Faculdade de Engenharia da Universidade do Porto

Computação Paralela e Distribuída

1º Projeto

-Diogo Miranda de Figueiredo Sarmento (up202109663@up.pt)

-Tomás Miguel Vieira da Câmara (up202108665@up.pt)

Porto, 16 de Março de 2024

1 Descrição do Problema

Na primeira parte do projeto vamos abordar o efeito do desempenho da hierarquia de memória quando são acessadas grandes quantidades de memória neste caso vamos testar implementando diferentes algoritmos de multiplicação de matrizes, na segunda parte do projeto vamos diferentes implementações de algoritmos que vão funcionar de maneira paralela.

Ao longo deste projeto utilizamos o PAPI (Performance API) quando possível para obter dados como os cache misses da cache de nivel L1 e L2, testamos também diferentes valores de maneira a melhor diferenciar a escalabilidade dos algoritmos em teste.

2 Algoritmos

Neste projeto vamos utilizar três algoritmos de multiplicação de matriz que não são paralelos e vão utilizar apenas um único core, mas, diferem na maneira que manipulam a memória que favorece os algoritmos que utilizam melhor o cache

Os Algoritmos são:

1. **Multiplicação simples de matriz**
2. **Multiplicação por linha de matriz**
3. **Multiplicação por bloco de matriz**

Para a multiplicação simples e de linha desenvolvemos código para python, desenvolvemos também multiplicação por linha e por bloco em c++, acabamos por escolher python porque é uma linguagem versátil e achamos que iria facilitar o desenvolvimento rápido dos algoritmos.

2.1 Multiplicação Simples de matriz

Para a multiplicação simples, fomos facilitados com uma versão funcional em c++,
que multiplica a primeira linha da primeira matrix pela primeira coluna da segunda matrix atribuindo assim o valor do primeiro numero da primeira coluna da matrix de resultado.

Com uma complexidade temporal de O(n^3) aqui está o algoritmo em pseudocódigo:

```
for(i=0; i<m_ar; i++)
      for( j=0; j<m_br; j++)
                  temp = 0;
                  for( k=0; k<m_ar; k++)
                            temp += pha[i*m_ar+k] * phb[k*m_br+j];
                  phc[i*m_ar+j]=temp;
```

2.2 Multiplicação por Linha de matriz

Este algoritmo vai multiplicar o primeiro número da primeira linha por todos os números da primeira coluna e assim adicionando progressivamente à matriz de resultado.
Este algoritmo tem a mesma complexidade temporal de O(n^3). Aqui está o pseudocódigo :

```
         for (int i=0; i<m_ar; i++)
                  for (int k=0; k<m_br; k++)
                           for (int j=0; j<m_ar; j++)
                                    phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
```

.

2.3 Multiplicação por Bloco de matriz

Para a multiplicação por blocos, implementamos uma versão semelhante à multiplicação por linha mas que antes divide em blocos menores a matriz, e calcula separadamente cada bloco adicionando os valores no fim.

A complexidade temporal da matriz acaba por ser O(n^3) e está aqui o pseudocódigo:

```
for (int i = 0; i < m_ar; i += bkSize)
  for (int j = 0; j < m_br; j += bkSize)
     for (int k = 0; k < m_ar; k += bkSize)
        for (int ii = i; ii < min(i + bkSize, m_ar); ii++)
          for (int kk = k; kk < min(k + bkSize, m_ar); kk++)
             for (int jj = j; jj < min(j + bkSize, m_br); jj++)
               phc[ii*m_ar + jj] += pha[ii*m_ar+kk] *    phb[kk*m_ar +jj];
```

2.4 Multiplicação paralela 1

Na primeira implementação de multiplicação paralela vamos implementar a multiplicação por linha e começamos por paralelizar o for loop mais externo com pragma `#pragma omp parallel for` que assim vai paralelizar a multiplicação linha a linha alocando assim uma linha para multiplicar por cada thread.

```
         #pragma omp parallel
         for (int i=0; i<m_ar; i++)
              for (int k=0; k<m_br; k++)
                     for (int j=0; j<m_ar; j++)
                            phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
```

2.5 Multiplicação paralela 2

Na segunda implementação de um algoritmo paralelo implementamos também o algoritmo de multiplicação por linha mas paralelizando o for loop mais interno este é o responsável por multiplicar um certo valor de uma linha por um conjunto de valores de uma coluna.

```
#pragma omp parallel
for (int i=0; i<m_ar; i++)
         for (int k=0; k<m_br; k++)
                  #pragma omp for
                  for (int j=0; j<m_ar; j++)
                                    phc[i*m_ar+j] += pha[i*m_ar+k] * phb[k*m_br+j];
```

