Paradigmas de Computação Paralela

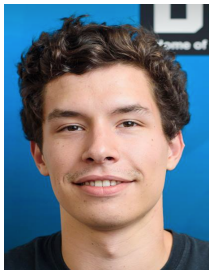
4º ano de MIEI

1º Fase de Entrega

Relatório de Desenvolvimento

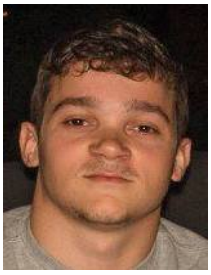
Heat Plate - Heat diffusion

Humberto Vaz



a73236

João Dias



A72095

25 de Novembro de 2017

Resumo

O presente documento apresenta o desenvolvimento do projeto de *Paradigmas de Computação Paralela* referente ao Mestrado Integrado de Engenharia Informática *(MIEI)* correspondente ao ano lectivo *2017/2018*, analisando o problema e as estratégias utilizadas para resolver o mesmo. O problema apresentado consiste em representar a evolução da difusão do calor ao longo de um numero máximo de iterações, casos como este tem sido profundamente estudados sendo também conhecidos por "Heat plate", o que se demonstra neste problema é a heurística utilizada consiste no calculo da média da soma do valor da célula com as suas vizinhas (linha e coluna).

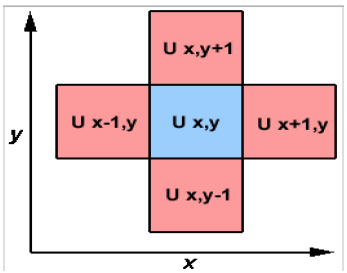


Figura 1: Cálculo da média

A utilização de um plano retangular na resolução do exercício leva-nos a optar por uma estrutura matricial, ficando assim a própria propagação do calor dependente desse "factor físico".

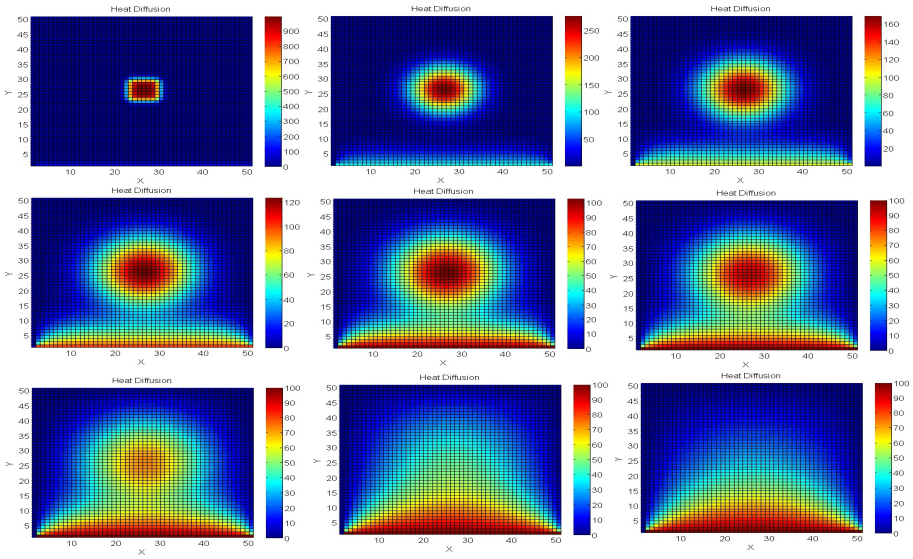


Figura 2: Exemplos de difusão de calor

Capítulo 1

Abordagem ao problema

1.0.1 Considerações Gerais

Através da analise do pseudo-código conseguimos reparar que existem 2 tipos de dependências no problema em questão, uma para os valores que existem para a obtenção do novo valor da temperatura, ou seja, dependências para com os vizinhos (Read after write (RAW) e Write after write (WAW)) que nos impossibilitam fazer a computação do valor na própria matriz, e posteriormente o bottleneck de cada iteração, ou seja, iniciamos a nova iteração apenas quando a ultima acaba.

A escolha das matrizes para as quais efetuamos as medidas tiveram a sua própria ciência, esta foi realizada tendo em conta os diversos tamanhos de Cache e memória RAM apresentados pelos nós do cluster. Para obter estes dados executa-se o comando *lscpu*.

Os testes posteriormente referidos são relativos a 3 computadores:

Designação no Cluster	**662**	**652**	**641**
Microarquitetura do CPU	Ivy Bridge	Ivy Bridge	Ivy Bridge
Designação do CPU	Dual CPU E5-2695v2	Dual CPU E5-2670v2	Dual CPU E5-2650v2
Cores	24 cores fisicos , 48 com ht	20 cores fisicos , 40 com ht	16 cores fisicos , 32 com ht
Frequência do clock	2.5ghz	2.5ghz	2.6ghz
Cache L1	L1d:32k;L1i:32k	L1d:32k;L1i:32k	L1d:32k;L1i:32k
Cache L2	256k	256k	256k
Cache L3	30720k	25600	20480
RAM	64GB	64GB	64GB

Através desta analise inicial identificamos a ideologia de teste a realizar:

Dados que fiquem inseridos na :

- Cache L1;
- Cache L2;
- Cache L3;
- RAM;

Este tema será mais aprofundado futuramente.

Capítulo 2

Código Sequencial

2.1 Abordagem ao problema

Iniciamos o ataque ao problema de duas formas, uma das implementações é uma forma mais natural cedida no próprio enunciado do problema que consiste no seguinte:

```
iteration() {
for(int i=1; i<N-1; i++) {
    for(int j=1; j<N-1; j++)
    G1[i][j] = ( G2[i-1][j]+ G2[i+1][j]+ G2[i][j-1]+ G2[i][j+1]+ G2[i][j]) / 5;
    }
}
... // copia G1 para G2
```

Aqui, após a analise do pseudo código, já é possível tirar algumas ilações sobre o comportamento da função no seu acesso à memoria dependendo do tamanho do Dataset que lhe é atribuído, assim sendo, estruturamos varias opções de matrizes.

- Matriz em Cache;

Uma possível solução seria usar apenas matrizes que coubessem na Cache a qual queríamos aceder, para isso teríamos usar uma formula deste tipo $\sqrt{(Cachesize/(\#Matrizes_usadas * sizeof(double))}$;

- "Linha"em Cache;

outra opção , solução sobre a qual mais nos focamos foi a utilização de matrizes para as quais o tamanho da linha triplicado (devido a dependência de dados que nos obriga ao uso de 3 linhas (i-1, i, i+1) e posteriormente duplicado por 2, pois usamos duas matrizes. Esta metodologia tem como intuito testar a utilização das diversas caches no decorrer do problema, com isto queremos dizer que iniciamos com um tamanho em que a estrutura consumida pela thread coubesse na cache L1, posteriormente na L2 e assim sucessivamente até ele ser obrigado a utilizar a RAM externa.
Outra atenção que tivemos foi deixar que as linhas fossem múltiplos de 8, visto que a Cache ao guardar um valor carrega sempre 64 bytes. Devemos então ter uma linha de doubles múltiplos de 8 para garantir que ao carregar valores para a cache evitamos carregar lixo que é enviado quando se faz o padding.

2.1.1 Considerações do Cód. Sequencial

Uma análise inicial ao algoritmo demonstra que a sua complexidade ronda o

$$\theta = N^2$$

infelizmente este valor não vai ser possível reduzir, assim sendo, apostamos os nossos esforços então na diminuição do esforço computacional no cálculo do mesmo.

Das duas dependências referidas inicialmente conseguimos resolver a primeira recorrendo à utilização de uma matriz auxiliar, a introdução destes valores não acarretará o risco de acesso e alteração de um valor que posteriormente poderá novamente ser utilizado.
Melhorias de performance aplicadas a este código resumem-se a utilização de uma multiplicação por 0.2 e não a divisão do valor das somas finais por 5, o que nos poupa nos CPI desta operação.
Foram ainda abordados dois tipos de cópia para a copia para a matriz temporária, estes são:

- Cópia iterativa
- Swap de apontadores das estruturas

Posteriormente foi abordada a opção de cálculo de matrizes por blocos, a utilização deste método será uma mais valia para matrizes de grades dimensões, isto é descoberto através de uma analise comparativa entre os tempos obtidos pelos restantes códigos paralelos.

2.1.2 Otimização por Blocos

A técnica de divisão de trabalho por blocos tem como objetivo aumentar a performance garantindo que cada thread acederá sempre a Cache L1 L2, sendo assim reduzimos os blocos a 32x32 e 64x64 e ainda um modo iterativo que parte a matriz pelo numero de threads existentes (Threads*Threads, representativo dos blocos), infelizmente, por falta de tempo não podemos explorar completamente esta implementação, fazendo apenas a sua aplicação num dos CPU do Cluster (652) e limitados numero de teste, mas, através dos testes efetuados e dos dados obtidos conseguimos observar uma escalabilidade para matrizes de grandes dimensões muito superiores àquilo que era de esperar comparativamente às outras implementações efetuadas para o máximo numero de threads definido. Uma possível razão para este resultado favorável poderá ser proximidade da Cache acedida, porém os resultados obtidos são um bocado dispares fazendo-nos duvidar sobre a sua veracidade.

