# Computação Paralela
# 3dfluid - 1º Fase


Rúben Silva
*pg57900*
*Mestrado Eng. Informática*

Pedro Oliveira
*pg55093*
*Mestrado Eng. Informática*

Henrique Faria
*a91637*
*Externo*



***Abstract*—Este artigo tem o objetivo de analisar um software de simulação de fluidos sem optimizações, fundamentado no projeto jgbarbosa/3dfluid. O código original, que representa interações entre partículas fluidícas com *runtimes* muito elevados.**


## I. Introdução

Nesta fase do Trabalho Prático de Computação Paralela, examinamos um software de simulação de partículas fluidas, empregando técnicas de profiling para aprimorar o código. O código original, executado em single-thread, enfrentava problemas de desempenho devido a ausência de optimização.

O objetivo foi implementar técnicas de optimização fundamentadas no *Paralelismo no Nível de Instrução* (ILP) e na *Hierarquia de Memória*. A avaliação concentrou-se em parâmetros cruciais, tais como tempo de execução, contagem de ciclos/instruções, *CPI* e *L1-dCache-Load-Misses*.

### A. Profiling

Com o ***gprof2dot***, foi retirado um *snapshot* do *Call-Graph* do código original (*Fig. 1*), onde foi obtido os seguintes dados:
(Por falta de legibilidade dos *Grafos*, as funções com mais custosas estão descritas nas Tabelas abaixo dos respetivos diagramas)

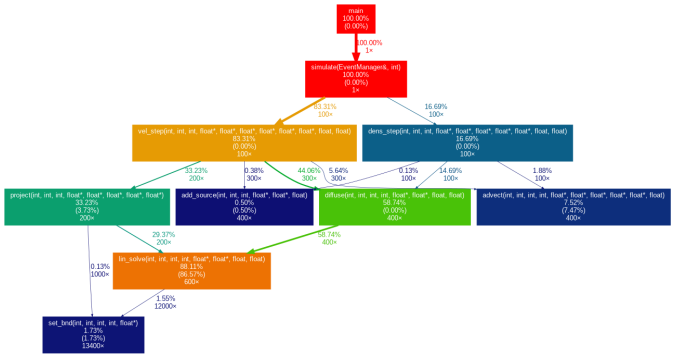


Fig. 1: Call-Graph Inicial

TABLE I: Call-Graph Inicial

| Nome da Função | Utilização Relativa | Nº Chamadas |
|---|---|---|
| lin_solve | 86.57% | 600 |
| advect | 7.47% | 400 |
| project | 3.73% | 200 |
| set_bnd | 1.73% | 13400 |
| add_source | 0.50% | 400 |

Após a implementação de todas as optimizações foi obtido o seguinte *Call-Graph*:

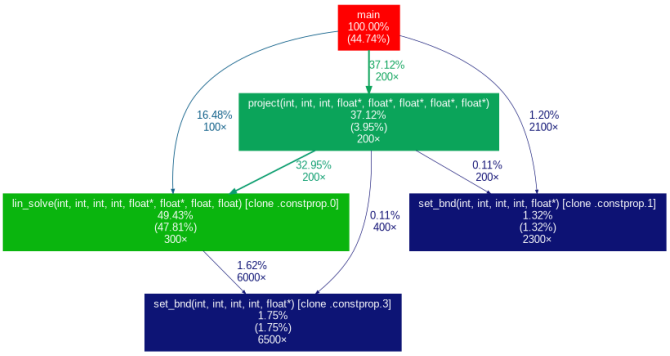


Fig. 2: Call-Graph Final

TABLE II: Call-Graph Final

| Função | Utilização Relativa | Nº Chamadas |
|---|---|---|
| lin_solve | 47.81% | 300 |
| main | 44.74% | —— |
| project | 3.95% | 200 |
| set_bnd | 3.07% | 8800 |

## II. Estimativa de Performace

Foi realizada uma estimativa da complexidade de cada uma das funções, que podem ser vistas nas seguinte tabela

