Faculdade de Engenharia da Universidade do Porto

1.º trabalho prático

Programação paralela no contexto de multiplicação de matrizes

Computação Paralela e Distribuída (L.EIC028) 2024/2025
Licenciatura em Engenharia Informática e Computação

Carlos Miguel Ferraz Baquero-Moreno	(Co-regente da Unidade Curricular)
João Miguel Maia Soares de Resende	(Co-regente da Unidade Curricular)
Pedro Alexandre Ferreira Souto	(Co-regente da Unidade Curricular)
Bruno Miguel Almeida Cunha	(Professor das aulas laboratoriais)

Turma 9, Grupo 13:

Duarte Souto Assunção	up202208319@up.pt
Guilherme Duarte Silva Matos	up202208755@up.pt
João Vítor da Costa Ferreira	up202208393@up.pt

Índice

1. Descrição do problema

A utilização eficiente dos recursos computacionais é fundamental na computação de alto desempenho. A multiplicação de matrizes é um bom exemplo de uma operação computacionalmente intensiva e altamente dependente da hierarquia da memória e da eficiência do processador.

Este projeto tem como objetivo analisar o impacto dos padrões de acesso à memória no desempenho do processador durante a execução de operações de multiplicação de matrizes de grande escala. Além disso, serão avaliados os benefícios da computação paralela implementando otimizações *multi-core*.

2. Explicação dos algoritmos

Tendo que a matriz **A** de dimensão **m** colunas por **n** linhas multiplicada pela matriz **B** de **n** colunas e **p** linhas resulta na matriz **C** de dimensão **m** colunas por **p** linhas, o elemento na coluna **i** e linha **j** da matriz **C** será o produto escalar entre a coluna **i** da matriz **A** com a linha **j** da matriz **B**, ou seja:

$$C_{j,i} = \sum_{k=1}^{p} (A_{k,i} \times B_{j,k})$$

2.1. Multiplicação Sequencial Simples

A multiplicação simples é a implementação direta do algoritmo de multiplicação de matrizes. Assim, percorre-se cada posição da matriz resultante e calcula-se o seu valor.

2.2. Multiplicação Sequencial em Linha

A multiplicação em linha é uma multiplicação matricial feita por uma ordem diferente. Em vez de computar cada produto escalar imediatamente, calcula-se linha a linha. O elemento na coluna **i** e linha **j** da matriz **A** irá multiplicar por todos os elementos da linha **j** da matriz **B**. Isto resultará em **n** elementos que serão acionados às posições correspondentes da linha **j** da matriz **C**.

2.3. Multiplicação Sequencial por Blocos

Este algoritmo consiste em dividir a matriz **C** em blocos de tamanhos iguais e, para cada um desses blocos, computar o resultado da multiplicação matricial individualmente.

2.4. Multiplicação Paralela Simples

Usando a mesma estratégia da multiplicação simples, computar a multiplicação matricial dividindo os cálculos por várias threads.

2.5. Multiplicação Paralela em Linha

Usando a mesma estratégia da multiplicação em linha, computar a multiplicação matricial dividindo os cálculos por várias threads, e com dupla paralelização.

3. Métricas de desempenho

A fim de avaliar o desempenho destes algoritmos, foram recolhidas estas métricas:

- **Tempo de execução** do algoritmo, que não inclui a inicialização das matrizes, a partir do `omp_get_wtime()`;
- **FLOPS**, a partir do contador `PAPI_DP_OPS`;
- **Quantidade** de ***cache misses*** **de nível 1**, a partir do contador `PAPI_L1_DCM`, **e de nível 2**, a partir do contador `PAPI_L2_DCM`.
- **Quantidade** de ***cache accesses*** **de nível 2**, a partir do contador `PAPI_L2_DCA`.

No código de Rust, é usado a biblioteca *"perf_event2"* em vez de *"papi_bindings"* para analisar os *cache L1 misses* e outros eventos suplementares.