2.1.3 Considerações Finais sobre Cód. Sequencial

Após uma analise e comparação entre os tempos realizados para os nossos diversos casos teste visualizamos um desempenho de performance bastante superior no caso em que usamos um **swap** de apontadores, o que faz sentido, visto que esse trabalho computacional é bastante reduzido comparativamente à copia iterativa dos elementos de uma matriz para a outra. Podemos já esperar que esse tipo de ganhos se continuem a manter para a versão paralela.
O que na comparação de tempos demonstra que é real (662):

20000 x 20000	1	2	3	4	5	mínimo	speed-up
tseq	2,953975	2,982339	2,952885	2,897951	2,944187	2,897951	-
2	1,666696	1,68382	1,672705	1,707484	1,685028	1,666696	1,73874
4	0,902377	0,897415	0,899167	0,893863	0,915658	0,893863	3,242053
8	0,533504	0,547363	0,53186	0,531571	0,51985	0,51985	5,574591
16	0,423528	0,437975	0,427491	0,445486	0,445064	0,423528	6,842407
32	0,385326	0,381398	0,379156	0,387331	0,365332	0,365332	7,932377
40	0,381996	0,366953	0,365847	0,358843	0,363103	0,358843	8,075819

Figura 2.1: Iterativo 20000x20000 (662)

Como é possível verificar pela analise de tempos existe de facto uma melhor performance por parte do código com o Swap de apontadores ao longo de todo o paralelismo. E outra coisa interessante é a capacidade que o código Swap mesmo em formato de paralelismo ser ter melhor escalabilidade que o iterativo, uma possibilidade de isto ocorrer será mesmo a questão dos cache miss e tempo que se perde com isso ao carregar iterativamente os valores para uma estrutura auxiliar.

20000 x 2000	1	2	3	4	5	mínimo	speed-up
tseq	2,35715	2,39111	2,33265	2,31822	2,31507	2,31507	-
2	1,33844	1,34202	1,30829	1,30643	1,30007	1,30007	1,78072
4	0,70362	0,68651	0,68449	0,68535	0,69005	0,68449	3,38217
8	0,37093	0,38246	0,3782	0,37784	0,46136	0,37093	6,24131
16	0,27559	0,28306	0,27711	0,26893	0,27952	0,26893	8,60844
32	0,22271	0,23145	0,23656	0,23531	0,23529	0,22271	10,3949
40	0,22604	0,20684	0,22534	0,21211	0,20847	0,20684	11,1928

Figura 2.2: Swap 20000x20000 (662)

2.2 PAPI

2.2.1 Algumas considerações

Iniciamos a análise das cache misses para as várias caches utilizando esta ferramenta, no entanto não conseguimos tirar partido da mesma devido à tardia implementação do código que serviria de base. Não afirmando que será esta a ferramenta a escolhida, mas como trabalho futuro sabemos que trará boas métricas para a procura e investigação de resultados.

Capítulo 3

Código Paralelo

3.0.1 Considerações Código Paralelo

Previamente a fazer qualquer medição sabemos que este código possui dependências como já tinha sido referido por isso não se trata de um algoritmo embaraçosamente paralelo, essas dependênciias não são completamente contornáveis o que vai sempre limitar os ganhos da performance.

Em todas as resoluções implementadas a granularidade do paralelismo é de "grão grosso", sendo que está subentendido que o paralelismo é efetuado num dos ciclos superiores, no nosso caso essa implementação é realizada sobre a iteração das linhas ou por colunas (no caso da abordagem por blocos).

3.1 Análise de resultados

Como era de esperar, de forma geral ,os speed-ups são maiores para matrizes maiores. Isto deve-se a haver mais trabalho para ser distribuído entre as várias threads. Para as matrizes de tamanho inferior (400x400) verificamos que o qualquer ganho obtido com a implementação de paralelismo é´ perdido pelo overhead de criação e distribuição de trabalho pelas threads. Para ficheiros de input abaixo de 1000x1000, o uso de 8 ou 16 threads permitem algum ganho de performance e será ótimo, visto já haver mais trabalho a ser distribuudo entre estas. Para 40 threads dependendo do tamanho da matriz e do próprio algoritmo aplicado pode ou não haver ganhos, (ganhos mais visíveis na aplicação do caso iterativo, pois possui maior computação para ser distribuída).

Podemos dizer que não se atinge sequer um speed-up "perfeito",contudo apróximamo-nos dele, provando assim a escalabilidade deste algoritmo.Contudo não consimos chegar ao melhor valor teórico.

Para cada CPU diferente foram feitos 30 testes (5 testes para diversos numeros de threads), cujos resultados foram guardados em tabelas para posterior análise. Decidimos apresentar os speedups para o algoritmo paralelo com swap pois este foi aquele que a nossa análise predominou. Para cada conjunto de 5 testes (k-best) calculamos o mínimo dos tempos de execução e o seu speed-up, que podem ser vistos nos seguinte gráficos:

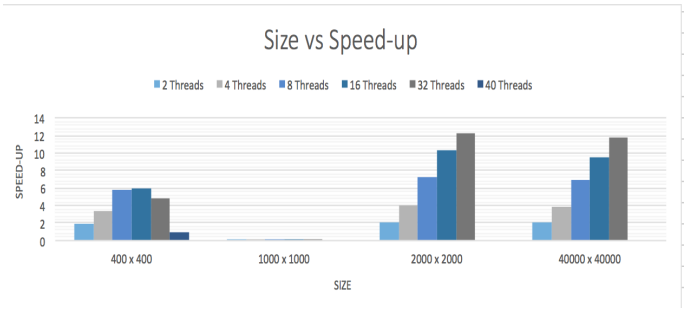


Figura 3.1: Size vs SpeedUp para o r652 (Paralelo com cópia por "swap")

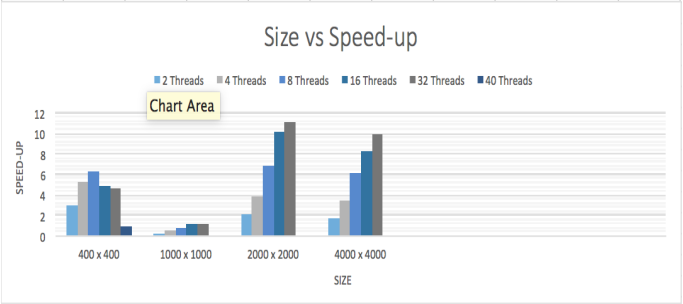


Figura 3.2: Size vs SpeedUp para o r662 (Paralelo com cópia por "swap")

Da figura anterior consegue-se perceber que o Dataset com menor speedup é o de 1000x1000. Uma possível razão poderá ser o tempo desnecessário no acesso ás várias Caches L1, L2 e L3 e só no final à RAM, ou seja um elevado indice de Cache misses que trará consecutivamente um valor alto de miss penalty.

Outro Dataset que não apresenta sempre um aumento "exponencial"é o de 400x400, no entanto tal é o expectável pois o overhead do escalonamento para threads superiores a 8 é demasiado elevado para este Dataset.

3.2 Trabalho futuro

No decorrer do projeto identificamos varias abordagens para o mesmo, algumas foram abandonadas e outras não foram completamente implementadas, mas acreditamos na sua capacidade de escalabilidade.

- Possíveis Melhorias:

Idealizamos outra resposta para comutar o bottleneck existente das "iterações globais"(*iter++;*), esta consiste na numeração das (threads) existentes e a computação de da matriz por determinada ordem, no nosso caso percorrida de cima para baixo, da esquerda para a direita. Assim sendo poderíamos correr paralelamente a computação da matriz por secções, e quando uma thread acabasse o seu trabalho via se havia uma secção posterior a precisar de ser computada, caso assim não fosse, via se as suas antecessoras tinham acabado o seu trabalho e podia iniciar a próxima iteração sem haver o problema de poder alterar dados sobre os quais alguém poderia processar.

Outra possível melhoria seria o carregamento para Cache das linhas com valores partilhados para a computação especialmente aquando do **HyperThreading**. Uma forma de minimizar este problema é a capacidade de **Set Afiliation** onde poderíamos "obrigar"a Thread do core físico a correr uma linha enquanto a Thread do HyperThreading corre a seguinte, havendo assim um maior numero de memória comum na cache e um possível ganho na performance pois teríamos um numero superior de hits sem haver a necessidade de carregar essa memoria para cache.

A utilização de memoria alinhada iria permitir que os carregamentos para cache não trouxessem dados que não nos eram relevantes (lixo) para a execução do algoritmo. Esta medida apesar de não ter sido testada nesta fase, muito provavelmente iria contribuir para ganhos de performance. Isto não foi medido pois não conseguimos implementar o Papi convenientemente, complicando assim a nossa possível analise de dados e posterior tomada de ilações

Por fim temos o caso do código de blocos. Na nossa abordagem, as fronteiras de cada bloco são calculada, porém existem outras aplicações sobre as quais esta pode não ser a melhor aproximação, sendo que utilizar blocos com mais e menos 1 elemento na sua região fronteira (i-1, j-1, etc.) vai aumentar-nos a computação mais vai diminui os carregamentos para cache com padding caso esta mini-matriz seja múltipla de 8. Existe ainda a opção de ignorar estes elementos e não os computar, porem sentimos que essa não seria a abordagem mais correta ao problema.

Capítulo 4

Conclusão

Terminado o primeiro projeto, depois de feita uma minuciosa avaliação do problema, chegou-se a conclusões, apresentadas ao longo do corpo deste documento, que se consideram ser satisfatórias.

Pensa-se que este relatório é capaz de apresentar explicações detalhadas acerca da performance da implementação paralela comparada com a sequencial do algorítmo heatplate.

Com isto pretende-se afirmar que as conclusões chegadas e o plano de acção elaborado terá sido bastante satisfatório tendo conseguido realizar o que nos foi proposto de diferentes formas e com uma estratégia e lógica clara onde qualquer pormenor foi refletido e nunca menosprezado.

