Bubble Sort - Otimização e teste de escalabilidade em ambiente de memória partilhada

João Miguel Afonso - A71874

***Resumo*—O Bubble Sort é um algoritmo de ordenação consideravelmente ineficiente e de implementação simples. Contudo, a sua implementação paralela não é tão trivial quanto à partida possa parecer. Para tirar partido do elevado grau de escalabilidade inerente aos algoritmos quadráticos, foi necessário dividir os dados em várias porções, processando o vetor num sistema de *pipeline*. Para além da versão sequêncial, foram estudadas quatro versões paralelas, correspondentes às implementações em *OpenMP* e *Posix Threads*, com espera ativa ou *mutex*.**

Keywords*—*OpenMP, Posix Threads, Paralelismo, Concorrência, Desempenho.

I. O ALGORITMO

Para ordenar um vetor de N elementos, o algoritmo Bubble Sort percorre-o N vezes, trocando cada par de elementos adjacentes que não satisfaçam a condição inicial (ordem crescente ou decrescente). Assim, cada iteração garante que a última posição do vetor processado contém o seu maior/menor elemento, não necessitando portanto de ser utilizado na iteração seguinte. Com o intuito de melhor clarificar o algoritmo apresentado, apresenta-se o seguinte pseudocódigo, correspondente a uma possível implementação:

```
BubbleSort( A : vetor ordenável ):
    n = comprimento(A)
    enquanto n>=2:
        para i de 0 a n-1:
            se A[i] > A[i+1] então:
                troca A[i] com A[i+1]
```

Como se pode verificar, este percorre todas as iterações mesmo que o vetor inicial já esteja ordenado. Sabe-se que é possível obter ganhos significativos apenas parando a execução do algoritmo a partir do momento em que o vetor esteja ordenado, isto é, quando se processa uma iteração completa sem efetuar qualquer troca. Porém, o propósito deste projeto centra-se maioritariamente na exploração de paralelismo e não tanto na otimização da sua versão sequencial, sendo portanto utilizada a versão apresentada.

Deste modo, uma vez que efetuamos N-1 iterações, cada uma delas comparando 1 até N elementos, podemos concluir que o custo global do algoritmo é de $\theta(\frac{N(N-1)}{2})$ comparações, sendo este de ordem quadrática. Por sua vez, o número de trocas efetuadas é apenas dependente dos valores contidos no vetor inicial, podendo variar entre 0 e $\frac{N(N-1)}{2}$.

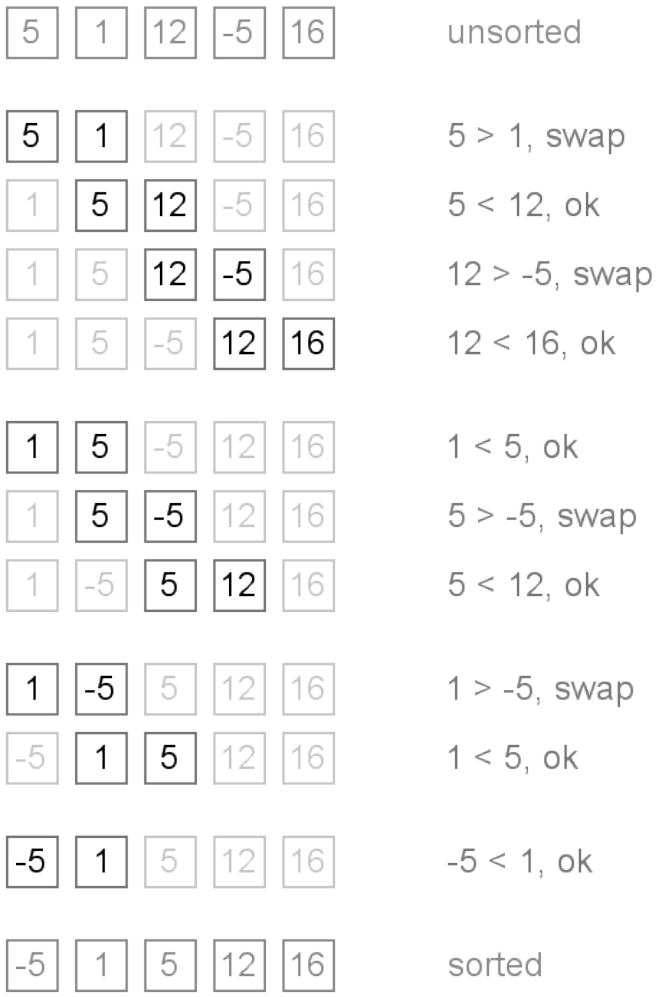


Figura 1. Exemplo de execução do algoritmo [1]