4. Resultados e análise

Os algoritmos foram implementados em C/C++, bem como em Rust para os algoritmos de multiplicação sequenciais simples e em linha. Estes foram avaliados nos computadores da faculdade com um processador Intel Core i7-9700 com 8 cores de 3GHz sem *"Hyper-Threading"*. Os resultados destas avaliações estão disponíveis nos apêndices.

4.1. Algoritmo simples e em linha

O algoritmo em linha demonstra ser mais eficiente que o algoritmo simples, diminuindo o tempo de execução em até 90% nas matrizes de maior dimensão. (Fig. 1)

Isto é devido à localidade dos dados, já que o processador assume que ao buscar um elemento à memória os elementos seguintes também serão necessários, portanto, na *cache* serão armazenados esses elementos, que só o algoritmo em linha imediatamente os usa. Tal é comprovado pelos baixos *cache misses* nos níveis L1 e L2 do algoritmo em linha, em comparação com o algoritmo simples. (Fig. 2)

Em Rust o algoritmo em linha é ligeiramente mais rápido do que em C++, e o algoritmo simples é aproximadamente idêntico. Feito um *disassembly,* foi possível observar algumas instruções SIMD.

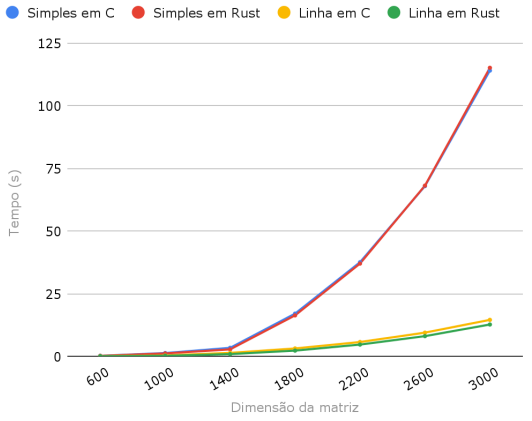

Figura 1. Tempo de execução em algoritmos simples e em linha nas duas linguagens

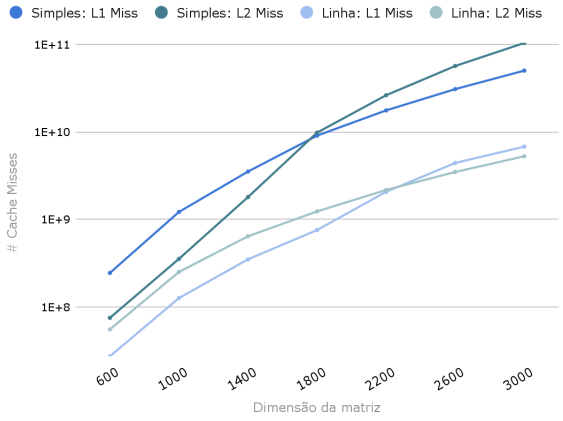

Figura 2. Cache L1 e L2 misses e nos algoritmos simples e em linha

4.2. Multiplicação em blocos

O tamanho dos blocos não afetou significativamente o tempo de execução do algoritmo em matrizes pequenas (Fig. 3), porém o número de *cache L2 misses* variou bastante entre matrizes. Em matrizes pequenas (entre 600 a 3000 elementos), um bloco de tamanho baixo (por exemplo, 128) é preferido para evitar *cache misses*. Isto deve-se ao limite de 256KB de *L2 Cache* deste processador, que só consegue armazenar até aproximadamente uma matriz de 178 elementos. Porém, em matrizes de maior dimensão, o oposto se aplica: 512 elementos num bloco é preferível. (Fig. 4)

O tempo de execução deste algoritmo foi muito eficaz para matrizes grandes, com uma redução de até 48% comparado com o algoritmo de linha.