Capítulo 5

Anexos

5.1 Medições 662

5.1.1 Iterativo

400 x 400	1	2	3	4	5	mínimo	speed-up
tseq	0,000594	0,000581	0,000586	0,000597	0,000602	0,000581	-
2	0,009711	0,010208	0,012485	0,012679	0,010205	0,009711	0,059829
4	0,00987	0,009757	0,009981	0,009551	0,009971	0,009551	0,060831
8	0,009693	0,00995	0,010056	0,009969	0,009959	0,009693	0,05994
16	0,010204	0,011447	0,012225	0,009992	0,009795	0,009795	0,059316
32	0,01005	0,01025	0,009639	0,010189	0,012868	0,009639	0,060276
40	0,009092	0,009963	0,011261	0,009889	0,010619	0,009092	0,063902

Figura 5.1: Iterativo 400x400

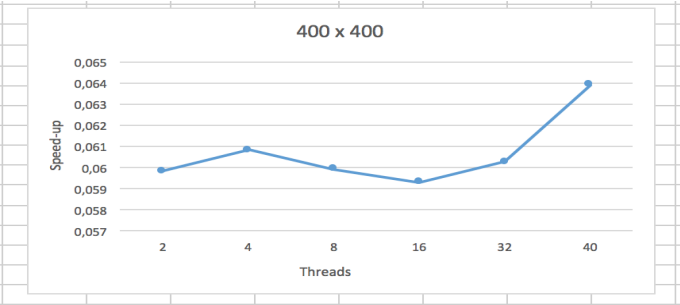


Figura 5.2: Curva Iterativo 400x400

4000 x 4000	1	2	3	4	5	mínimo	speed-up
tseq	0,110056	0,11112	0,111586	0,113015	0,111024	0,110056	-
2	0,111024	0,060216	0,059834	0,060063	0,059857	0,059834	1,839356
4	0,03204	0,032184	0,03183	0,032261	0,032082	0,03183	3,457619
8	0,0195	0,019542	0,020013	0,019699	0,019398	0,019398	5,673575
16	0,016469	0,016003	0,01594	0,01584	0,015979	0,01584	6,94798
32	0,014323	0,014323	0,014648	0,01458	0,014671	0,014323	7,683865
40	0,014332	0,014115	0,014181	0,014359	0,014129	0,014115	7,797095

Figura 5.3: Iterativo 4000x4000

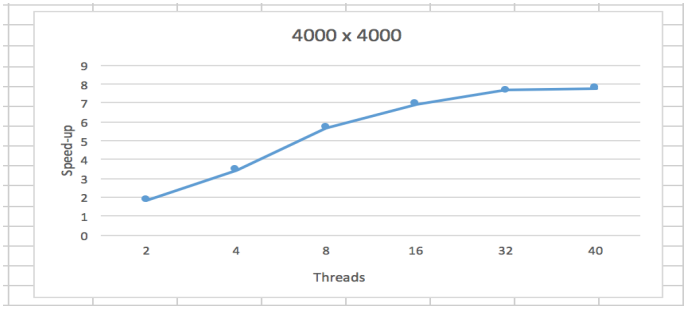


Figura 5.4: Curva Iterativo 4000x4000

10000x10000	1	2	3	4	5	Mínimo	speed-up
tseq	0,701838	0,960578	0,699582	0,725111	0,713239	0,699582	-
2	0,363294	0,369954	0,372071	0,375250	0,361667	0,361667	1,934326328
4	0,218880	0,221737	0,226226	0,219706	0,221636	0,218880	3,196189693
8	0,144007	0,148198	0,149704	0,146706	0,140250	0,140250	4,988106952
16	0,120907	0,121675	0,123889	0,121745	0,133822	0,120907	5,786116602
32	0,114920	0,110026	0,110379	0,115898	0,110066	0,110026	6,358333485
40	0,109721	0,109849	0,109450	0,109876	0,109066	0,109066	6,414299598

Figura 5.5: Iterativo 10000x10000

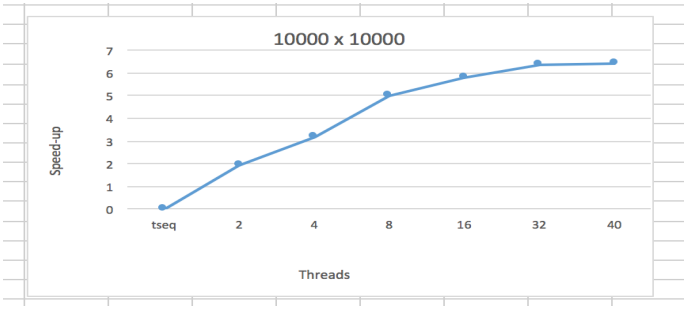


Figura 5.6: Curva Iterativo 10000x10000

20000 x 20000	1	2	3	4	5	mínimo	speed-up
tseq	2,953975	2,982339	2,952885	2,897951	2,944187	2,897951	-
2	1,666696	1,68382	1,672705	1,707484	1,685028	1,666696	1,73874
4	0,902377	0,897415	0,899167	0,893863	0,915658	0,893863	3,242053
8	0,533504	0,547363	0,53186	0,531571	0,51985	0,51985	5,574591
16	0,423528	0,437975	0,427491	0,445486	0,445064	0,423528	6,842407
32	0,385326	0,381398	0,379156	0,387331	0,365332	0,365332	7,932377
40	0,381996	0,366953	0,365847	0,358843	0,363103	0,358843	8,075819

Figura 5.7: Iterativo 20000x20000

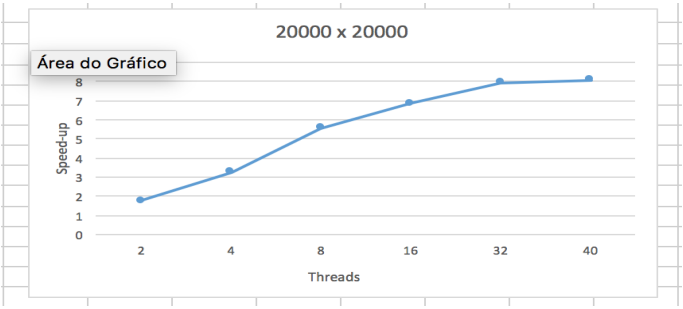


Figura 5.8: Curva Iterativo 20000x20000

5.1.2 Swap

400 x 400	1	2	3	4	5	mínimo	speed-up
tseq	0,00043	0,00042	0,00042	0,00043	0,00043	0,00042	-
2	0,00015	0,00014	0,00014	0,00014	0,00016	0,00014	3
4	8,3E-05	9,3E-05	9,7E-05	8,2E-05	0,00011	8,2E-05	5,15854
8	8,4E-05	6,7E-05	7,7E-05	8,3E-05	7,4E-05	6,7E-05	6,31343
16	8,7E-05	8,9E-05	8,6E-05	0,00011	9,1E-05	8,6E-05	4,9186
32	0,00011	0,00012	0,00011	0,00012	9,1E-05	9,1E-05	4,64835
40	0,00011	0,00011	0,0001530,000109		0,00012	0,00011	3,91667

Figura 5.9: Swap 400x400

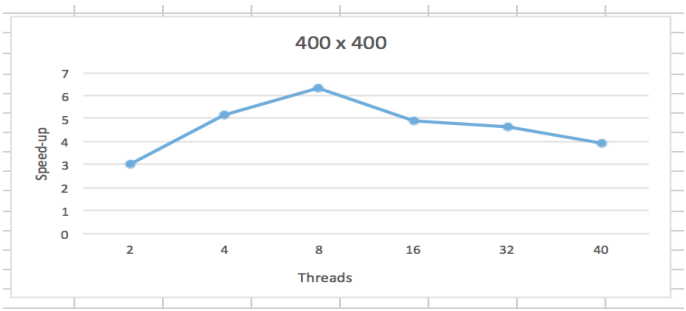


Figura 5.10: Curva c/ swap 400x400

1000 x 1000	1	2	3	4	5	mínimo	speed-up
tseq	0,00043	0,00042	0,00042	0,00043	0,00043	0,00042	-
2	0,00193	0,00188	0,00187	0,00186	0,00187	0,00186	0,22779
4	0,001	0,001	0,001	0,001	0,001	0,001	0,4247
8	0,00055	0,00053	0,00055	0,00054	0,00054	0,00053	0,79511
16	0,00041	0,0004	0,00038	0,00041	0,00039	0,00038	1,10444
32	0,00039	0,00038	0,0004	0,00038	0,00042	0,00038	1,128
40	0,0004	0,00039	0,0004	0,00038	0,00041	0,00038	1,10156

Figura 5.11: Swap 1000x1000

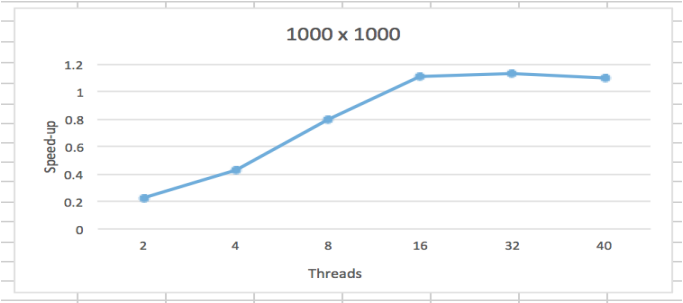


Figura 5.12: Curva c/ swap 1000x1000

2000 x 2000	1	2	3	4	5	mínimo	speed-up
tseq	0,01974	0,01978	0,01953	0,01975	0,01942	0,01942	-
2	0,0099	0,00989	0,00979	0,00986	0,00989	0,00979	1,98356
4	0,00519	0,00532	0,00539	0,00532	0,00525	0,00519	3,74128
8	0,00283	0,00289	0,00293	0,0029	0,00292	0,00283	6,8577
16	0,00212	0,00193	0,00213	0,00204	0,00206	0,00193	10,0523
32	0,00181	0,00177	0,00188	0,00184	0,00175	0,00175	11,0977
40	0,00171	0,00166	0,00177	0,00183	0,00176	0,00166	11,6924

