

UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

Algoritmos Paralelos
Stencil2D e All-Pairs Shortest Paths

João Teixeira (A85504)
José Filipe Ferreira (A83683)

June 17, 2021

Contents

1	Introdução	3
2	Stencil2D	4
3	All-Pairs Shortest Path	6
A	Medições Realizadas	7

Chapter 1

Introdução

O objetivo deste trabalho pratico é escolher dois algoritmos e analisar a sua escalabilidade. Os algoritmos escolhidos para este propósito foram o *All-Pairs Shortest Path* e o *Stencil2D*.

Ao longo deste relatório iremos descrever os passos tomados para fazer esta analise. Começando por descrever como foi desenvolvida a versão sequencial do algoritmo, em seguida descrevendo como foi criada a versão paralela e por fim comparando a performance das duas.

Todos os testes de performance foram realizados no cluster SeARCH da Universidade do Minho nas maquinas 641 com dual Intel Xeon E5-2650v2 a 2.60GHz e 64GiB de memória RAM. Para cada valor foram retiradas cinco medidas distintas e é apresentada a mediana dessas cinco medições. Uma tabela com todas as medições está disponível no Apêndice A.

Chapter 2

Stencil2D

O algoritmo Stencil2D é um algoritmo que consiste em criar múltiplas iterações sobre uma dada matriz. No início de cada iteração percorre-se cada ponto da matriz, excluindo os pontos na moldura exterior, e cria-se uma matriz nova em que cada ponto corresponde à media entre os pontos a norte, sul, este e oeste da matriz da iteração anterior (Imagem 2.1). Desta forma, cada iteração tem complexidade $O(n^2)$ sendo n o tamanho da matriz.

```
Begin
  for it := 0 to IT, do
    for i := 1 to N-1, do
      for j := 1 to N-1, do
        G1[i,j] = 0.2 * (G2[i-1,j] + G2[i+1,j] + G2[i,j-1] + G2[i,j+1] + G2[i,j]);
      done
    done
    copy(G1,G2)
  done
End
```

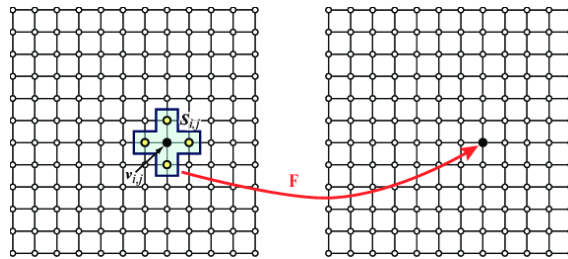


Figure 2.1: representação de uma iteração do algoritmo

Numa primeira versão da implementação deste algoritmo copiamos no fim de cada iteração a matriz onde se esta a escrever para uma matriz auxiliar que guarda a iteração anterior fazendo uso da função *memcpy*. Esta copia é extremamente lenta e pode ser otimizada.

Inspirados na forma como uma placa gráfica lida com buffers para renderizar imagens decidimos, em vez de copiar a matriz para uma matriz auxiliar, alternar a matriz onde se escreve e a matriz onde se lê em cada iteração. Desta forma evitam-se as copias de matriz e melhora-se a performance.

Com o objetivo de paralelizar esta última versão desenvolvida decidimos analisar o *vectorization report* criado pelo gcc. Desta forma constatamos que o loop que percorre ao longo de cada linha da matriz (o loop mais interior) vetoriza (Imagem 2.2) e, por isso, paralelizar este loop, em princípio, seria contraproducente. Por outro lado, o loop exterior referente às iterações tem dependência de dados entre cada uma. Desta forma resta o loop referente a percorrer todas as linhas para ser vetorizado.

```
stencil2D.c:27:34: optimized: loop vectorized using 32 byte vectors
stencil2D.c:27:34: optimized: loop versioned for vectorization because of possible aliasing
stencil2D.c:27:34: optimized: loop vectorized using 16 byte vectors
stencil2D.c:24:6: note: vectorized 1 loops in function.
```

