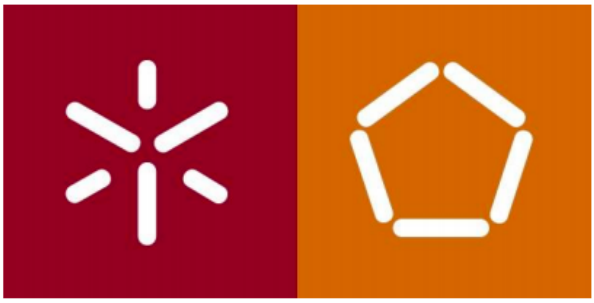


Universidade do Minho

Mestrado Integrado em Engenharia Informática

Paradigmas de Computação Paralela

Paralelização de um Algoritmo de Esqueletização de Imagens

Eduardo Lourenço da Conceição (A83870)
Rui Nuno Borges Cruz Oliveira (A83610)

06/12/2020
Braga

Índice

1 Introdução

Para o trabalho prático de Paradigmas de Computação Paralela, optamos por fazer uma implementação paralela de um algoritmo de esqueletização de imagens, utilizando imagens PGM, cuja especificação é de simples utilização. Assim, pretendemos estudar como paralelizar este algoritmo e explicar quais as maiores barreiras à compleção da tarefa que encontramos.

2 Versão Sequencial

O algoritmo que decidimos implementar foi o que nos foi exposto no enunciado. Como tal, de uma forma sucinta, o algoritmo vai percorrer a matriz de píxeis e altera o valor do píxel se as seguintes condições forem verdadeiras:

- o **número de vizinhos do ponto a preto** (N) deverá estar entre 2 e 6;
- o **número de transições de 0 para 1 nos vizinhos** (S) deverá ser 1;
- o ponto tem de respeitar uma de duas *checksums* que são expostas no enunciado e que variam entre as duas passagens.

Assim, são feitos dois varrimentos à matriz que são repetidos até não haver mais alterações à matriz.

Figure 1: Imagem *input* (esquerda) e seu respetivo esqueleto (direita)

Com base nisto, desenvolvemos duas versões sequenciais: uma que percorre a matriz linha a linha, coluna a coluna, dita **Versão Linha-a-Linha**, e outra que percorre as diagonais, chamada **Versão Diagonal**. Apesar desta versão ser mais complexa e mais lenta sequencialmente, apresentava uma melhor divisão dos dados, como vamos ver.

3 Paralelização

3.1 Partição do problema e dos dados a processar

Para começar a ver possíveis níveis de paralelização que podemos aproveitar, vamos analisar a **partição funcional e do domínio de dados** que podemos fazer para cada uma das versões paralelas que implementamos:

- **Versão Linha-a-linha**:
  - **Partição Funcional**: Como este está divido em dois ciclos *for*, podemos executar qualquer um deles em paralelo. Mais ainda, as funções que fazem verificações às condições podem ser executadas em paralelo, uma vez que **funcionalmente** são independentes umas das outras.
  - **Partição de Dados**: Os problemas com esta forma de percorrer a matriz são as **dependências de dados**. Como cada píxel depende dos seus oito vizinhos, não é possível dividir os dados nesta implementação sem haver *data races*, que precisam de ser controladas. No entanto, decidimos dividir os dados pelas linhas da matriz, que é uma forma simples de fazer a divisão e o resultado final, como vamos ver, é consistentemente válido, mas nem sempre é o mesmo.
- **Versão Diagonal**:
  - **Partição Funcional**: Semelhante à versão anterior, mas neste caso executamos os ciclos interiores em paralelo (ver *Anexo 1*). Há que ajustar as *loop variables* e *conditions* para se conformarem aos *standarts* do OpenMP.
  - **Partição de Dados**: ao contrário da versão anterior, se nós fizermos a divisão do domínio em diagonais separadas entre si por uma coluna, eliminando *data dependencies* e podemos analizar os elementos das diagonais em paralelo. Note-se que o *workload* não é tão equilibrado neste caso, pois as diagonais variam de tamanho.