Figura 5.13: Swap 2000x2000

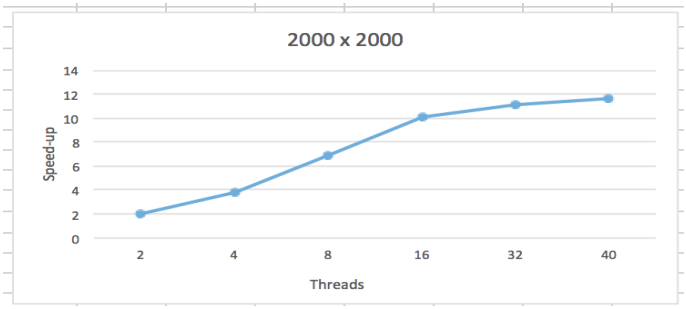


Figura 5.14: Curva c/ swap 2000x2000

4000 x 4000	1	2	3	4	5		mínimo		speed-up
tseq	0,08211	0,08278	0,08176	0,0827	0,08238		0,08176		-
2	0,04513	0,04574	0,0455	0,04624	0,04575		0,04513		1,81169
4	0,04574	0,02454	0,02394	0,02387	0,0239		0,02387		3,42557
8	0,0455	0,01353	0,01366	0,01345	0,01357		0,01345		6,07776
16	0,04624	0,01	0,01049	0,00991	0,01032		0,00991		8,25255
32	0,04575	0,00849	0,00861	0,00824	0,00887		0,00824		9,92209
40	0,04513	0,0081	0,00784	0,00789	0,00855		0,00784		10,431

Figura 5.15: Swap 4000x4000

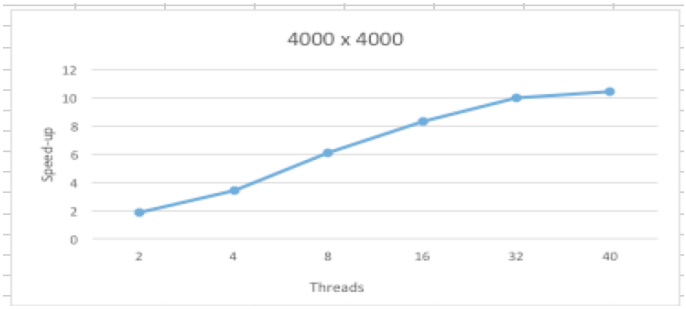


Figura 5.16: Curva c/ swap 4000x4000

10000 x 1000	1	2	3	4	5	mínimo	speed-up
tseq	0,54144	0,53981	0,54049	0,53955	0,54139	0,53955	-
2	0,3093	0,30385	0,30317	0,30283	0,30123	0,30123	1,79113
4	0,15937	0,16094	0,16152	0,16049	0,1621	0,15937	3,38558
8	0,08672	0,08916	0,08969	0,09002	0,08898	0,08672	6,22146
16	0,06352	0,06419	0,06782	0,06548	0,06663	0,06352	8,49484
32	0,05471	0,05452	0,05547	0,0578	0,05702	0,05452	9,89655
40	0,0517	0,04968	0,05257	0,05089	0,05175	0,04968	10,8603

Figura 5.17: Swap 10000x10000

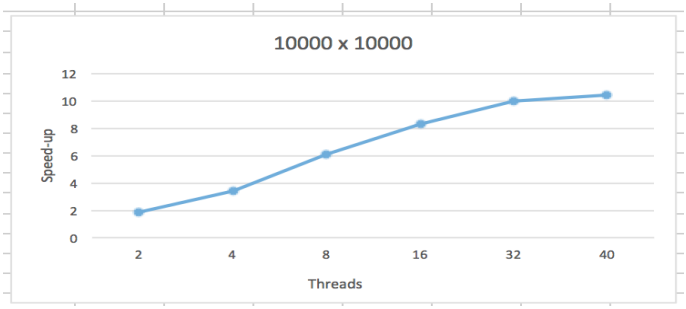


Figura 5.18: Curva c/ swap 10000x10000

20000 x 2000	1	2	3	4	5	mínimo	speed-up
tseq	2,35715	2,39111	2,33265	2,31822	2,31507	2,31507	-
2	1,33844	1,34202	1,30829	1,30643	1,30007	1,30007	1,78072
4	0,70362	0,68651	0,68449	0,68535	0,69005	0,68449	3,38217
8	0,37093	0,38246	0,3782	0,37784	0,46136	0,37093	6,24131
16	0,27559	0,28306	0,27711	0,26893	0,27952	0,26893	8,60844
32	0,22271	0,23145	0,23656	0,23531	0,23529	0,22271	10,3949
40	0,22604	0,20684	0,22534	0,21211	0,20847	0,20684	11,1928

Figura 5.19: Swap 20000x20000

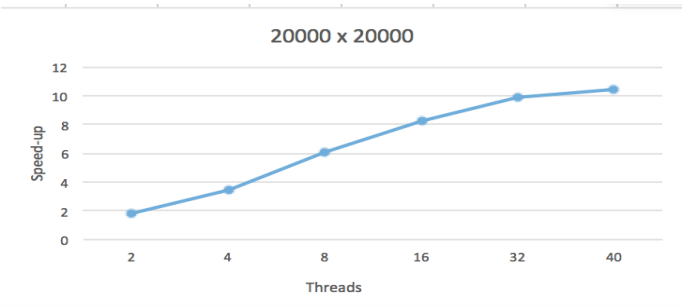


Figura 5.20: Curva c/ swap 20000x20000

40000 x 4000	1	2	3	4	5		mínimo		speed-up
tseq	9,29108	9,33318	9,46663	9,23393	9,47001		9,23393		-
2	5,30956	5,28701	5,30651	5,28788	5,41567		5,28701		1,74653
4	2,7585	2,80606	2,80001	2,78645	2,78668		2,7585		3,34744
8	1,50028	1,52269	1,54921	1,58173	1,5351		1,50028		6,15479
16	1,1363	1,18712	1,08769	1,11861	1,08985		1,08769		8,4895
32	0,91986	0,95571	0,99541	0,98779	0,95877		0,91986		10,0385
40	0,88407	0,89926	0,93805	0,87864	0,91306		0,87864		10,5093

Figura 5.21: Swap 40000x40000

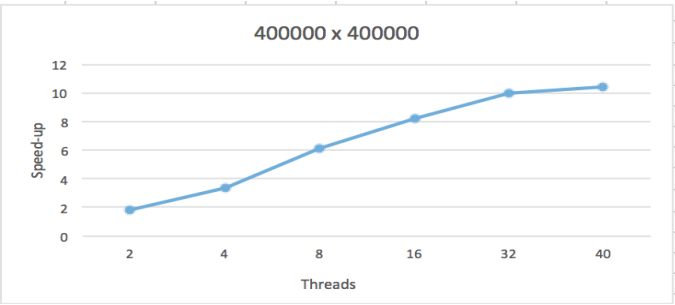


Figura 5.22: Curva c/ swap 40000x40000

40000 x 40000	1	2	3	4	5	mínimo	speed-up
tseq	11,9123	11,93189	12,17382	11,89274	11,53692	11,53692	-
2	6,88054	6,82073	6,927839	6,979214	6,786286	6,786286	1,700035
4	3,695992	3,620451	3,646256	3,639493	3,577954	3,577954	3,224447
8	2,247671	2,180606	2,192755	2,143502	2,177033	2,143502	5,382278
16	1,706194	1,762661	1,665448	1,763246	1,766213	1,665448	6,92722
32	1,518176	1,599563	1,598055	1,615912	1,618249	1,518176	7,599201
40	1,50618	1,595776	1,492553	1,630799	1,558231	1,492553	7,729658

Figura 5.23: Iterativo 40000x40000

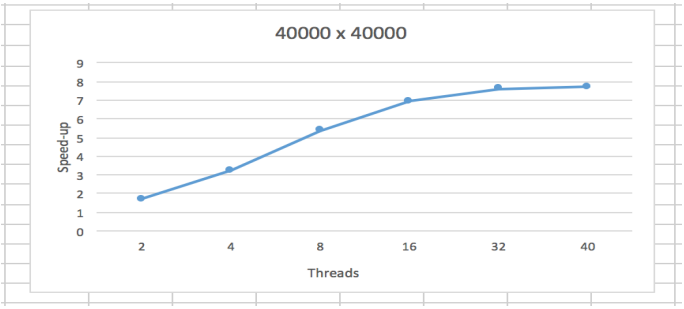


Figura 5.24: Curva Iterativo 40000x40000

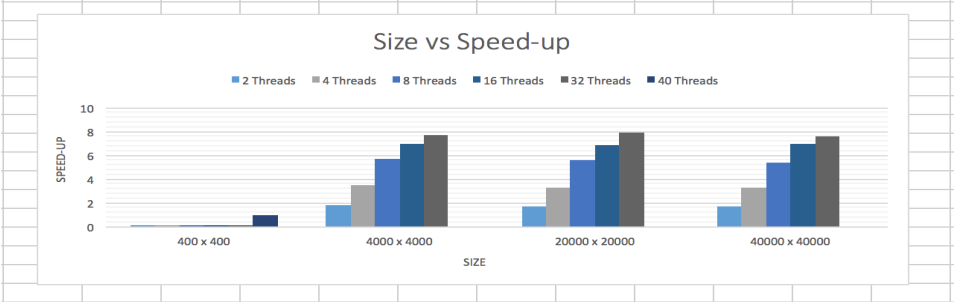


Figura 5.25: Analise geral ao SpeedUp Iterativo

Size \ Threads	2	4	8	16	32
400 x 400	0,05982906	0,060831327	0,059940163	0,059315978	0,060275962
4000 x 4000	1,83935555	3,457618599	5,673574595	6,947979798	7,683865112
20000 x 20000	1,738739998	3,242052753	5,574590747	6,842407114	7,932376578
40000 x 40000	1,700035041	3,224447268	5,38227816	6,927219583	7,59920062

Figura 5.26: Analise geral ao SpeedUp Iterativo

5.2 Medições 652

5.2.1 Iterativo

400 x 400	1	2	3	4	5	mínimo	speed-up
tseq	0,000527	0,000518	0,000428	0,000518	0,000523	0,000428	-
2	0,000353	0,000384	0,000392	0,000375	0,000362	0,000353	1,212465
4	0,000123	0,000235	0,000197	0,000224	0,000197	0,000123	3,479675
8	0,000092	0,000113	0,000114	0,000107	0,000108	0,000092	4,652174
16	0,000087	0,000092	0,000095	0,0001	0,000096	0,000087	4,91954
32	0,000118	0,000108	0,000132	0,00011	0,000113	0,000108	3,962963
40	0,007082	0,00991	0,008151	0,009124	0,010756	0,007082	0,060435

