintes:

1) O ciclo que inicializa a *array* das acelerações a zero;
2) O ciclo que calcula a energia potencial de cada partícula;
3) O ciclo aninhado no ciclo anterior que calcula a força necessário para o cálculo da energia potencial.

Estas três secções de código são as que possuem maior vantagem de serem paralelizadas, uma vez que, todas partilham algo em comum, a execução de um ciclo de **N iterações**, ou seja, instruções que vão ser repetidas inúmeras vezes e que, em função disso, podem ser atribuídas a diferentes fios de execução, para, desta forma, diminuir o tempo de execução. Daqui surge a primeira diretiva passível de ser utilizada no âmbito do paralelismo: a `#pragma omp parallel for`. Esta última serve para inicializar múltiplos fios de execução ao longo de um ciclo.

Surge, contudo, um problema com a execução concorrente do mesmo bloco de código por diferentes fios de execução, as **data races**, que ocorrem quando dois ou mais fios de execução tentam escrever simultaneamente no mesmo conjunto de dados, o que compromete a correção de todo o algoritmo. As *data races* podem ocorrer em dois momentos no nosso programa:

1) Na escrita simultânea das mesmas variáveis locais;
2) Na escrita dos *arrays* globais do programa, que neste caso se resume apenas ao *array* das acelerações.

Daqui surge a segunda diretiva passível de ser aplicada e que corresponde à `#pragma omp private (var)`. Esta é utilizada para colmatar uma das causas das *data races* descritas anteriormente, fazendo com que todas as variáveis dentro desta diretiva sejam instanciadas para cada fio de execução, fazendo com que cada um destes fios possua a sua própria variável local, não correndo o risco de "corromper" as variáveis utilizadas pelos outros fios de execução com escritas incorretas. Da mesma forma, poderíamos utilizar a diretiva `#pragma omp critical` que cria uma secção crítica no código e que garante que apenas um fio de execução executa aquele bloco de código de cada vez, não correndo o risco de corrupção de dados. Da mesma forma, a diretiva `#pragma omp atomic` também dá garantias de que um bloco de código apenas é executado por um fio de execução, realizando as operações de maneira indivisível.

Relativamente às *data races*, é possível utilizar a diretiva `#pragma omp reduction (op:var)` para criar em cada fio de execução a sua cópia local de cada elemento do *array*, efetuando no final a operação matemática apenas uma vez com todas as variáveis locais dos fios de execução, resolvendo assim o problema das *data races*. Esta diretiva seria aplicada no ciclo que calcula a energia potencial, de modo a reduzir o número de operações matemáticas, permitindo que seja calculado um resultado agregado das variáveis de cada fio de execução, com referência mais concreta à variável **Pot** que representa a energia potencial de cada partícula e o ***array* de acelerações** utilizado no cálculo deste valor.

Focando agora a nossa atenção no ciclo que inicializa o *array* das acelerações a zero, é possível que as suas operações sejam vetorizadas "à mão", mas também pode ocorrer através da diretiva `#pragma omp simd`. Da mesma forma, no ciclo aninhado onde são calculadas as acelerações de cada partícula para auxilio ao cálculo da energia potencial, esta mesma diretiva pode ser utilizada para a vetorização destas operações.

Por fim, o último conjunto de diretivas que são passíveis de serem utilizadas baseiam-se no balanceamento da carga em cada fio de execução. De forma a potenciar o desempenho do programa e a minimizar o tempo de execução, pretende-se que todos os fios de execução executem o mesmo trabalho, ou seja, executem o mesmo

número de instruções. Isto não é tão trivial como parece, já que, neste caso em particular, possuímos um ciclo aninhado noutro ciclo onde o número de iterações do ciclo interior depende do número de iterações do ciclo exterior, aquando do calculo de variáveis auxiliares para o cálculo das acelerações das partículas. Para nos auxiliar neste balanceamento de carga, surge a diretiva `#pragma omp schedule(estratégia,chunk_size)` que define uma estratégia de escalonamento que determina como as iterações do ciclo serão divididas entre os vários fios de execução. Existem quatro tipos de estratégias utilizadas com esta diretiva, que são:

1) **static** - Divide o ciclo em pedaços de tamanho fixo especificado (*chunks*).
2) **dynamic** - Distribui os *chunks* de iterações dinamicamente entre as *threads*.
3) **guided** - Semelhante ao *dynamic*, mas os *chunks* são atribuídos aos fios de execução em tamanhos progressivamente menores.
4) **auto** - Deixa o compilador decidir o melhor método de agendamento.

C. Abordagens selecionadas para explorar o Paralelismo

Como os três blocos de código analisados consistiam em ciclos, a diretiva `#pragma omp parallel for` foi implementada pois sem ela, não seria possível utilizar fios de execução nestes ciclos. Após uma análise e comparação cuidadosa, optamos por utilizar as seguintes diretivas:

1) **#pragma omp simd** - Foi decidido utilizar pois consideramos fundamental a vetorização de instruções dentro de ambos os ciclos, sendo que o *overhead* que esta diretiva implica é compensado pela performance que os fios de execução oferecem.
2) **#pragma omp reduction** - Foi utilizado de modo a impedir as data races no acesso ao array global das acelerações e com o objetivo de reduzir as operações matemáticas de duas variáveis, devido aos cálculos de valores consideravelmente elevado e visto que o overhead é relativamente pequeno quando comparado ao tempo de execução ganho em cada iteração.
3) **#pragma omp scheduled (dynamic, 50)** - Optamos por usar esta diretiva em detrimento das outras três pois é aquela que balanceia de forma mais uniforme a carga computacional de cada fio de execução.