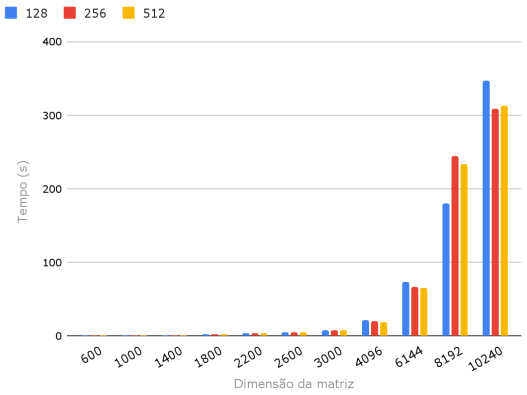

Figura 3. Tempo de execução do algoritmo em blocos de 128, 256 e 512

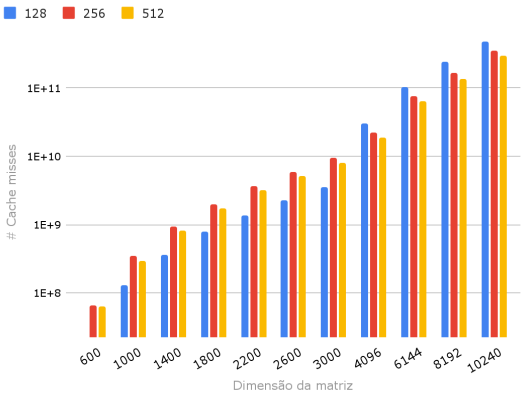

Figura 4. Cache L2 misses do algoritmo em blocos de 128, 256 e 512

4.3. Benefícios do paralelismo: eficiência e *speedup*

Em comparação com os seus algoritmos sequenciais, os algoritmos simples e em linha paralelizados são consideravelmente mais rápidos, com um *speedup* de, no mínimo, superior a 3. (Fig. 6)

No entanto, apenas o algoritmo em linha é razoável para matrizes de grande tamanho, já que as falhas de *cache* evidenciadas no algoritmo simples se propagam para os oito núcleos, o que torna a execução de uma multiplicação de dimensão 10240 de uma grandeza de minutos para uma grandeza de horas. (Fig. 5)

Mesmo assim, é de notar que para matrizes pequenas, o *speedup* do algoritmo simples foi próximo de 8, devido ao uso de uma variável temporária, evitando, desta forma, memória partilhada entre threads, o que obrigaria a uma zona crítica.

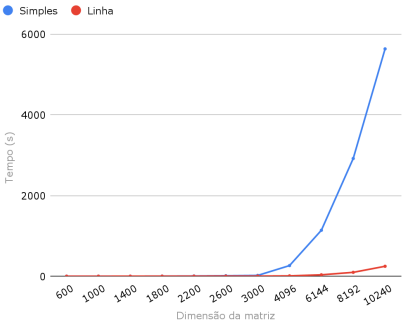

Figura 5. Tempo de execução em algoritmos paralelos

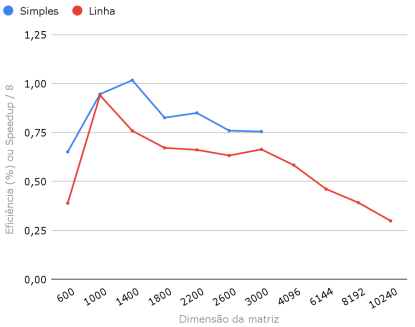

Figura 6. Eficiência/Speedup em algoritmos paralelos

4.4. Análise de MFLOPS

Todos os algoritmos sequenciais implementados em C/C++ executam o mesmo número de operações quando comparados com o mesmo tamanho de matriz. No entanto, algoritmos paralelos executam cerca de 5 vezes mais operações. Assim sendo, os MFLOPS de um algoritmo não influenciarão apenas no seu tempo de execução.

Por consequência, o algoritmo sequencial de blocos executa mais operações por segundo do que os algoritmos paralelos, sendo um bom candidato para um algoritmo genérico. Contudo, vale a pena mencionar o impressionante desempenho do algoritmo em linha desenvolvido em Rust quando executado com matrizes pequenas.