Figura 5.27: Iterativo 400x400

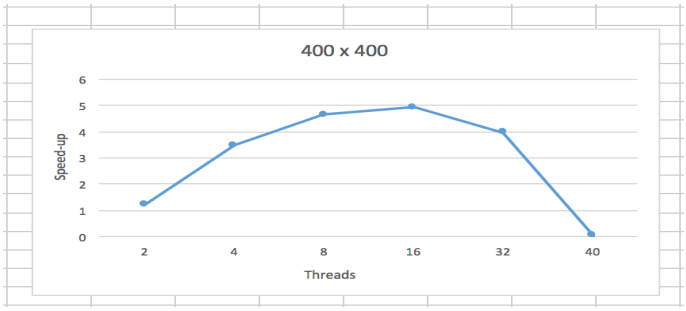


Figura 5.28: Curva Iterativo 400x400

4000 x 4000	1	2	3	4	5	mínimo	speed-up
tseq	0,100952	0,100875	0,100213	0,103842	0,100731	0,100213	-
2	0,053669	0,053794	0,05368	0,056173	0,051212	0,051212	1,956827
4	0,029121	0,028742	0,028599	0,028632	0,028754	0,028599	3,504074
8	0,017203	0,017266	0,01811	0,017306	0,017495	0,017203	5,825321
16	0,014347	0,01436	0,01423	0,014317	0,014885	0,01423	7,042375
32	0,012616	0,012717	0,012501	0,012547	0,01255	0,012501	8,016399
40	0,01878	0,013888	0,023403	0,022445	0,022526	0,013888	7,215798

Figura 5.29: Iterativo 4000x4000

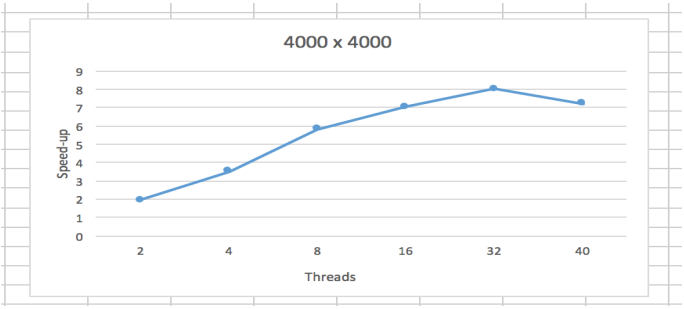


Figura 5.30: Curva Iterativo 4000x4000

20000 x 20000	1	2	3	4	5	mínimo	speed-up
tseq	2,315342	2,385659	2,392193	2,795197	2,328816	2,315342	-
2	1,271843	1,30182	1,305512	1,315835	1,471161	1,271843	1,820462
4	0,778187	0,78473	0,786282	0,78763	0,788512	0,778187	2,975303
8	0,457157	0,463614	0,456881	0,459053	0,458765	0,456881	5,067713
16	0,355892	0,335613	0,344735	0,354523	0,350366	0,335613	6,898845
32	0,298042	0,314362	0,308223	0,302416	0,305686	0,298042	7,768509
40	0,313417	0,312122	0,322678	0,318022	0,317655	0,312122	7,418067

Figura 5.31: Iterativo 20000x20000

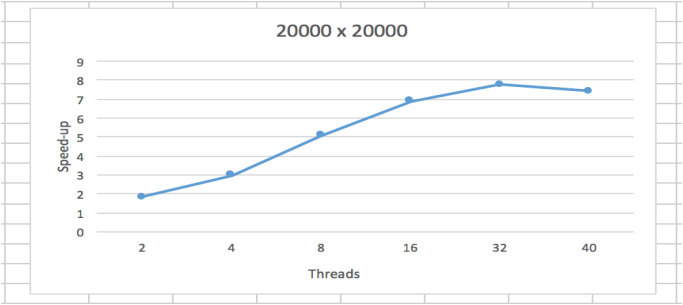


Figura 5.32: Curva Iterativo 20000x20000

40000 x 40000	1	2	3	4	5	mínimo	speed-up
tseq	9,510761	9,782072	9,219152	9,723804	9,727605	9,219152	-
2	6,10173	6,054628	6,043019	5,754316	6,145953	5,754316	1,602128
4	3,241832	3,215336	3,249606	3,041043	3,239423	3,041043	3,031576
8	1,718466	1,745373	1,781822	1,747842	1,768897	1,718466	5,364757
16	1,366499	1,460616	1,485497	1,595381	1,434905	1,366499	6,746549
32	1,291896	1,302525	1,247577	1,313077	1,289465	1,247577	7,389646
40	1,25707	1,261958	1,222838	1,241109	1,25621	1,222838	7,539144

Figura 5.33: Iterativo 40000x40000

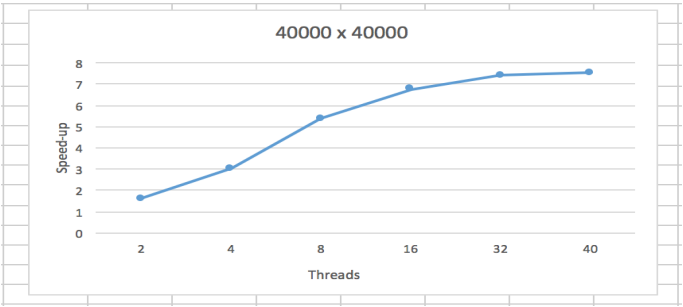


Figura 5.34: Curva Iterativo 40000x40000

Size \ Threads	2	4	8	16	32
400 x 400	3	5,158536585	6,313432836	4,918604651	4,648351648
1000 x 1000	0,227786753	0,424698795	0,795112782	1,104438642	1,128
2000 x 2000	1,983556327	3,74128299	6,85769774	10,05227743	11,09771429
4000 x 4000	1,81169119	3,425566682	6,077757954	8,252548703	9,922087379

Figura 5.35: Analise geral ao SpeedUp Iterativo 652

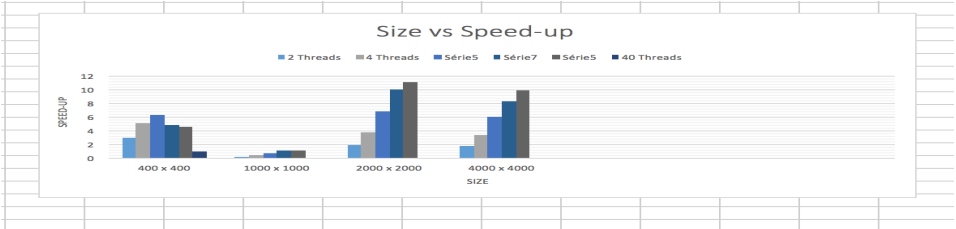


Figura 5.36: Analise geral ao SpeedUp Iterativo 652

5.2.2 Swap

400 x 400	1	2	3	4	5	mínimo	speed-up
tseq	0,000428	0,000424	0,000423	0,000427	0,000431	0,000423	-
2	0,000225	0,000223	0,000223	0,000225	0,000223	0,000223	1,896861
4	0,000126	0,000127	0,000125	0,000129	0,000162	0,000125	3,384
8	0,00008	0,000079	0,000079	0,000074	0,000075	0,000074	5,716216
16	0,000097	0,000073	0,000072	0,000098	0,000074	0,000072	5,875
32	0,000157	0,000089	0,000109	0,000114	0,000089	0,000089	4,752809
40	0,005197	0,005404	0,004312	0,005411	0,004567	0,004312	0,098098

Figura 5.37: Swap 400x400

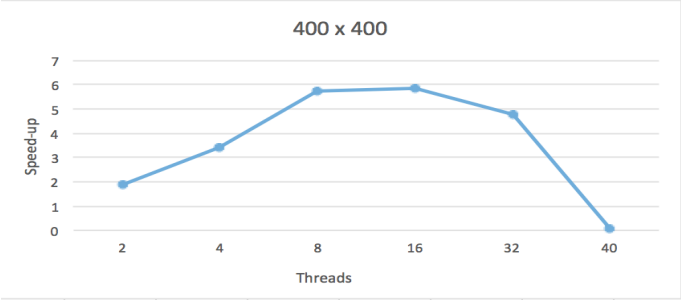


Figura 5.38: Curva Swap 400x400

1000 x 1000	1	2	3	4	5	mínimo	speed-up
tseq	0,000428	0,000423	0,000424	0,000431	0,000427	0,000423	-
2	0,275909	0,273524	0,273172	0,273259	0,277242	0,273172	0,001548
4	0,144878	0,145543	0,142766	0,145963	0,145044	0,142766	0,002963
8	0,082225	0,078172	0,079028	0,081113	0,078852	0,078172	0,005411
16	0,055759	0,056255	0,056128	0,057704	0,054307	0,054307	0,007789
32	0,043632	0,044886	0,044852	0,043638	0,044018	0,043632	0,009695
40	0,048469	0,048072	0,047381	0,049892	0,045196	0,045196	0,009359

