Computação Paralela 23/24
Fase 1

João Castro - pg53929
Vasco Rito - a98728

***Key Words*—ILP improvements, memory hierarchy, data structures organization, vectorisation**

I. Introdução

Para a primeira fase do projeto de Computação Paralela foi-nos proposto fazer uma análise, utilizando ferramentas de profiling, de forma a optimizar o código fornecido.

II. Profiling

Análise do call-graph inicial

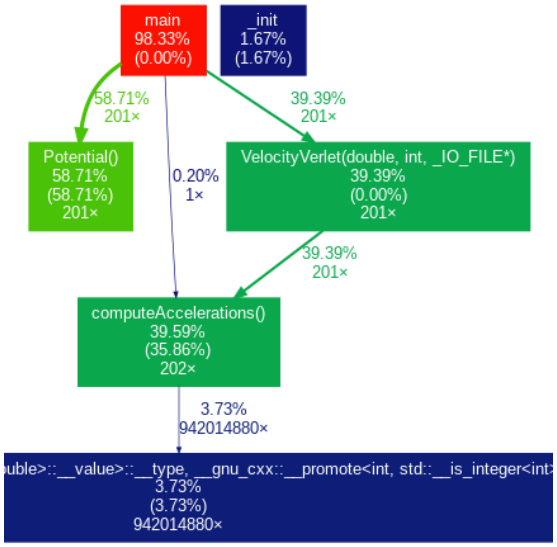

Fig. 1. Call-Graph Inicial

Depois de analizar o call-graph inicial, pode aferir-se que cerca de 59% do tempo é gasto na função Potencial() e, por isso, esta deve ser a função a melhorar. Pode ver-se também que cerca de 39% da VelocityVerlet() é feito na ComputeAccelerations(), sendo que 4% são invocações de bibliotecas.

III. Estimativa da performance de execução

Procedeu-se a uma estimativa dos tempos de complexidade de cada um dos métodos juntamente com as respetivas estimativas do número de acessos à memória de cada.

Functions	Estimated Time Complexity	
	Overall Time	***Memory Accesses***
initialize()	N^3	$3 * N^3$
MeanSquaredVelocity()	N	$6N$
Kinetic()	$3N$	$6N$
initializeVelocities()	$15N$	$24N$
VelocityVerlet()	$9N + T(CA)$	$39N + MA(CA)$
ComputeAcelerations()	$3N + ((N-1) * N)/2$	$3 * N^3$
Potential()	N^2	$N * (N-1) * 12$

* CA-ComputeAcelerations* MA - memory accesses

IV. Otimizações

A. Lógica do código

O método Potential é constituído por dois loops aninhados, sendo que o cálculo do r2 é feito quando i é diferente de j. O r2 é formado por uma multiplicação de diferenças, logo, irá haver sempre duas iterações em que o valor de r2 será o mesmo, pois, (r[i][k]-r[j][k])*(r[i][k]-r[j][k]) = = (r[j][k]-r[i][k])*(r[j][k]-r[i][k]). Ou seja, o resultado da subtração é simétrica mas como é elevada ao quadrado o resultado é igual. Assim, podemos reduzir o número de iterações no loop de j, começando este em i+1 (pois j tem de ser diferente de i). Para o cálculo do Potencial se manter consistente no final procede-se à sua duplicação.
Após uma análise aos método Potential e ComputeAccelerations, verificou-se que a estruturação do código era semelhante, iterando pela matriz de posições com dois loops aninhados em ambos, pelo que se criou um método que realiza esse trabalho,de modo a reduzir o número de instruções e a redundância. Como o outter loop no ComputeAccelerations tem N-1 iterações, o novo método recebe como argumento o nº de iterações total, já que no Potential é N.
De seguida, verificámos que era possível reduzir o número de invocações de métodos como o Kinetic() e o ComputeAccelerations(). O output do Kinetic é possível obter a partir do cálculo antecedente do MeanSquaredVelocity, uma vez que estes métodos têm um comportamento semelhante (somam os quadrados dos componentes da velocidade), porém o Kinetic no final multiplica a soma por m/2 enquanto que o MeanSquaredVelocity no final faz a divisao da soma por N. Logo, a relação destas é **Kinetic() / const2m == MeanSquaredVelocity() * N**, pelo que podemos escrever a Kinetic em função da MeanSquaredVelocity da seguinte forma (e visto que a multiplicação tem menos ciclos clock que uma divisão): **Kinetic() == MeanSquaredVelocity() * N * const2m**.
Por último, podemos reparar que o método potAccWork é

invocado mais vezes do que o necessário já que este é invocado quando o VelocityVerlet é executado e quando o Potential é invocado para obter a energia potencial. Como no nosso método potAccWork é possível obter o valor do potential mesmo quando o argumento passado é N-1, podemos obter esse valor no VelocityVerlet. Logo, **mvs = MeanSquaredVelocity()**; **KE = mvs * N * const2m**; **PE = potential**, diminuindo assim o nº de clock cycles;