A Figura B representa uma completa execução do algoritmo, recebendo como entrada o vetor $[5, 1, 12, -5, 16]$. Tal como referido, são efetuadas $\frac{N(N-1)}{2} = \frac{5*(5-1)}{2} = 10$ comparações.

Para poder estabelecer um termo de comparação entre todas as implementações do algoritmo, foram feitas várias medições dos tempos de execução sequenciais, de acordo com as condições estabelecidas na secção III.

Como podemos ver na Figura 2, o ruído do graficoo é bastante reduzido, fazendo com que a curva pareça constante quando se representam os dados numa escala superior.

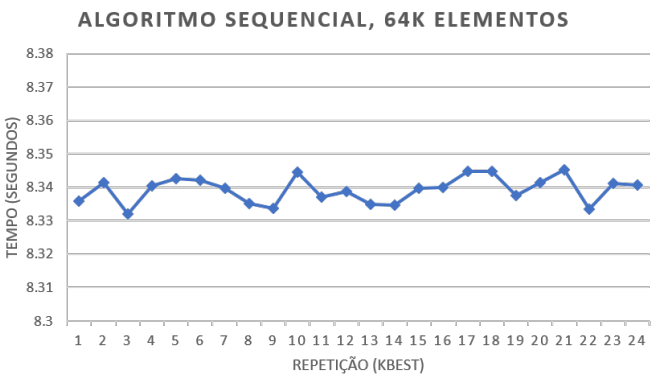


Figura 2. Tempo de execução, algoritmo sequencial, 64k elementos

II. Versão paralela

Existem várias formas de paralelizar o algoritmo, cada uma delas com diferentes pormenores passiveis de ser analisados. Seguem-se alguns exemplos, bem como a respetiva caracterização e teste de escalabilidade.

A. Odd Even

Esta implementação do algoritmo passa por dividir cada iteração em duas, de forma a permitir que fossem processados vários pares de elementos em paralelo. Assim, a primeira iteração trataria de processar os elementos das posições ímpares, fazendo a troca com a posição seguinte caso necessário, ficando a segunda iteração responsável por repetir o processo para as posições pares, tudo isto de acordo com o seguinte pseudocódigo:

```
BubbleSort( A : vetor ordenável ):

    enquanto A não está ordenado:

        #ciclo processado em paralelo
        para cada posição ímpar de A:
            se A[i] > A[i+1] então:
                troca A[i] com A[i+1]

        #ciclo processado em paralelo
        para cada posição par de A:
            se A[i] > A[i+1] então:
                troca A[i] com A[i+1]
```

Uma vez que a alternância entre posições pares e ímpares permite evitar a implementação de um mecanismo de controlo de concorrência, o algoritmo resultante é bastante simples.

Como se pode observar, a implementação pode ainda ser melhorada, por exemplo criando a região paralela no inicio do algoritmo. Contudo, esta alteração implicaria que fossem colocadas barreiras entre as execuções dos ciclos internos (execuções pares e ímpares), acabando por fazer com que o algoritmo não escale devidamente.

Neste sentido, uma vez que para conjuntos de dados de dimensões reduzidas o peso de criação das *threads* ultrapassa largamente o tempo útil do algoritmo, este facilmente se comporta pior do que a versão sequencial. Deste modo, conjugando uma paralelização relativamente trivial com uma implementação simples, não foram conseguidos ganhos superiores a dois, tal como demonstrado na Figura 3.