3.2 Primeira Fase de Paralelização

Numa primeira fase de implementação, utilizamos apenas a versão Linha-a-linha, tentando manter a resultado correto ou obter *speed-ups* relevantes, mas não necessariamente os dois em conjunto. Para isto, fizemos três implementações:

- Uma em que paralelizamos a execução das linhas da matriz, sem nenhum controlo de dados, e uma segunda que segue a mesma lógica mas ao nível das colunas;
- Uma terceira em que utilizamos *tasks* do OpenMP 4.0 de modo a defininir explicitamente as dependências de dados que cada iteração dos ciclos tem. De notar que não conseguimos que esta implementação funcionasse no *cluster* do DI. Desta forma, não achamos que os resultados que obtivemos desta implementação deveriam ser tidos em conta, mas achamos por bem referi-la.

Para os dois primeiros casos, utilizamos apenas cláusulas *parallel for* ao nível dos ciclos que percorriam as linhas e a colunas, respetivamente. No segundo caso, criamos a *team* de *threads* fora do ciclo *do while*, o qual é percorrido apenas por uma *thread* com *single*. Depois, a cada iteração dos ciclos *for* que percorrem as colunas será atribuída uma *task*, cuja execução é dependente nos vizinhos do píxel analisado como *input* e no próprio píxel como *input* e *output*. Depois, para garantir que as *tasks* são executadas

pela ordem correta, temos uma barreira sob a forma de um *taskwait* no final da iteração (ver *Anexo 2*).

Tendo isto tudo em conta, os resultados obtidos podem ser encontrados no *Anexo 3*.

No que toca à versão de *tasks*, os resultados obtidos quando corridos fora do *cluster* não foram promissores, e os tempos de execução eram muito mais elevados que a versão sequencial devido, provavelmente, ao ***overhead* de sincronização que as dependências introduziam**, que é um problema que não é resolúvel com esta versão. Quanto às outras duas (paralelização por colunas e linhas), ambas têm limites devido à implementação sequencial que obriga a haver uma fração sequencial algo pesada no cálculo do S (**Lei de Amdahl**), sendo que estimamos que esta seja aproximadamente 11% do trabalho (assumindo para duas *threads* um *speedup* ótimo de 1.8, baseado nos resultados que podemos consultar no *Anexo 3.1*). De notar que a má primeira implementação do S levou a um ligeiro **memory hall** devido à necessidade de alocações de memória.

4 Segunda Fase de Paralelização

4.1 Versão Diagonal

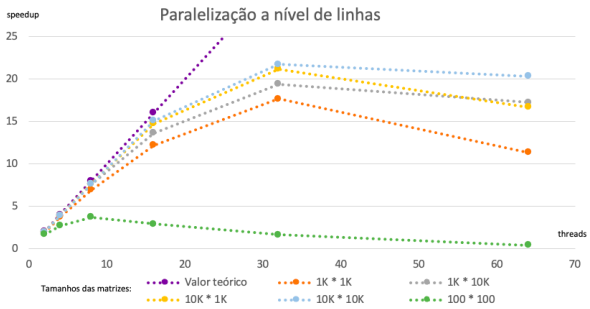
Ao implementar paralelização ao nível das diagonais, apesar de nos livrarmos do problema das dependências de dados e das *data races*, não obtivemos ganhos nenhuns (ver *Anexo 4*). Isto deve-se à existência de um ***workload* muito desequilibrado**, devido ao tamanho variável das diagonais, e também a um problema de *granularidade paralela excessiva*, pois temos obrigatoriamente de criar equipas de *threads* sempre que encontramos as regiões paralelizáveis, não podendo nós fazê-lo de outra forma pois criaria problemas ao nível de acessos por ordens inaceitáveis aos dados. Como tal, descartamos esta versão também.

Assim, decidimos melhorar a versão paralela ao nível das linha, visto ser a que apresentou melhores resultados.