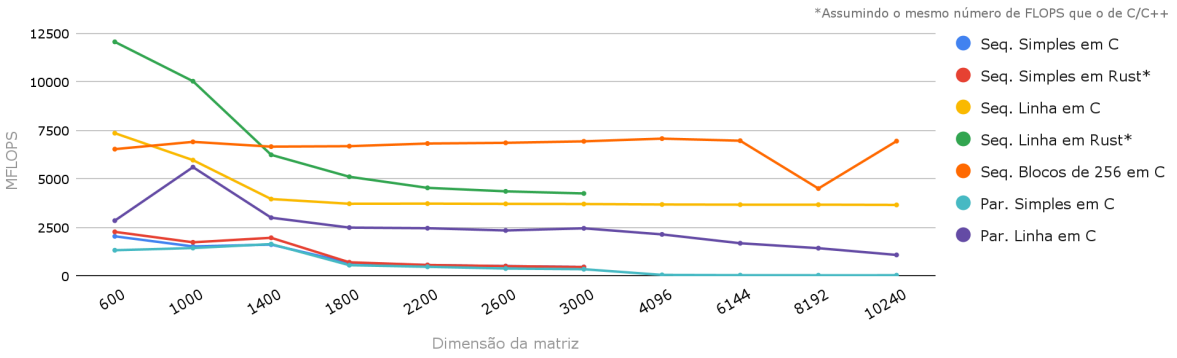

Figura 7. MFLOPS de todos os algoritmos testados

4.5. Outras observações

- O número de *cache L2 misses* é maior do que o número de *cache L1 misses*, para quase todos os tamanhos de matrizes. Isto poderá ser porque, quando há uma *cache miss* nos dois níveis, apenas é contabilizado o último, ou seja, se houver *cache miss* no L1 e no L2, apenas é contabilizada a do L2.
- Os *cache L2 acesses* também foram recolhidos durante a avaliação dos algoritmos, no entanto, este contador é muitas vezes menor que o número de *cache L2 misses*, principalmente em grandes matrizes. Tal leva a crer que, ou o PAPI não está a fazer uma medição correta destes valores, ou, visto que há *counters* diferentes de leitura e de escrita na *cache*, o CPU possa não contabilizar algum deles quando há acessos à *cache*. É também possível que possam existir *misses* no TLB *(translation lookaside buffer)*, levando a acessos diretos a um nível superior de memória (L3, RAM, etc.). Uma outra explicação seria a diferença de atualização dos contadores dependendo da instrução (na *pipeline*, o estágio, bem como se é ativado em *rising-edge* ou *falling-edge*) e de *out-of-order, speculative execution*.
- Durante o desenvolvimento deste projeto, uma versão anterior do código de C/C++, que usava funções da classe `Matrix` (sem diretiva `inline`) em vez dos operadores primitivos de acesso a *arrays* (com diretiva `inline`), tinha um tempo de execução muito aquém do esperado. Isto deve-se ao compilador não otimizar essas chamadas de funções e a memória (e consequentemente a *cache*) ficarem "poluídas" com informações de funções.

5. Conclusões

A realização destes testes de desempenho em vários algoritmos, e consequentemente, este projeto como um todo, permitiu consolidar que:

- Mudanças súbitas no código (principalmente em C/C++) podem ter um grande impacto no desempenho do programa, nomeadamente variáveis e funções auxiliares;
- A localidade dos dados e a análise do código na perspetiva da *cache* é essencial para construir algoritmos mais eficientes;
- As diretivas `#pragma` do OpenMP facilitam o desenvolvimento de aplicações paralelas, no entanto, mudanças que parecem subtis podem ter um impacto enorme.

Conclui-se que, tal como esperado, otimizar e paralelizar um algoritmo é importante para aproveitar o máximo dos recursos disponíveis de uma forma mais económica, mas tal deve ser implementado cuidadosamente.