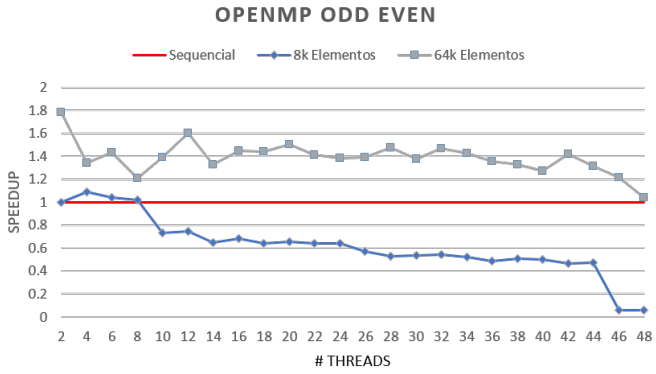


Figura 3. Ganhos da versão odd even em relação ao algoritmo sequencial

B. Paralelização por Blocos

Tal como explicado anteriormente, o desafio inerente à paralelização do Bubble Sort está diretamente relacionado com a eficiência da gestão de concorrência entre as várias *threads*. Com o seguinte pseudocódigo procura-se representar uma solução genérica, que vai derivar nas quatro últimas versões implementadas.

```
BubbleSort( A: vetor ordenável ):
    n = comprimento(A)
    enquanto n>1:
        bloqueia bloco 0
        n <- n-1
        tamanho <- n
        para cada bloco b (até tamanho):
            para cada índice i do bloco:
                se A[i] > A[i+1] então:
                    troca A[i] com A[i+1]
            se não é o último bloco:
                bloqueia bloco b+1
                se A[i] > A[i+1] então:
                    troca A[i] com A[i+1]
                liberta bloco b
            senão:
                liberta bloco b
                parar ciclo
```

Tal como apresentado, as próximas soluções para o problema passam por dividir o vetor inicial de dados em diversos conjuntos (*chunks*). Assim, as várias iterações vão ser divididas pelas várias *threads*. Cada uma delas, para executar o algoritmo sobre cada bloco, implementa um mecanismo de *lock*, de forma a que nenhuma das outras possa ler ou escrever naquela região dos dados. No que diz respeito às fronteiras entre blocos, estas vão ser tratadas adquirindo o *lock* de ambos os blocos, de forma a garantir que nem a *thread* que está a tratar o bloco anterior nem a que está no bloco seguinte trabalhem sobre os mesmos dados que a que está a computar a fronteira.

1) ***OpenMP Espera Ativa****:* A primeira variante do algoritmo apresentado foi realizada recorrendo ao OpenMP. Nesta versão não foi ainda utilizado qualquer mecanismo de exclusão mútua intrínseco ao sistema operativo, tendo sido implementado um sistema de *Round Robin* com espera ativa.

No inicio da execução do algoritmo, o *lock* de cada um dos blocos do vetor vai apenas poder ser adquirido pela *thread* 0. Todas as outras vão ficar presas num ciclo. Quando a *thread* 0 (ou outras em fase posterior) libertam o *lock*, o valor da variável que representa qual das threads vais ser executada é incrementado em uma unidade. Assim, sabemos que as threads vão executar sempre de forma ordenada. Posto isto, foram feitos vários testes aos tempos de execução, produzindo os seguintes resultados:

Uma vez que estamos à espera da execução ordenada das *threads*, o recurso à diretiva *ordered* poderia ser de especial interesse para a implementação em questão. No entanto, não foi utilizado a tempo de ser incluído no presente relatório. Mesmo assim, como podemos observar na Figura 4, foram conseguidos ganhos perto das vinte vezes, para o vetor de maior dimensão.

Comparando os gráficos obtidos, podemos ver a influência do tamanho do vetor nos ganhos obtidos, uma vez que no primeiro o esforço computacional necessário à criação das várias *threads* tende a anular os ganhos conseguidos por paralelização, resultando em tempos praticamente constantes, o que já não acontece no vetor de tamanho superior.

2) ***Posix Threads Espera Ativa****:* Mantendo o mecanismo de espera ativa utilizado na versão anterior, procurou-se agora implementar o algoritmo recorrendo ao Posix Threads.