3 Métricas de desempenho

Para avaliar o desempenho dos diferentes algoritmos medimos o tempo que demora a obter o resultado ambos em python e c++, no entanto apenas utilizamos o PAPI em c++ através do PAPI conseguimos obter não só o tempo mas o número de cache misses tanto no L1 como no L2, uma vez que vamos fazer

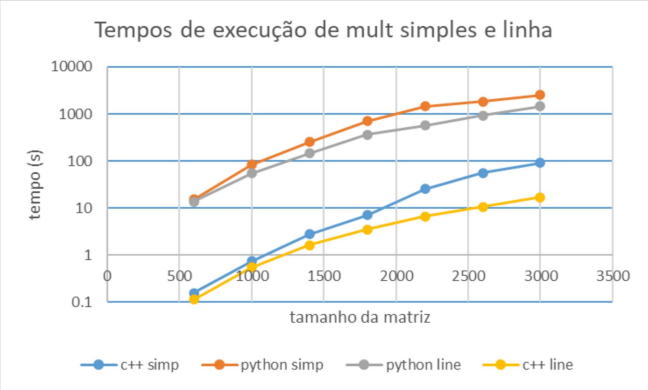
operações que vão aceder a quantidades grandes de memória uma cache miss vai atrasar essa operação significativamente assim vai ser importante analisar os valores.

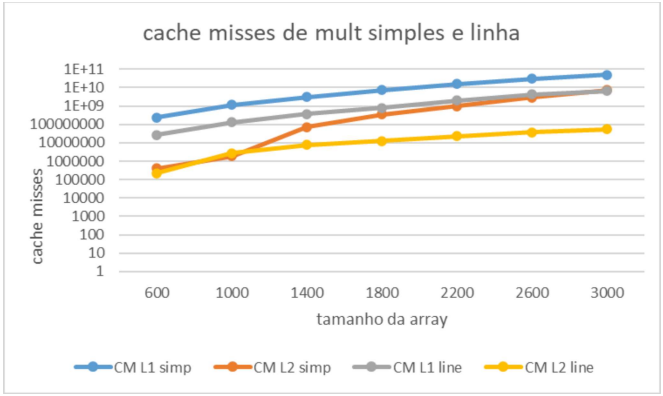
Para garantir resultados consistentes, o mesmo computador foi utilizado para todos os testes.

O computador utilizando rodava com o sistema operativo Windows 11 (Version 10.0.22631 Build 22631), com um Ryzen 5 5600U single clocked at 2.3 GHz going up to 4.2 GHz, com por core, com 64 KB de L1 cache, com 512 KB de L2 cache, e 16MB de L3 partilhada. Em todos os testes de c++ os programas foram compilados com a flag de optimização -O2 em adição à do papi -lpapi, nos testes de paralelização adicionamos ainda a flag -fopenmp.

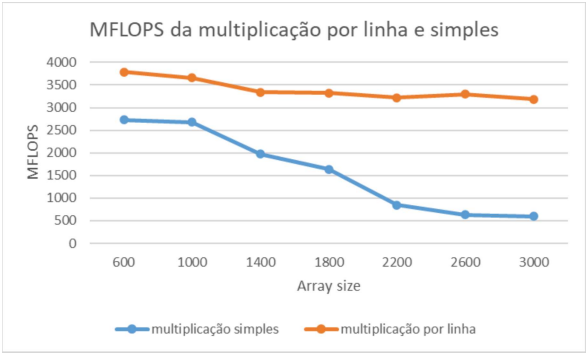
4.1 Análise e comparação entre a multiplicação Simples e por linha

Nesta comparação a escala dos valores de tempo de execução vão estar na escala logarítmica para facilitar a visibilidade dos resultados.





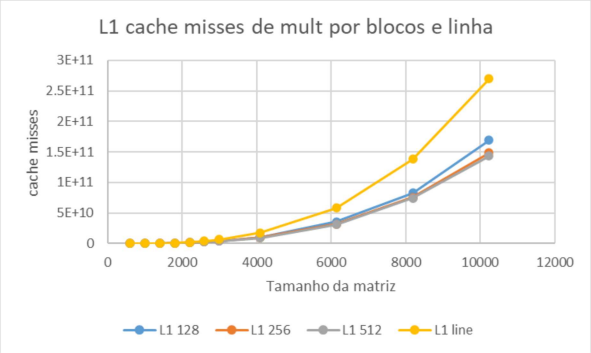
Comparando os tempos de execução dos mesmos algoritmos em python e em c++ facilmente conseguimos concluir que os valores diferem colossalmente, isto causado porque as estrutura de dados que usamos em python (listas) são muito mal otimizadas em comparação com as de c++.