Figure 2.2: excerto do Vectorization Report no GCC

Versão	OpenMP threads	tempo (s)	speedup
stencil2D sequencial	-	17.703	-
stencil2D sequencial sem copias	-	7.004	2.528
stencil2D paralelizado sem copias	16	4.39	4.033
	32	1.835	9.647

Table 2.1: tempos e speedup para stencil2D

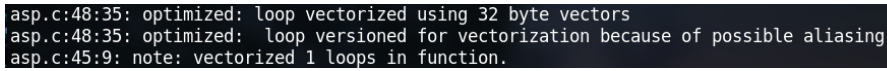
Chapter 3

All-Pairs Shortest Path

O algoritmo *All-Pairs Shortest Path* consiste em percorrer um grafo orientado pesado representado sobre a forma de uma matriz em que o valor de cada posição (i,j) representa o custo da aresta que vai do nodo i para o nodo j . No fim deste algoritmo correr, em cada posição (i,j) estará presente o custo mínimo de ir do nodo i para o nodo j .

```
Begin
  for k := 0 to N, do
    for i := 0 to N, do
      for j := 0 to N, do
        if cost[i,k] + cost[k,j] < cost[i,j], then
          cost[i,j] := cost[i,k] + cost[k,j]
        done
      done
    done
  done
End
```

Primeiro analisamos o código disponibilizado. Com base no *vectorization report* disponibilizado pelo gcc constatamos que o loop mais interior vetoriza quando se aplica a flag -O3 no compilador (Imagem 3.1). Por isso, paralelizar este loop seria contra produtivo.



```
asp.c:48:35: optimized: loop vectorized using 32 byte vectors
asp.c:48:35: optimized: loop versioned for vectorization because of possible aliasing
asp.c:45:9: note: vectorized 1 loops in function.
```

Figure 3.1: excerto do Vectorization Report do GCC

Com o objetivo de analisar qual dos dois loops restantes é o melhor para paralelizar foram criadas duas versões distintas da função. Analisando os resultados presentes na tabela 3.1 constatamos que é mais vantajoso paralelizar o loop exterior (neste caso o loop com a variável k).

Após analisarmos como a matriz é percorrida constatamos que a matriz é percorrida coluna por coluna. Desta forma as colisões na cache são maiores. Se trocarmos a ordem dos dois loops exteriores este problema fica resolvido. Para tentar extrair ainda mais performance decidimos alinhar a matriz, diminuindo assim ainda mais as colisões na cache do CPU.

```
int tab[N][N]__attribute__((aligned (32)));
```

Versão	OpenMP threads	tempo (s)	speedup
asp sequencial	-	6.825	-
asp paralelo i	16	1.695	4.027
	32	0.904	7.550
asp paralelo k	16	1.124	6.072
	32	0.638	10.697
asp swap paralelo alinhado	16	1.147	5.950
	32	0.596	11.451

Table 3.1: tempos e speedup para stencil2D

Appendix A

Medições Realizadas

Algoritmo	Versão	n threads	teste 1	teste 2	teste 3	teste 4	teste 5	mediana
Stencil2D	com copias	-	17.220	16.901	21.043	17.703	19.520	17.703
	sem copias	-	6.779	7.004	7.010	6.811	7.074	7.004
	paralelo sem copias	16	4.390	4.425	4.410	4.040	4.221	4.390
		32	1.842	1.778	1.835	1.854	1.787	1.835
ASP	sequencial	-	7.492	6.824	7.308	6.753	6.825	6.825
	paralelo k	16	0.972	1.133	0.939	1.124	1.291	1.124
		32	0.842	0.628	0.616	0.638	0.663	0.638
	paralelo i	16	1.733	1.636	1.695	2.190	1.660	1.695
		32	0.904	0.914	1.235	0.884	0.886	0.904
	swap paralelo alinhado	16	1.119	1.147	1.153	1.160	1.162	1.147
		32	0.574	0.595	0.598	0.664	0.594	0.595

Table A.1: Medições realizadas