Figura 5.39: Swap 1000x1000

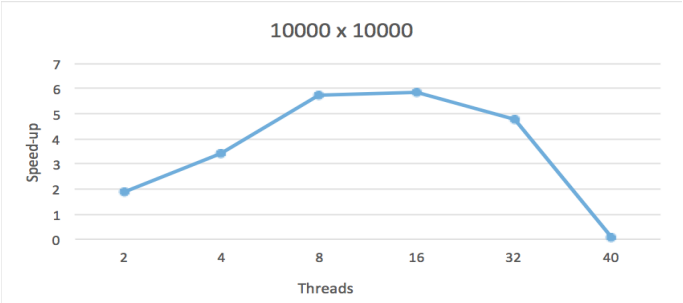


Figura 5.40: Curva Swap 1000x1000

2000 x 2000	1	2	3	4	5		mínimo		speed-up
tseq	0,019736	0,019778	0,019534	0,019747	0,019421		0,019421		-
2	0,009144	0,009087	0,009171	0,00914	0,009178		0,009087		2,137229
4	0,004933	0,004877	0,004882	0,004922	0,004948		0,004877		3,982161
8	0,002658	0,003185	0,003434	0,003588	0,003923		0,002658		7,306622
16	0,001882	0,002561	0,002559	0,001999	0,002697		0,001882		10,31934
32	0,001748	0,002092	0,0017	0,002271	0,001591		0,001591		12,20679
40	0,005034	0,007204	0,005931	0,006963	0,007263		0,005034		3,857966

Figura 5.41: Swap 2000x2000

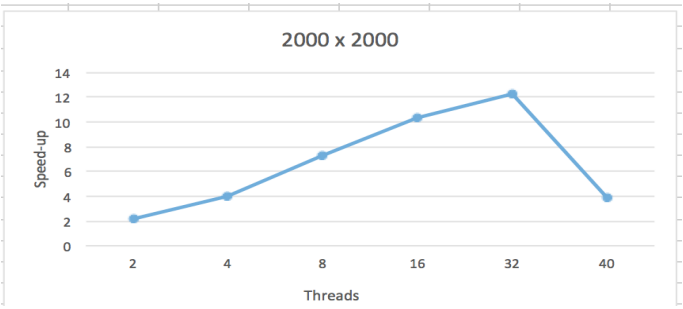


Figura 5.42: Curva Swap 2000x2000

4000 x 4000	1	2	3	4	5		mínimo		speed-up
tseq	0,082105	0,082779	0,081758	0,082702	0,082384		0,081758		-
2	0,041133	0,040721	0,040735	0,04061	0,040773		0,04061		2,013248
4	0,021692	0,021924	0,021691	0,021861	0,021636		0,021636		3,778795
8	0,012292	0,012167	0,012324	0,012308	0,011838		0,011838		6,906403
16	0,008582	0,008566	0,008801	0,008999	0,00909		0,008566		9,544478
32	0,007101	0,006948	0,007099	0,007006	0,006908		0,006908		11,83526
40	0,013329	0,009528	0,012261	0,012112	0,012022		0,009528		8,580814

Figura 5.43: Swap 4000x4000

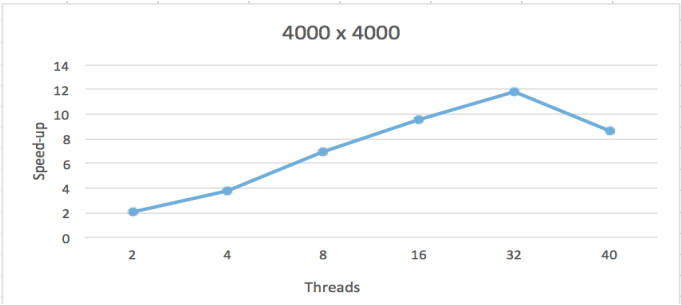


Figura 5.44: Curva Swap 4000x4000

10000 x 10000	1	2	3	4	5		mínimo		speed-up
tseq	0,541443	0,539813	0,54049	0,53955	0,541388		0,53955		-
2	0,276685	0,275741	0,276804	0,274425	0,143393		0,143393		3,762736
4	0,144176	0,144771	0,145253	0,146471	0,080928		0,080928		6,667037
8	0,080146	0,08012	0,081033	0,081562	0,053798		0,053798		10,02918
16	0,061693	0,057241	0,056665	0,056825	0,043514		0,043514		12,39946
32	0,043257	0,044295	0,044316	0,043788	0,04758		0,043257		12,47313
40	0,04736	0,048129	0,047461	0,045512	0,045512		0,045512		11,85512

Figura 5.45: Swap 10000x10000

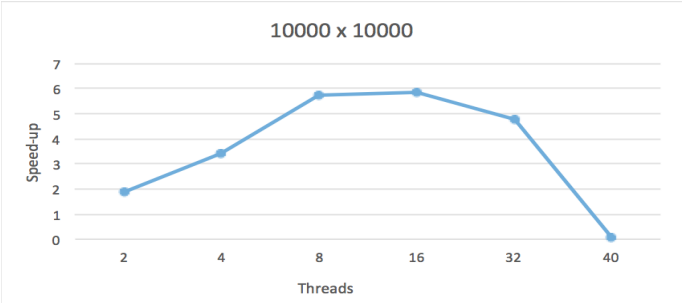


Figura 5.46: Curva Swap 10000x10000

40000 x 40000	1	2	3	4	5		mínimo		speed-up
tseq	9,333182	9,291082	9,466634	9,470009	9,233932		9,233932		-
2	4,770076	4,804926	4,794645	4,914237	4,882955		4,770076		1,935804
4	2,536431	2,575031	2,516776	2,557094	2,498865		2,498865		3,69525
8	1,274741	1,26445	1,257652	1,311893	1,239042		1,239042		7,452477
16	0,91692	0,957537	0,929487	0,960555	0,959919		0,91692		10,0706
32	0,765773	0,792458	0,751404	0,743211	0,74083		0,74083		12,46431
40	0,758012	0,744728	0,758173	0,768909	0,733091		0,733091		12,59589

Figura 5.47: Swap 40000x40000

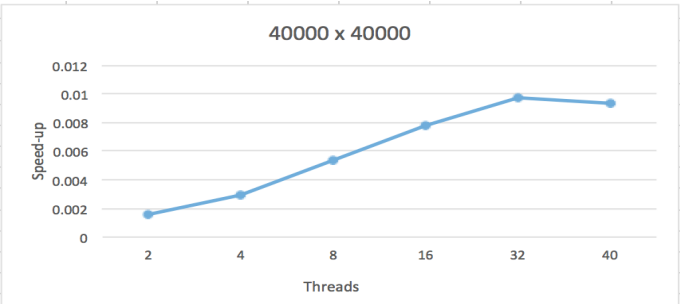


Figura 5.48: Curva Swap 40000x40000

5.2.3 Blocos

blocos de 32

32000 x 32000	1	2	3	4	5	mínimo	speed-up
tseq	6,898246	6,876871	6,809601	6,842952	6,825114	6,809601	-
2	6,941096	6,848298	5,975859	6,681348	5,920791	5,920791	1,150117
4	4,703413	4,608231	4,628515	4,140564	4,017165	4,017165	1,695126
8	4,03429	4,104365	3,918065	3,901131	3,171629	3,171629	2,147036
16	3,733396	3,619545	3,655396	3,521191	3,739708	3,521191	1,933891
32	0,027996	0,026543	0,025899	0,025704	0,026748	0,025704	264,9238
40	0,026456	0,026467	0,025858	0,034001	0,027758	0,025858	263,346

Figura 5.49: Blocos 32000x32000 blocos fixos de 32

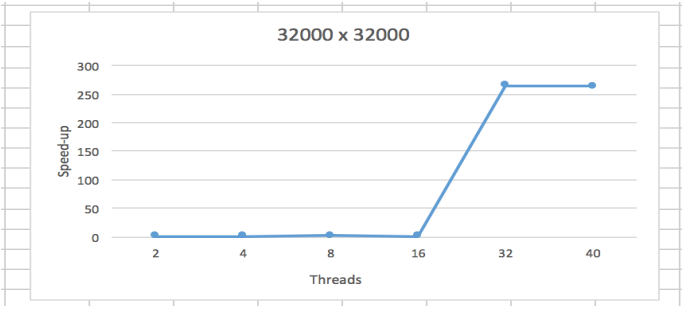


Figura 5.50: Curva Blocos 32000x32000 blocos fixos de 32

blocos 64							
32000 x 32000	1	2	3	4	5	mínimo	speed-up
tseq	8,787966	8,763285	8,743731	8,774719	8,771792	8,743731	-
2	6,686387	5,303971	4,766766	5,254705	5,391977	4,766766	1,834311
4	4,794232	5,485396	4,840877	4,651159	5,286394	4,651159	1,879904
8	3,989534	4,012246	4,047936	4,229695	3,967653	3,967653	2,203754
16	3,746488	2,981961	3,627495	3,641604	3,713959	2,981961	2,932208
32	0,026317	0,028448	0,025494	0,026589	0,027103	0,025494	342,9721
40	0,0266	0,028457	0,026497	0,02679	0,025788	0,025788	339,062

Figura 5.51: Blocos 32000x32000 blocos fixos de 64

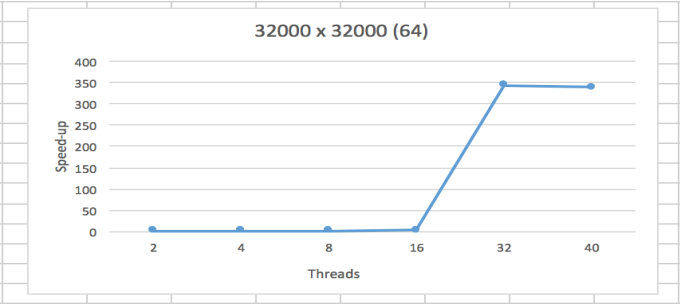


Figura 5.52: Curva Blocos 32000x32000 blocos fixos de 64

iterativo							
32000 x 320000	1	2	3	4	5	mínimo	speed-up
tseq	8,780356	8,753562	8,773315	8,747004	8,808312	8,747004	-
2	19,4046	16,78524	17,42021	18,91216	16,59485	16,59485	0,527091
4	12,35054	12,24011	12,31199	10,98145	11,02103	10,98145	0,796526
8	8,289189	8,385678	8,20122	8,569345	6,155877	6,155877	1,420919
16	4,651561	6,252863	6,974304	6,966238	6,917568	4,651561	1,880445
32	0,027996	0,026543	0,025899	0,025704	0,026748	0,025704	340,2974
40	0,026456	0,026467	0,025858	0,034001	0,027758	0,025858	338,2707