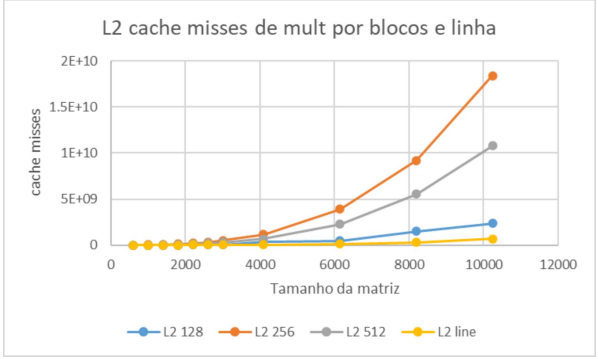


Agora comparando os tempos de execução conseguimos concluir que a multiplicação por linha é significativamente mais rápida e observando os cache misses podemos concluir que o número de cache misses da mesma também é menor assim podemos dizer com confiança que a melhor gestão de memória da multiplicação por linha consegue manter melhor os valores utilizados na cache assim evitando cache misses, pois vai utilizar o ciclo intermédio para iterar linha a linha da primeira matriz e multiplicando pelo correspondente na segunda matriz assim aumentando a disponibilidade dos valores da linha iterada em cache assim diminuindo os cache misses.

Agora analisando a tabela dos valores dos MFLOPS podemos mais uma vez reforçar que o maior número de cache misses vai penalizar o desempenho do algoritmo ao ter de ir buscar os valores necessários para calcular a multiplicação a memórias mais lentas e aumentando o tempo de acesso à memória.Assim o algoritmo de multiplicação por linha consegue manter um valor de Mflops maior constantemente em relação à multiplicação simples.

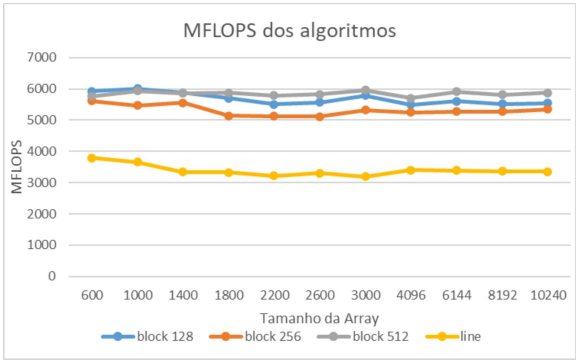
4.2 Análise e comparação dos resultados da multiplicação por bloco com diferentes tamanhos de bloco e multiplicação por linha

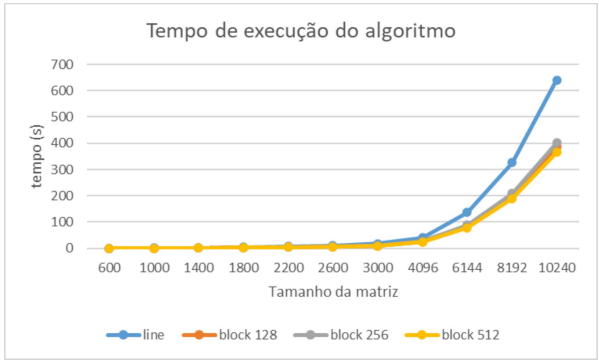




Conseguimos concluir que a multiplicação por bloco consegue um número inferior de cache misses no cache do nível L1 comparando à multiplicação por linha, no entanto é ao nível do cache L2 que a multiplicação por linha teve menos cache misses, mas a penalidade do maior número de cache misses no nivel L1 penalizou mais o algoritmo de linha tornando o de bloco mais rápido,é importante referir que os valores de bloco não afetaram o tempo de execução muito significantemente acabando por ser o bloco 512 com melhor tempo e por esse motivo teve também o maior MFLOPS dos algoritmos testados .

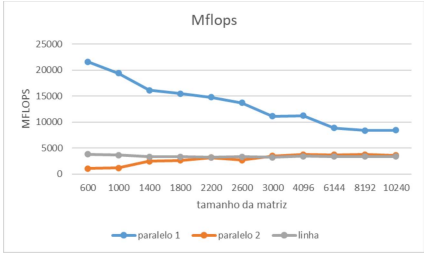
Relativamente ao valores de cache misses do block de 128 ele não utiliza na totalidade a cache L2 (512KB) e assim acaba penalizada nos resultados de tempo por esta razão não faria sentido testar com por exemplo 64x64 e seguindo um raciocínio semelhante não faria sentido testar com uma matriz maior que 512x512 pois ela já utiliza na totalidade a cache L2 e aumentar a matriz apenas iria deixar o algoritmo mais dependente da cache L3 que é partilhada e eventualmente ser penalizado a aceder à memória RAM.