Apêndices

As folhas de cálculo com todas as avaliações de desempenho está disponível em [CPD] Recolha de dados.

Algoritmo sequencial e simples em C/C++

Matrix Dimensions	Time (s)	L1 Cache Misses	L2 Cache Misses	L2 Cache Acesses	FP Operations
600	0,21067	243874457	74664171	217216773	432000004
1000	1,30645	1215619762	353017140	1094915933	2000000004
1400	3,37975	3522441834	1797968976	3179604871	5488000004
1800	17,0255	9072015413	9824000722	8343803974	11664000004
2200	37,5326	17644375276	26300004399	16291053439	21296000004
2600	67,9138	30909279890	56800791281	28587987359	35152000004
3000	114,006	50301946711	104123182586	46494384817	54000000004

Algoritmo sequencial e simples em Rust

Matrix Dimensions	Time (s)	L1 Access	L1 Miss	LL Access	LL Miss
600	0,189441676	432000229	244178043	68769962	1011798
1000	1,147994482	2000000641	1220537319	436550638	9637593
1400	2,779583816	5488001187	3492810111	1731397711	208285029
1800	16,33181287	11664004358	9077871393	8227960087	857367449
2200	36,99760237	21296008512	17628660731	25791289350	1781836832
2600	68,09348085	35152014978	30872107857	56025370706	3133988226
3000	115,1072562	54000025060	50285427690	103375593116	4907871233

Algoritmo sequencial em linha em C

Matrix Dimensions	Time (s)	L1 Cache Misses	L2 Cache Misses	L2 Cache Acesses	FP Operations
600	0,0587052	27104852	55095924	8153480	432000004
1000	0,334942	125725154	250158089	37974003	2000000004
1400	1,38296	348826783	640417284	75794924	5488000004
1800	3,13059	754824640	1232464081	99673374	11664000004
2200	5,70567	2065527043	2170815298	296662993	21296000004
2600	9,44665	4416611112	3481978714	582912103	35152000004
3000	14,5464	6789405354	5266943273	822480532	54000000004
4096	37,265	17313649319	13223629839	2079578524	137438953476
6144	126,107	58362709686	44299539117	6495753721	463856467972
8192	298,883	138203596502	104840789196	14668898123	1099511627780
10240	585,88	269798595969	207190120584	34740166197	2147483648004

Algoritmo sequencial em linha em Rust

Matrix Dimensions	Time (s)	L1 Access	L1 Miss	LL Access	LL Miss
600	0,035829283	108360206	27087129	50416861	158211
1000	0,199369086	501000471	125648512	232672345	5836169
1400	0,878908591	1373960977	345667566	621312328	129077292
1800	2,280349745	2919241277	742494708	1302208337	437406218
2200	4,686242171	5328842196	1918491634	2323526611	997988323
2600	8,053230858	8794763570	4413105285	3641211008	1788620834
3000	12,68985358	13509005096	6779832197	5532535478	2892630824

Algoritmo sequencial por blocos

Matrix Dimensions	Block size	Time	L1 Cache Misses	L2 Cache Misses	L2 Cache Acesses	FP Operations
600	128	0,06324	30701009	22382067	15036727	432000004
600	256	0,06609	29414252	65461579	11172868	432000004
600	512	0,06246	28662903	63237443	10084257	432000004
1000	128	0,2737	140224425	127458632	67250253	2000000004
1000	256	0,2894	133665297	347069859	50680488	2000000004
1000	512	0,27263	129654224	291457236	48681726	2000000004
1400	128	0,7536	389207338	359279825	182741221	5488000004
1400	256	0,82346	370155527	923612618	141602772	5488000004
1400	512	0,76825	357599775	805704340	135721900	5488000004
1800	128	1,65166	820631681	779494586	381822935	11664000004
1800	256	1,74387	782148881	1971632968	300075580	11664000004
1800	512	1,67316	761401033	1726511091	285276196	11664000004
2200	128	2,95537	1482491829	1348214456	711305861	21296000004
2200	256	3,12006	1429792161	3656179708	549175774	21296000004
2200	512	2,98882	1391449092	3190130694	529797835	21296000004
2600	128	4,85304	2476897622	2247750555	1188179547	35152000004
2600	256	5,1233	2360209636	5896230042	920373596	35152000004
2600	512	4,8722	2297597253	5106113413	890632224	35152000004
3000	128	7,3746	3787436595	3489402791	1815029806	54000000004
3000	256	7,78308	3609946749	9346491501	1388648686	54000000004
3000	512	7,31631	3509727882	7899819744	1350883012	54000000004
4096	128	21,6676	9565731035	29937612865	4495378283	137438953476
4096	256	19,4179	8999791541	22241567860	5027972725	137438953476
4096	512	18,7817	8758479300	18933675372	4564304880	137438953476
6144	128	73,9134	32113759152	102313048341	16119525996	463856467972