Figura 5.53: Blocos 32000x32000 blocos iterativos pelo numero de threads

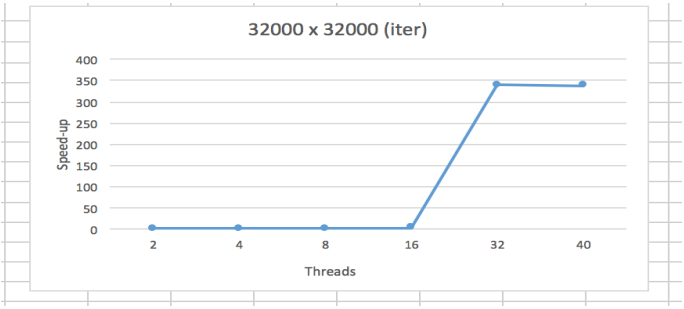


Figura 5.54: Curva Blocos 32000x32000 blocos iterativos pelo numero de threads

5.3 Medições 641

5.3.1 Iterativo

10000x10000	1	2	3	4	5	Mínimo	speed-up
tseq	0,826377	0,866344	1,021317	1,032262	0,572746	0,572746	-
2	0,445637	0,489332	0,452989	0,431316	0,436688	0,431316	1,32790344
4	0,322463	0,285202	0,307619	0,336762	0,311321	0,285202	2,008211724
8	0,274872	0,226243	0,231021			0,226243	2,531552357
16	0,142421	0,139097	0,13632			0,13632	4,201481808
32	0,118652	0,127433	0,119757			0,118652	4,827107845

Figura 5.55: Iterativo 10000x10000

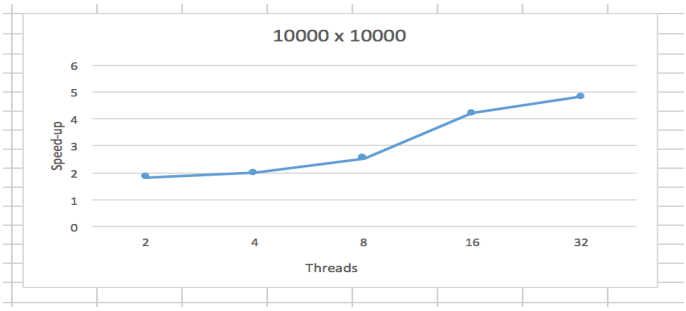


Figura 5.56: Curva Iterativo 10000x10000

	qsub -qmei -lnodes=1:r641:ppn=1,walltime=59:00							
20000x20000	1	2	3	4	5		Mínimo	speed-up
tseq	3,264848	3,299675	4,398335	3,754996	2,361898		2,361898	-
2	1,753987	2,072193	1,726798	1,678546	1,63758		1,63758	1,442309994
4	1,268196	1,12223	1,290685				1,12223	2,104646997
8	0,773399	0,776171	0,874142				0,773399	3,053919128
16	0,504107	0,533052	0,513126				0,504107	4,685310857
32	0,423153	0,443655	0,435676				0,423153	5,581664315

Figura 5.57: Iterativo 20000x20000

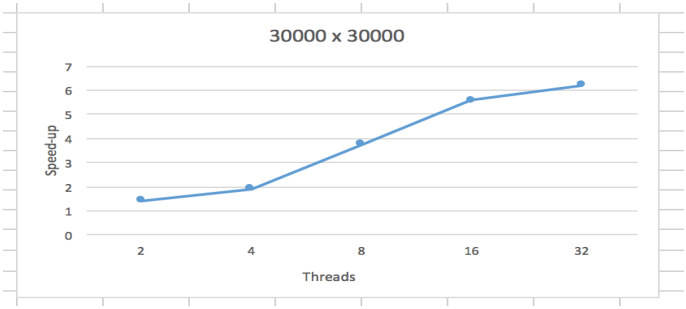


Figura 5.58: Curva Iterativo 20000x20000

qsub -qmei -lnodes=1:r641:ppn=1,walltime=59:00

30000x30000	1	2	3	4	5	Mínimo	speed-up
tseq	7,814763	7,80005	9,856986	9,484506	5,458602	5,458602	-
2	4,032117	3,901158	4,337273	3,885062	3,878072	3,878072	1,40755561
4	2,95578	2,870983	2,85286			2,85286	1,913378855
8	1,456919	1,870552	1,604134			1,456919	3,746675004
16	1,110969	1,196779	0,980971			0,980971	5,564488655
32	0,903944	0,879693	0,969599			0,879693	6,205121559

Figura 5.59: Iterativo 40000x40000

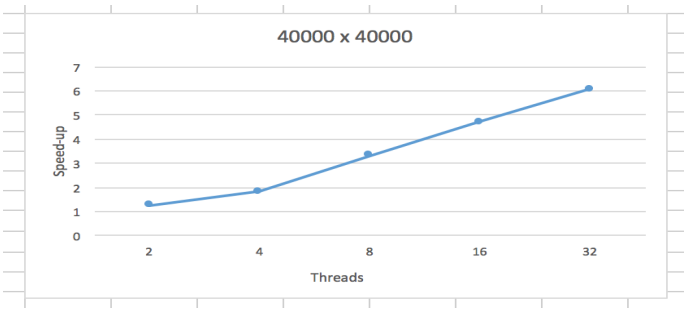


Figura 5.60: Curva Iterativo 40000x40000

qsub -qmei -lnodes=1:r641:ppn=1,walltime=59:00

50000 x 50000	1	2	3	4	5	Mínimo	speed-up
tseq	11,577514	11,205898	12,048756	10,283748	10,62183	10,283748	-
2	12,453517	12,928377	13,070923	12,805717	12,57173	12,453517	0,825770584
4	7,959623	7,962716	7,767402			7,767402	1,323962375
8	5,158196	5,879885	4,076869			4,076869	2,522462213
16	3,30347	3,10655	3,383603			3,10655	3,310343629
32	2,678764	2,465354	2,688462			2,465354	4,171306839

Figura 5.61: Iterativo 50000x50000

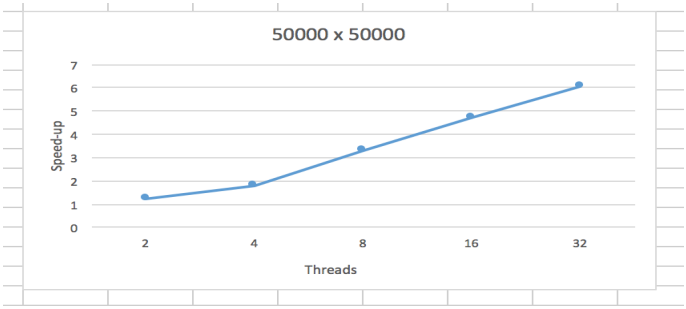


Figura 5.62: Curva Iterativo 50000x50000

	r641				
Size \ Threads	2	4	8	16	32
10000 x 10000	1,32790344	2,008211724	2,531552357	4,201481808	4,827107845
20000 x 20000	1,442309994	2,104646997	3,053919128	4,685310857	5,581664315
30000 x 30000	1,40755561	1,913378855	3,746675004	5,564488655	6,205121559
40000 x 40000	1,590331221	1,813280756	3,326972208	4,716672923	6,072185271
50000 x 50000	0,825770584	1,323962375	2,522462213	3,310343629	4,171306839

Figura 5.63: Geral 641 Iterativo

```
[a73236@login-0-0 PCPWork1]$ qsub -qmei -lnodes=1:r641:ppn=40,walltime=2:00:00 400x400ParallelSwapr
641.sh
qsub: Job exceeds queue resource limits MSG=cannot locate feasible nodes (nodes file is empty or al
l systems are busy)
[a73236@login-0-0 PCPWork1]$
```

Figura 5.64: Mensagem de erro do Nó r641

5.3.2 Outras abordagens abandonadas

Uma outra abordagem, menos intuitiva utilizando uma técnica designada por stencil. Esta pressupõe ilações que não nos são fornecidas tais como:

- Numero de fontes de calor
- Temperatura interior da matriz (se é ou não homogénea)

Por tais motivos esta foi posteriormente abandonada no caso de paralelismo pois a resolução deste problema se deve manter o mais geral possível e não sofrer com alguma possível interpretação do autor.
Pseudo Codigo Stencil

```
        //linhas superiores
        for(int i=1; i<N-1 && i <= iter+1; ++i){
            for(int j=1; j<M-1; ++j){
                //Codigo para o calculo da media numa c[U+FFFD]lula
            }

        //linhas inferiores
        for(int i=N-2-iter; i<N-1 ; ++i){ //i=N-1-(iter+1)
            for(int j=1; j<M-1; ++j){
                //Codigo para o calculo da media numa c[U+FFFD]lula
            }
        }

        //colunas lado direito
        for(int j=1; j<M-1 && j <= iter+1; ++j){
            for(int i=2+iter; i<N-2-iter; ++i){
                //Codigo para o calculo da media numa c[U+FFFD]lula
            }
        }

        //colunas lado esquerdo

        for(int j=M-2-iter; j<M-1; ++j){
            for(int i=2+iter; i<N-2-iter; ++i){
                //Codigo para o calculo da media numa c[U+FFFD]lula
            }
        }

... // copia G1 para G2
```

A resolução consistia em fazer um acerto no interior da matriz , isto é, reduzir proporcionalmente os elementos do centro a zero (tornando a matriz esparsa) e acompanhar esta redução também para as placas. Na última iteração iremos acertar proporcionalmente o decréscimo feito inicialmente no sistema

Esta análise ao problema permitia diminuir exponencialmente o numero de cálculos realizados imprimindo um ganho óbvio na performance do problema evitando cálculos desnecessárias visto que nesta abordagem a complexidade obtida estaria na ordem de

$$\theta = (N * (iter * 2) + ((N - iter) * 2) = N$$

o que comparativamente à abordagem tradicional bastante menor pois neste caso a complexidade de cálculos seria não ordem de

$$\theta = N^2$$

Outra opção ainda considerada foi tradução da matriz inicial, caso esta fosse uma matriz esparsa com as fontes de calor nas regiões de fronteira, para *CRS* (Compressed Sparse Row) que nos elimina os 0 da informação, e seria

útil até que o numero de iterações multiplicado pelas fontes de calor supere metade do valor da matriz, deixando esta de ser esparsa e aumentando o seu custo computacional associado a cada operação. A utilização deste tipo de representação costuma ser feita com o intuito de poupar espaço de memoria, porém, no nosso caso serviria também para restringir o esforço computacional como foi feito na operação anterior. Novamente devido à especificidade do problema e mesmo a sua complexidade na criação de um algoritmo universal e o constante re-size dos arrays por cada iteração entendemos que não valeria a pena abordar mais profundamente este caso, porem, compreendemos também que existiriam ganhos significativos comparativamente a uma abordagem mais clássica do problema caso este fosse relativo a "poucas"iterações e caso matriz estivesse num estado homogéneo excluindo as fontes de calor.