B. Simplificação do cálculo de fórmulas e do modelo do potential de Lennard-Jones

No cálculo do potential temos que rnorm=sqrt(r2) e que quot=sigma/rnorm, no entanto como quot é elevado a números pares(6 e 12) não é necessário fazer o sqrt, passando os expoentes para metade. Assim, como sigma é sempre 1, **term2= sigma/ pow(r2,6)** e **term1 = term2*term2**(para poupar ciclos clock com calculos desnecessários). Por fim, retirou-se a invocação do método pow, pois esse envolve cálculos intermédios e iterações com expoentes decimais que se traduzem num CPI maior comparativamente com sucessivas multiplicações que é uma operação básica que geralmente é realizada em 1 ciclo clock. Assim, **Pot += term1 - term2**. Por fim, (para não gerar overhead de multiplicações dentro do loop) Pot é multiplicado por 4 * epsilon e por 2 como foi explicado no capítulo A. Na fórmula f, tivemos que simplificar o cálculo colocando tudo ao mesmo denominador de modo a retirar o expoente negativo: **f = 24 * (2 - pow(r2,3)) / pow(r2,7))**. Depois, podemos substituir os pow por multiplicações. As restantes simplificações foram essencialmente retirar multiplicações dentro de loops para não gerar overhead de operações.

C. Hierarquia de memória

De forma a explorar a hierarquia de memória é necessário garantir que no que toca a operações entre matrizes a localidade espacial é obedecida (travessias row-order) e que os dados utilizados no loop caberão na cache até que sejam reutilizados pois temos que ter em conta que as linhas da cache vão sendo substituídas, o que implica cache misses e novas leituras da memória(se for miss em L3) e que podem ter algum custo. Portanto, a estratégia utilizada foi **Loop Block Optimization**. Assim, restruturámos o código do potAccWork do seguinte modo:

```
1    for (int j = 0; j < N; j += blockSize) {
2        for (int i = 0; i < fun; i += blockSize) {
3            for (int jb = j; jb < j + blockSize &&
     jb < N; jb++) {
4                for (int ib = i; ib < i + blockSize
     && ib < fun && ib < jb; ib++) {
```

O blockSize corresponde ao tamanho da cache L1, que no cluster possui 32K (comando lscpu). Existem 4 loops aninhados sendo que o jb e o ib são necessários para fazer a travessia de submatrizes e explorar a localidade espacial da cache. Para além disso, adicionámos a condição $ib < jb$ de forma a diminuir para metade o nº de iterações, já que os valores das subtrações seriam o simétrico mas como são elevados ao quadrado o output é o mesmo como já referímos no capítulo de Otimizações-A.

D. A nível do compilador

A máquina de teste para obtenção dos resultados com as diferentes *flags* tem as seguintes características:

- Model name: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
- CPU family: 6
- CPU max MHz: 4100,0000
- Caches L1d: 192 KiB
- Caches L2: 1,5 MiB
- Caches L3: 9 MiB

Flags	Estimated Time Complexity	
	Elapsed Time (seconds)	*CPI*
-O0	14, 27	0, 41
-O2	3, 44	0, 35
-O3	3, 32	0, 47
-O2 -funroll-all-loops	3, 18	0, 44
-O3 -funroll-all-loops	3, 24	0, 46
-O2 -funroll-all-loops -ftree-vectorize -msse4	2, 88	0, 38
-O3 -funroll-all-loops -ftree-vectorize -msse4	3, 21	0, 46
-O2 -funroll-all-loops -ftree-vectorize -mavx	2, 72	0, 46
-O3 -funroll-all-loops -ftree-vectorize -mavx	3, 14	0, 54
-Ofast	3, 23	0, 46
-Ofast -funroll-all-loops	3, 32	0, 47
-Ofast -funroll-all-loops -ftree-vectorize -msse4	3, 29	0, 47
-Ofast -funroll-all-loops -ftree-vectorize -mavx	3, 12	0, 54

Como podemos observar na flag O3 temos tempos superiores a O2 furtuito de possíveis dependências de dados que se traduzem em CPI's superiores.

V. Profiling

Análise do call-graph final

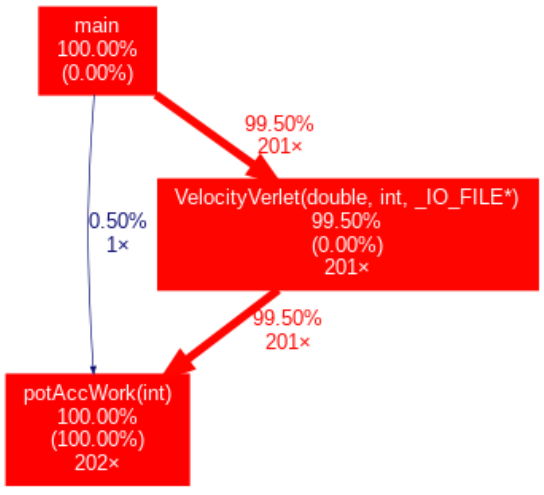

Fig. 2. Call-Graph Final

Depois de analizar o call-graph final, vê-se que a função potAccWork() é a que gasta a maior parte do tempo, visto que todo o trabalho da VelocityVerlet() é realizado pela potAccWork().