6144	256	66,5563	30397645653	74868583788	16179321154	463856467972
6144	512	64,6322	29583989401	63784981396	15589958303	463856467972
8192	128	179,637	75754117960	240167116988	41609979595	1099511627780
8192	256	243,79	72368193827	164509313853	28011304408	1099511627780
8192	512	232,853	70124617026	134528685317	19936887759	1099511627780
10240	128	346,981	148601305287	475595530374	74817028459	2147483648004
10240	256	309,009	140369044110	348235081605	83717685464	2147483648004
10240	512	312,58	136839209137	294859052025	64901527412	2147483648004

Algoritmo paralelo e simples

Matrix Dimensions	Time (s)	L1 Cache Misses	L2 Cache Misses	L2 Cache Accesses	FLOPS	Efficiency	Speedup
600	0,0404749	30705075	8988016	27377756	54000004	65,06192727%	5,204954181
1000	0,17273	153695668	44906041	138661870	250000004	94,544230888%	7,56353847
1400	0,41558	435678563	192396886	393361510	686000004	101,6576231%	8,132609846
1800	2,57951	1135524758	1124227830	1045806848	1458000004	82,503556888%	6,60028455
2200	5,52412	2207273258	3253130976	2037393245	2662000004	84,928911775%	6,79431294
2600	11,186	3867970333	6933720568	3572508391	4394000004	75,891516188%	6,071321294
3000	18,8936	6286554912	12686372698	5801287049	6750000004	75,426334844%	6,034106787
4096	262,963	18341051262	18006277273	17954549316	17179869188		
6144	1138,32	61924790599	79864215796	60584470235	57982058500		
8192	2916,32	146754673313	170013424643	143952069817	137438953476		
10240	5632,68	287008475265	317393073038	281288857569	268435456004		

Algoritmo paralelo em linha

Matrix Dimensions	Time (s)	L1 Cache Misses	L2 Cache Misses	L2 Cache Accesses	FLOPS	Efficiency	Speedup
600	0,0188999	3393512	6848833	938181	54000004	38,826395991%	3,106111673
1000	0,044564	15712912	31376729	4263369	250000004	93,9497127	7,51597702

						7%	2
1400	0,227986	43240039	77829478	6903272	686000004	75,82483135%	6,065986508
1800	0,583284	93672212	159266900	13355259	1458000004	67,0897453%	5,367179624
2200	1,07923	256618203	292547418	50806648	2662000004	66,08496335%	5,286797068
2600	1,86862	553293290	471162075	132364747	4394000004	63,19269033%	5,055415226
3000	2,74196	850591402	751666032	235178790	6750000004	66,31387766%	5,305110213
4096	7,9926	2186678870	1843339691	604125181	17179869188	58,28047194%	4,662437755
6144	34,2223	7372607795	5874559593	1975653201	57982058500	46,06170538%	3,68493643
8192	95,4598	17459038519	13442470855	4959660251	137438953476	39,13728606%	3,130982885
10240	245,672	34041833854	25311721410	7667877350	268435456004	29,81007197%	2,384805757