TABLE III: Estimativa de Performance

| Função | Complexidade Temporal |
|---|---|
| lin_solve | O(LINEARSOLVERTIMES×M×N×O) |
| advect | O(M×N×O) |
| project | O(M×N×O) |
| set_bnd | O(M×N+N×O+M×O) |
| add_source | O(M×N×O) |

## III. Optimizações

### A. Hierarquia de Memória

A *Hierarquia de Memória* afeta diretamente a performance do código, devido às velocidades distintas entre o Cache e a Memória *RAM*. A técnica de **Loop Tiling** segmenta os ciclos em partes menores, possibilitando ao máximo o reaproveitamento de dados na Cache. Isso diminui os acessos lentos à memória e aprimora a localização dos dados, levando a um aumento na eficiência e performance do código.

Para determinar o melhor tamanho de bloco (Block Size), foi necessário considerar que a cache L1 do processador do Cluster da Universidade possuía 32KB. Inicialmente, foi utilizado um valor de *bloco* (diferente em casa função) por ciclo de execução (for). Após testar vários valores, os seguintes foram os que demonstraram maior eficiência.:

- *lin_solve* $\rightarrow$ **TILE = 6**
- *advect* $\rightarrow$ **TILE = 26**
- $project_{for1} \rightarrow$ ***TILE = 16***
- $project_{for2} \rightarrow$ ***TILE = 16***

Os valores apresentados são relativamente pequenos, pois o simulador utiliza 3 loops for aninhados, o que implica que o valor final é um valor cúbico ($\mathbf{TILE}^3$), logo, facilmente exponencializa.

Para conseguirmos melhorar mais a performance do simulador, também foi trocada a ordem dos **Loops**. Inicialmente era seguida a ordem "*M-N-O*", como "*M*" é a grandeza que está ligada diretamente ao vizinhos mais proximas na memória, é fundamental este ter de ser o primeiro a ser "percorrido" para maximizar o reaproveitamente de valores presentes na cache. Com isto chegamos a duas potenciais ordens para os Loops, "*N-O-M*" ou "*O-N-M*". Por tentativa e erro concluimos que a ordem "*N-O-M*" seria que a trouxe mais vantagens ao código ganhando algo em torno de 10% de mais "*run-time*"

As siglas representam as asbtrações das dimensões do simulador compondo assim uma Matrix de *M x N x O*, sendo crucial uma selação critica da ordem na qual se irá percorrer-la.

### *B. Paralelismo no Nível de Instrução*

O *Paralelismo no Nível de Instrução* é uma técnica crucial para melhorar o desempenho do código, aproveitando ao máximo a capacidade e a arquitetura do processador. Para otimizar a execução do código, utilizámos diversas flags do GCC, que permitem ao compilador reordenar e paralelizar instruções de forma eficaz.

As flags usadas foram:

- **-O3**: Ativa otimizações agressivas, como a eliminação de código redundante e a *inlining* de funções.
- **-Ofast**:Além das otimizações do nível -O3, ativa ainda outras otimizações que podem violar padrões de precisão em cálculos, resultando em um desempenho ainda melhor, mas com risco de perda de precisão.
- **-funroll-all-loops**: Desenrola todos os loops, o que significa que o compilador expande o código do loop para reduzir o overhead de controle e aumentar a execução paralela das instruções.
- **-ftree-vectorize**: Habilita a vetorização automática, permitindo que o compilador use instruções *SIMD* para processar múltiplos elementos de dados em uma única operação, o que melhora a eficiência no processamento de arrays e matrizes.
- **-mavx**: Ativa o uso de instruções *AVX* (Advanced Vector Extensions), que permitem a execução de operações vetoriais em 256 bits, proporcionando um aumento significativo no desempenho em cálculos intensivos.
- **-march=native**: Instrui o compilador a optimizar o código para a arquitetura específica do processador onde está a ser compilado, utilizando instruções e recursos específicos de uma forma mais optimizada ao sistema.
- **-flto**: Link Time Optimization, permite ao compilador optimizar o código após a fase de compilação, resultando em um executável mais eficiente em termos de performance e tamanho.

As flags mencionadas anteriormente não foram necessariamente aplicadas em simultâneo. Foram realizadas diversas combinações distintas, a fim de identificar as configurações que proporcionassem os resultados mais relevantes.