É importante notar que, não foi utilizada a diretiva `#pragma omp private` optando-se pela declaração das variáveis dentro dos ciclos. Isto permite que se resolva o problema de múltiplas escritas incorretas em variáveis partilhadas (*data races*), o que é mais eficiente em virtude da inexistência de qualquer *overhead* associado, o que não seria o caso no uso da diretiva. De realçar também que, caso não se optasse por esta resolução, esta teria sido a diretiva utilizada em detrimento das diretivas `#pragma omp critical` e `#pragma omp atomic`, uma vez que, estas duas tornam blocos do código acessíveis apenas a um fio de execução de cada vez, tornando esses blocos de código efetivamente sequenciais ao invés da diretiva `#pragma omp private` que apenas instanciaria uma variável por *thread* e que acabaria por compensar em termos de tempo de execução em *runtime*, mesmo que o *overhead* desta última seja ligeiramente superior devido ao alocamento de recursos para instanciação das variáveis.

Discussão de Desempenho e Análise de Escalabilidade

Após a implementação das diretivas que paralelizam o bloco de código considerado passível de paralelização, foram efetuadas medições do tempo de execução com vários fios de execução, nomeadamente 1, 2, 4, 8, 16, 20, 32 e 40. Estes números não foram escolhidos arbitrariamente, tendo-se optado por números de potências de base 2 devido a eficiência de memória cache bem como a facilidade de cálculos e divisões simétricas, bem como o número 20 (número de núcleos do *Cluster SeARCH*) e 40 (número máximo de fios de execução possível para o SeARCH, com dois fios de execução por núcleo). Na tabela que se segue podemos observar o Tempo de execução que obtivemos por fio de execução.

Threads	Real Time (s)
1	29.195
2	14.631
4	7.366
8	3.749
16	1.957
20	1.594
32	1.660
40	1.668

Tempos de Execução para diferentes número de Threads

Após efetuadas estas medições, foi calculado o **SpeedUp** para cada número de *threads* utilizadas, através do uso da fórmula que se segue.

$$SpeedUp = \frac{T_{\text{exec Sequencial}}}{T_{\text{exec Paralelo}}} \tag{1}$$

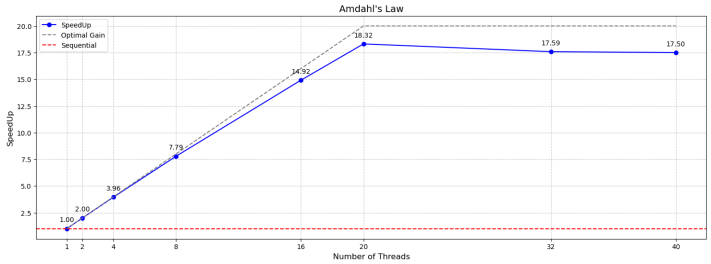

Fig. 2: SpeedUp Graph

No gráfico de SpeedUp da figura 2, podemos observar duas linhas distintas. A linha a tracejado expõe o ganho ideal que se expectava com o paralelismo do código enquanto que a linha contínua apresenta o ganho real baseado nas medições e testes ao nosso programa. Esta linha de *SpeedUp* ideal foi calculada com base na **Lei de Amdahl**, que obtém o valor do ganho ideal com base no número de núcleos utilizados e na percentagem de tempo de código despendido que será paralelizado (fórmula abaixo). Dado que a percentagem de código despendido paralelizado é muito próxima de 100% (cerca de 99,57%), o ganho ideal torna-se diretamente proporcional ao número físico de núcleos da máquina.

$$SpeedUpIdeal = \frac{1}{(F_p + \frac{1-F_p}{N})} \tag{2}$$

Na análise dos resultados efetivos, verifica-se que o ganho é contínuo até que o número de fios de execução alcance o número de núcleos do SEARCH (20 - *threshold*), diminuindo ligeiramente e, eventualmente, estabilizando com o aumento do número de fios de execução. Este facto leva-nos a concluir que não é muito vantajoso a utilização de um número de fios de execução superior ao número de núcleos do SeARCH, pois cada núcleo passaria a lançar duas *threads* que competiriam por recursos computacionais e memória, diminuindo portanto o SpeedUp do programa. Também podemos concluir que o *hardware* onde executamos o programa não nos limita em termos de performance, pois o ganho mantêm-se constante com o aumento do número de fios de execução a partir do *threshold* previamente mencionado.

De realçar também que os valores reais se aproximam bastante do SpeedUp ideal, o que nos leva a concluir que o código paralelizado possui uma forte escalabilidade.