4.2 Versão Linha-a-Linha

A segunda implementação da Versão Linha-a-Linha, quando comparada com a primeira, apresenta maioritariamente diferenças ao nível do cálculo dos três testes, que são mais amigáveis à paralelização. Optamos também por substituir todas as estruturas *if* possíveis por estruturas condicionais de máscara. Para além disto, para manter a consistência dos resultados, colocamos a *loop variable* j como privada às *threads*.

Os resultados obtidos foram os seguintes:



Aqui podemos ver que obtivemos resultados com *speedup* quase linear em relação ao número de *threads*, até 32 *threads*. Aqui é possível que o tempo da sincronização de *threads* e o *overhead* introduzido pela sua gestão comece a ser quase equivalente ao do trabalho computacional do algoritmo e a paralelização passa a deixar de apresentar tantos ganhos em termos de tempo. Tendo em conta que estamos a trabalhar com 8 processadores, é provável que a diferença que podemos notar a crescer entre o valor teórico e o real a partir das 16 *threads* seja devido ao facto que obrigamos cada *thread* física (a arquitetura *Ivy Bridge* com que estamos a trabalhar é *hyperthreaded*) trabalhe com mais do que uma *thread* de software.

Os resultados que esta implementação produz são **sempre válidos** (i.e. produzem um esqueleto válido para a imagem dada), mas **não são sempre iguais**. Isto porque há *data races* introduzidas ao ler os valores dos vizinhos de um dado píxel enquanto que outras duas *threads* acedem aos mesmos valores para fazer escritas.

A implementação desta versão encontra-se no ficheiro **tp1_par_v2_line.c**.

5 Justificação dos testes realizados

Primeiro há que ter noção que, para matrizes grandes, onde os *speedups* seriam notórios e relevantes, em vez de gerarmos imagens, criamos matrizes com valores aleatórios (0 ou 1) para processar. Isto retira os tempos de IO que seriam muito demorados e permite obter mais facilmente resultados que serão mais aptos a análise. Para mitigar diferenças de número de iterações necessárias para completar o algoritmo, colocamos um limite de 4 iterações do algoritmo, de modo a normalizar os resultados. Para testar se o algoritmo produzia resultados corretos, utilizamos o ficheiro **snake.pgm**.

Em todos os testes que realizamos, utilizamos valores para as matrizes que pudessem ser alojados na *cache L1* (100 por 100), na *cache L3* (1000 por 1000), e que não pudessem ser alojados na *cache* e, portanto, obrigassem a fazer acessos à DRAM (os restantes). Para isto, assumimos os seguintes valores, para o nodo computacional 652, com 8 *cores* (não conseguimos aceder ao nodo interativamente com um número maior nas alturas que tivemos disponíveis para testar)[1]:

- L1: 8 * 32KB (8-way set associative data cache)

- L2: 8 * 256KB (8-way set associative data caches, inclusive)
- L3: 256MB (20-way set associative shared cache, exclusive)

Isto permitia-nos ver de que forma os acessos à memória iriam impedir os ganhos, sendo que pudemos concluir que não afetaram de formas realmente significativas, uma vez que os maiores valores para as matrizes até acabaram por ser os com melhores ganhos, o que de certa forma faz sentido, pois são os valores que teriam mais a ganhar com paralelização em relação a uma execução sequencial.

Mais ainda, as matrizes não quadradas testadas serviram para experimentar conjuntos de dados de igual tamanho mas mais dispersos por linhas ou por colunas. Como seria de esperar, o valor com mais linhas que colunas teve resultados melhores na versão melhorada linha-a-linha, pois cada *thread* que trata das linhas tem menos dados a processar.

6 Conclusão

Em geral, ficamos contentes com o resultado final. Não só conseguimos obter bons *speedups*, como também conseguimos entender o porquê desses resultados. Para além disto, pudemos entender quais são alguns dos maiores entraves à paralelização, e tivemos a capacidade de lidar com uma boa parte deles. No entanto, ficamos um pouco desiludidos por não termos tido a capacidade de implementar uma versão paralela que lidasse com as dependências de dados, sendo que não produzimos resultados consistentemente iguais ao resultado da execução sequencial.