5.4 Código

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#include <string.h>
#include "papi.h"
#define NUM_EVENTS 2
#define min(a,b) ( ((a) < (b)) ? (a) : (b) )

double total;
int mode,ITER,N,M;
double **G1;
double **G2;
double tempo;
double wtime=0;
double clearcache [30000000];
long long values[NUM_EVENTS];
int Events[NUM_EVENTS]={PAPI_L1_TCM,PAPI_L2_TCM};
int EventSet;
void clearCache (void) {
   for (unsigned i = 0; i < 30000000; ++i)
      clearcache[i] = i;
}

void fillMatrix(){
    int i,j;
    #pragma omp parallel shared (G2) private (i,j)
    {
        //CIMA
        #pragma omp for
            for ( i = 0; i < M ; i++ ){
                G2[0][i] = 100;
                G1[0][i] = 100;
            }
        //BAIXO
        #pragma omp for
            for ( i = 0; i < M ; i++ ){
                G2[N-1][i] = 100;
                G1[N-1][i] = 100;
            }
        //ESQUERDA
        #pragma omp for
            for ( i = 1; i < N-1 ; i++ ){
                G2[i][0] = 100;
                G1[i][0] = 100;
            }
        //DIREITA
        #pragma omp for
            for ( i = 1; i < N -1 ; i++ ){
                G2[i][M-1] = 100;
                G1[i][M-1] = 100;
            }
        //RESTA
        #pragma omp for
            for ( i=1; i < N - 1 ; i++)
                for( j=1; j< M - 1 ; j++){
```

```
                G2[i][j] = 0;
            }
    }
}
void iterationSequentialCopIter(){
    int iter=0;
    while (iter < ITER){
        for(int i=1; i<N-1; i++){
            for(int j=1; j<M-1; j++){
                G1[i][j] = 0.2*(
                           G2[i-1][j]+
                           G2[i+1][j]+
                           G2[i][j-1]+
                           G2[i][j+1]+
                           G2[i][j]);
            }
        }
        ++iter;

        for(int i=1; i<N-1;++i)
            for(int j=1;j<M-1;++j)
                G2[i][j]=G1[i][j];

    }
}
void iterationSequentialCopSwap(){
    int iter=0;
    double** temp;
    while (iter < ITER){
        for(int i=1; i<N-1; i++){
            for(int j=1; j<M-1; j++){
                G1[i][j] = 0.2*(
                           G2[i-1][j]+
                           G2[i+1][j]+
                           G2[i][j-1]+
                           G2[i][j+1]+
                           G2[i][j]);
            }
        }
    ++iter;
    temp = G2;
    G2 = G1;
    G1 = temp;
    }
}
void iterationSequentialCopMem(){
    int iter=0;
    while (iter < ITER){
        for(int i=1; i<N-1; i++){
            for(int j=1; j<M-1; j++){
                G1[i][j] = 0.2*(
                           G2[i-1][j]+
                           G2[i+1][j]+
                           G2[i][j-1]+
                           G2[i][j+1]+
                           G2[i][j]);
            }
        }
        ++iter;
```

```
        memcpy(G2,G1,sizeof(double)*N*M);
    }
}

void swapBlocks(){
int nbx, bx, nby, by;
  nbx = omp_get_max_threads(); \\iterativo
  \\nbx=10000; \\blocos 32
  \\nbx=4000; \\blocos 64
  bx = M/nbx + ((M%nbx) ? 1 : 0);
  nby = nbx;
  by = N/nby + ((N%nby) ? 1 : 0);
  int iter=0;
  wtime = omp_get_wtime();
  int i,j,ii,jj;
  double** temp;
    while(iter<ITER){
        #pragma omp parallel for
        for (ii=0; ii<nbx; ii++)
            for (jj=0; jj<nby; jj++)
                for (i=1+ii*bx; i<=min((ii+1)*bx, M-2); i++)
                    for (j=1+jj*by; j<=min((jj+1)*by, N-2); j++) {
                        G1[i][j]=0.2*(
                           G2[i-1][j]+
                           G2[i+1][j]+
                           G2[i][j-1]+
                           G2[i][j+1]+
                           G2[i][j]);
            }

    temp = G2;
    G2 = G1;
    G1 = temp;
    ++iter;
  }

}

void iterationParallelSwapCpy(){
    int iter=0;
    double** temp;
    while (iter < ITER){
        wtime = omp_get_wtime ();
        #pragma omp parallel for
        for(int i=1; i<N-1; ++i){
            for(int j=1; j<M-1; ++j){
                G1[i][j] = 0.2*(
                    G2[i-1][j]+
                    G2[i+1][j]+
                    G2[i][j-1]+
                    G2[i][j+1]+
                    G2[i][j]);
            }
        }
    temp = G2;
    G2 = G1;
    G1 = temp;
    ++iter;
    }
```

```
}
void iterationParallel(){
    int iter=0;
    while (iter < ITER){
        wtime = omp_get_wtime ();
        #pragma omp parallel for
        for(int i=1; i<N-1; ++i){
            for(int j=1; j<M-1; ++j){
                G1[i][j] = 0.2*(
                    G2[i-1][j]+
                    G2[i+1][j]+
                    G2[i][j-1]+
                    G2[i][j+1]+
                    G2[i][j]);
            }
        }
            #pragma omp parallel for
            for(int i=1; i<N-1;++i){
                    for(int j=1;j<M-1;++j)
                        G2[i][j]=G1[i][j];
            }
    ++iter;
    }
}
void init(){
    G1 = (double **) malloc(N*sizeof(double));
    G2 = (double **) malloc(N*sizeof(double));
    for(int i = 0; i < N; i++){
        G1[i] = (double *) malloc(M*sizeof(double));
        G2[i] = (double *) malloc(M*sizeof(double));
    }
}
int main(int argc, char* argv []){
    if(argc>1){
        mode = atoi(argv[1]);
        ITER= atoi(argv[2]);
        N = atoi(argv[3]);
        M = atoi(argv[4]);
        init();
        clearCache();
        fillMatrix();
        //PAPI_library_init(PAPI_VER_CURRENT);
        //PAPI_create_eventset(&EventSet);
        //PAPI_add_events(EventSet,Events,NUM_EVENTS);
        if(mode == 1){
            tempo = omp_get_wtime ();
            //PAPI_start(EventSet);
            iterationSequentialCopIter();
            //PAPI_stop(EventSet,values);
            tempo = omp_get_wtime () - tempo;
            printf("Tempo SEQUENTIAL NORMAL: %lf \n",tempo);
            //printf("L1 Cache Misses %d",values[0]);
            //printf("L2 Cache Misses %d",values[1]);

        }
        else if (mode == 2){
            tempo = omp_get_wtime ();
            //PAPI_start(EventSet);
            iterationSequentialCopSwap();
```

```
            PAPI_stop(EventSet,values);
            //PAPI_stop_counters(values,NUM_EVENTS);
            //PAPI_stop(EventSet,values);
            tempo = omp_get_wtime () - tempo;
            printf("Tempo SEQUENTIAL W/ Swap: %lf \n",tempo);
            //printf("L1 Cache Misses %d",values[0]);
            //printf("L2 Cache Misses %d",values[1]);
        }

        else if (mode == 3){
            tempo = omp_get_wtime ();
            //PAPI_start(EventSet);
            iterationParallel();
            PAPI_stop(EventSet,values);
            //PAPI_stop_counters(values,NUM_EVENTS);
            //PAPI_stop(EventSet,values);
            tempo = omp_get_wtime () - tempo;
            printf("Tempo PARALLEL: %lf \n",tempo);
            //printf("L1 Cache Misses %d",values[0]);
            //printf("L2 Cache Misses %d",values[1]);
        }
        else if (mode == 4){
            tempo = omp_get_wtime ();
            //PAPI_start(EventSet);
            iterationParallelSwapCpy()
            PAPI_stop(EventSet,values);
            //PAPI_stop_counters(values,NUM_EVENTS);
            //PAPI_stop(EventSet,values);
            tempo = omp_get_wtime () - tempo;
            printf("Tempo PARALLEL W/ SWAP: %lf \n",tempo);
            //printf("L1 Cache Misses %d",values[0]);
            //printf("L2 Cache Misses %d",values[1]);
        }
        else if (mode == 5){
            tempo = omp_get_wtime ();
            //PAPI_start(EventSet);
            swapBlocks();
            PAPI_stop(EventSet,values);
            //PAPI_stop_counters(values,NUM_EVENTS);
            //PAPI_stop(EventSet,values);
            tempo = omp_get_wtime () - tempo;
            printf("Tempo SWAP BLOCKS: %lf \n",tempo);
            //printf("L1 Cache Misses %d",values[0]);
            //printf("L2 Cache Misses %d",values[1]);
        }

    }
return 0;
}
```