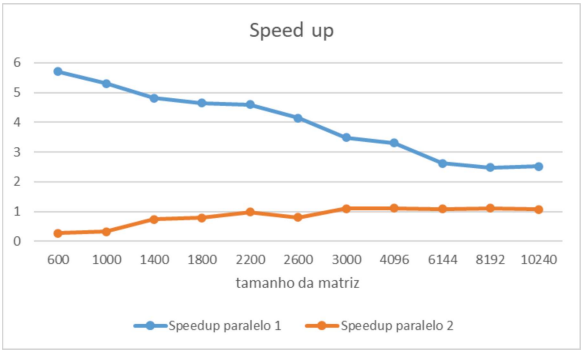


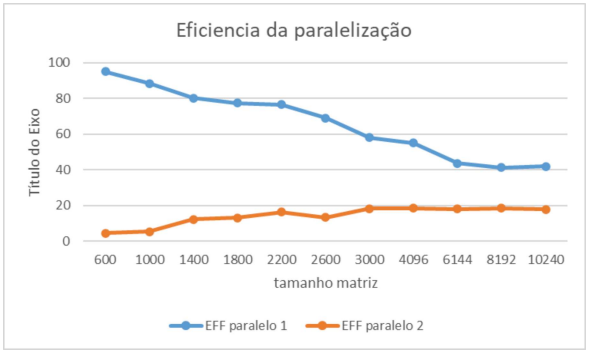
4.3 Análise e comparação dos resultados da multiplicação dos algoritmos paralelizados

Como observamos nos gráficos o algoritmo paralelo 2 acaba por ter um tempo de execução semelhante ao algoritmo original sem paralelização conseguindo aumentar no fim pois ele paraleliza as multiplicações singulares de valores somando posteriormente ao valor na matriz solução, No entanto o paralelo



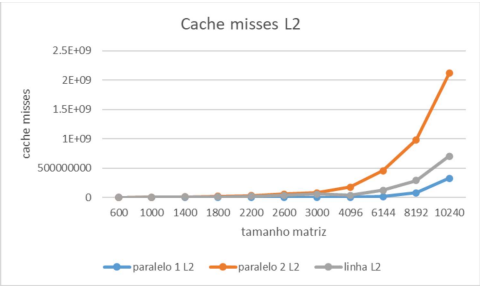
1 apresenta valores de MFLOPS muito mais altos do que ambos os algoritmos, diminuindo com o tamanho uma vez que o cache ficando mais cheio e os benefícios de paralelizar as linhas se torna menos viável uma vez que começam a encher o cache o algoritmo paralelo 2 não tem esse problema daí a ir aumentando com o tamanho.

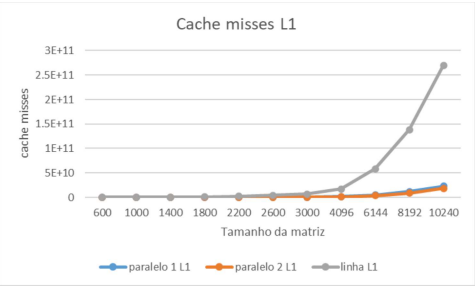




Como referi acima a Eficiência do algoritmo paralelo 1 vai diminuindo com o tamanho da matriz uma vez que ao encher o cache vai ter mais cache misses e atrasos que vai provocar a queda da eficiência e do speed up enquanto que o paralelo 2 acaba por se desempenhar um pouco com o aumento da matriz melhor, o que traduz uma melhor eficiência.

O segundo algoritmo,desperdiça desempenho na medida que os valores de j estão mais próximos e quando os valores são inicialmente carregados para cache é carregada uma linha com os valores adjacentes e





paralelizando o loop j dessa maneira vai carregar dados que inerentemente estão a ser processados noutra thread o que vai causar overhead, já no caso do primeiro onde para além da escala da paralelização ser maior ou seja cada thread ter tarefas maiores, aproveita ainda desta característica conseguindo ter ainda menos cache misses no nível L2. Assim, ambos os paralelos têm menores valores de cache misses no nível de L1 comparando com o original, mas o paralelo 2 têm valores muito mais altos de cache misses na L2 enquanto que o paralelo tem menos que ambos.

5 Conclusão

Em conclusão, a realização deste trabalho permitiu-nos aprofundar nos tópicos de paralelização de tarefas e de gestão de memória assim aumentando a eficiência dos programas. Aprendemos também a melhorar programas sequências com algumas alterações de maneira a gerir melhor a memória cache assim evitando os longos períodos de overhead quando é necessário buscar informações fora da cache L1 e L2.