7 Referências

[1] *Intel Xeon E5-2670 v2 specifications*. URL: https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%5C%20E5-2670%5C%20v2.html.

8 Anexos

```
for(i = 1; i < linhas-1; i++)
{
  for(j = i, k = 1; j >= 1 && k < colunas-1; j--, k+=2)
  {
    testeB1(m, j, k, &changed);
  }

  for(j = i, k = 2; j >= 1 && k < colunas-1; j--, k+=2)
  {
    testeB1(m, j, k, &changed);
  }
}

for(j = 3; j < colunas-1; j++)
{
  for(k = j, i = linhas-2; k < colunas-1 && i > 1; k+=2, i--)
  {
    testeB1(m, i, k, &changed);
  }
}
```

Figure 2: Anexo 1: 1ª Passagem na Versão Onda (ciclos ideais para paralelizar destacados)

```
for(i = 1; i < linhas-1; i++)
{
  #pragma omp task shared(m) \
  depend(in: m[i-1][j-1], m[i-1][j], m[i-1][j+1],m[i][j+1], m[i][j-1], m[i+1][j-1], m[i+1][j], m[i+1][j+1]) \
  depend(inout: m[i][j])
  for(j = 1; j < colunas-1; j++)
  {
    //Teste ao pixel
    { …
  }
  #pragma omp taskwait
}
```

Figure 3: Anexo 2: Implementação com OpenMP4.0

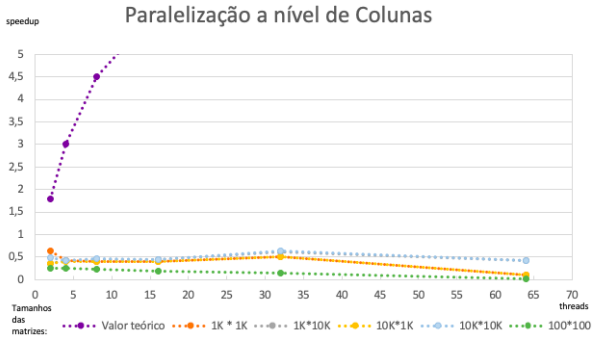


Figure 4: Anexo 3.1: Resultados de Testes com a primeira implementação (por linhas)

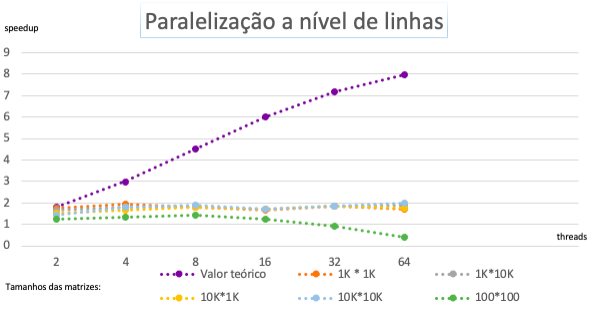


Figure 5: Anexo 3.2: Resultados de Testes com a primeira implementação (por colunas)

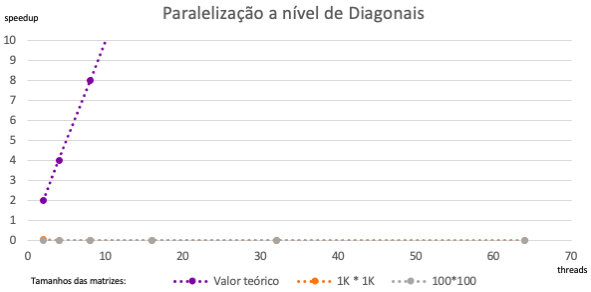


Figure 6: Anexo 4: Resultados de Testes com a implementação diagonal