## IV. Resultados

Nesta secção, é descrito os resultados no processador do Cluster da Universidade do Minho descrito com as informações relevantes para o estudo, na Tabela *IV*

TABLE IV: Características do Processador do Cluster

| **Características** | **Cluster UMinho** |
|---|---|
| Marca | Intel |
| Modelo | Xeon E420 |
| Frequência | 2.5 GHz |
| Cache L1 | 32 KB |
| Cache L2 | 6144 KB |

Para um melhor entedimento de como ambas as optimizações, as tabelas abaixo mostram os resultados obtidos com as várias *flags* do GCC com e sem Hierarquia de Memória.

Para facilitir a legibilidade da próxima tabela, será usada as seguintes abreviaturas:

- **FRLoop** $\rightarrow$ funroll-all-loops
- **FtrVect** $\rightarrow$ ftree-vectorize
- **L1dcLM** $\rightarrow$ L1-dcache-load-misses
- **MNative** $\rightarrow$ march=native

TABLE V: Resultados Sem Hierarquia de Memória

| Métricas | No Flags | `-O2 -FRLoop -FtrVect -mavx` | `-O3 -FRLoop -FtrVect -mavx` | `-Ofast -FRLoop -MNative -flto` |
|---|---|---|---|---|
| Instructions | 166613827657 | 16291008681 | 15645802313 | 16671358346 |
| Cycles | 94417810364 | 43400774402 | 34525319160 | 22778985010 |
| L1dcLM | 2420130899 | 2846690855 | 2594925840 | 2530349767 |
| Cache-Misses | 1026 | 1273 | 6025 | 1276 |
| Cache-Reference | 1336375896 | 2420720487 | 1287953269 | 1205093828 |
| **CPI** | 0,567 | 2,664 | 2,207 | 1,366 |
| **Run-Time (ms)** | 31.2821 | 13.5709 | 10.7519 | 7.072 |
| **Result** | 81981.3 | 81981.3 | 81981.3 | 81981.5 |

TABLE VI: Resultados Com Hierarquia de Memória

| Métricas | No Flags | `-O2 -FRLoop -FtrVect -mavx` | `-O3 -FRLoop -FtrVect -mavx` | `-Ofast -FRLoop -MNative -flto` |
|---|---|---|---|---|
| Instructions | 168117489688 | 19328951330 | 17286351975 | 18894376660 |
| Cycles | 63205633692 | 21383833040 | 20042667776 | 12453730302 |
| L1dcLM | 243535495 | 244202794 | 243172253 | 245089457 |
| Cache-Misses | 737 | 859 | 3041 | 724 |
| Cache-Reference | 97980725 | 100185792 | 98714473 | 98948982 |
| **CPI** | 0.376 | 1,106 | 1.159 | 0,659 |
| **Run-Time (ms)** | 19.6830 | 6.5753 | 6.09 | 3.62 |
| **Result** | 81981.2 | 81981.2 | 81981.2 | 81981.5 |

Dado os dados das tabelas anteriores, é observado uma melhoria de aproximadamente **89%** do run-time da simulação (**31.3s** $\rightarrow$ **3.62s**) com $\approx$ **3.4%** *L1-dcache-misses*.

Durante o desenvolvimento desta fase, foi notado que algumas implementações da Hierarquia de Memória, como a troca da ordem dos "*for*" pioravam os tempos das combinações de flags mais simples e melhorava bastante as de flags mais agressivas.

## V. Conclusão

Neste trabalho, exploramos diversas técnicas de otimização de desempenho em um simulador de fluidos baseado no projeto jgbarbosa/3dfluid. Através do uso de técnicas como **Loop Tilling**, foi espectado uma melhoria *spatial locality* (Hierarquia de Memória) e através dos recursos dos compiladores modernos, foi pretendido explorar/melhorar o Paralelismo no Nível de Instrução (ILP).

Em suma, os resultados obtidos comprovam a eficácia das técnicas de otimização aplicadas, destacando a relevância do *profiling* detalhado e de ajustes no código para obter melhorias substanciais em termos de desempenho computacional.