Assim, as poucas alterações efectuadas passam por criar manualmente as várias *threads*, juntando-as no final da execução, bem como definir um *kernel* que cada uma delas deva executar. Existem aqui duas possibilidades, na mediada em que podemos ou não fazer com que a *thread* principal tenha um papel ativo no processo de ordenação. Porém, verificou-se que as duas versões têm desempenhos idênticos, tendo-se optado por fazer com que todas as *threads* computassem a função de ordenação. Os resultados obtidos apresentam-se em seguida:

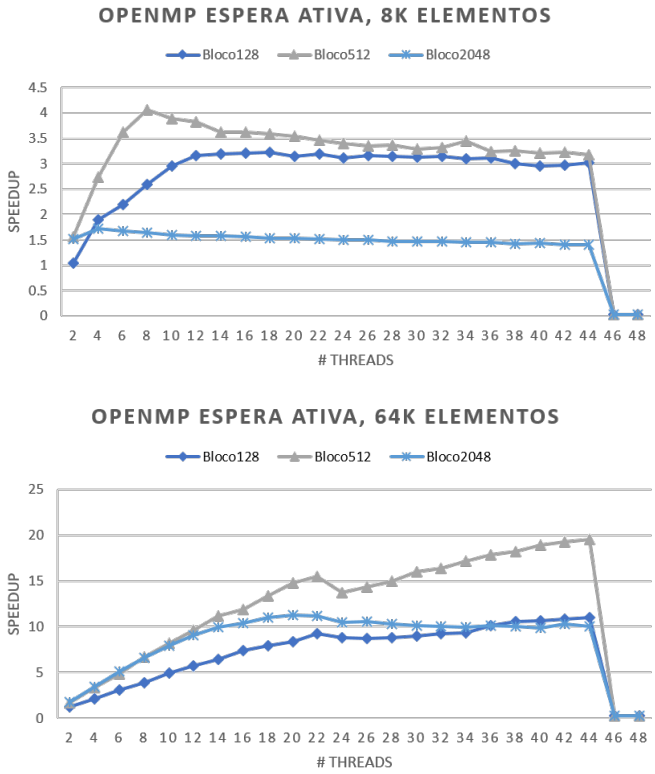


Figura 4. Ganhos da paralelização por blocos com OpenMP e espera ativa

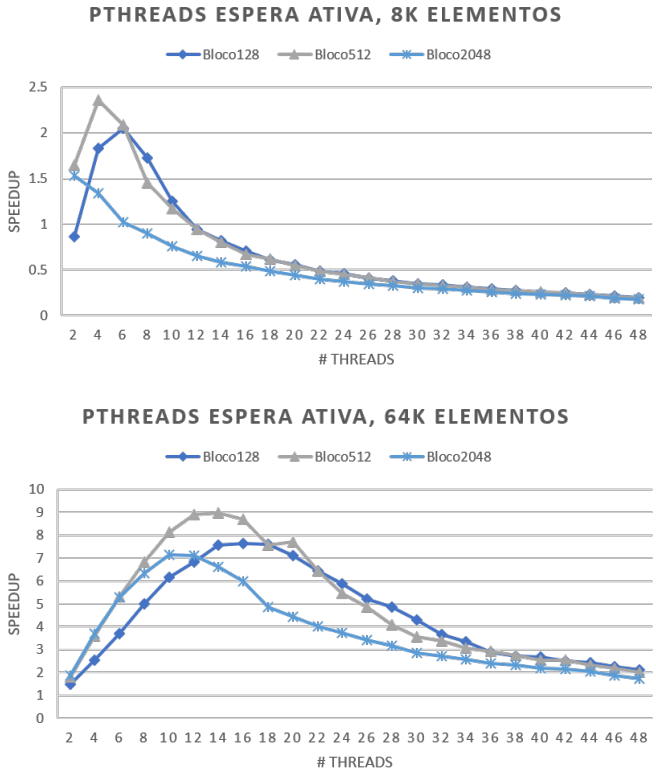


Figura 5. Ganhos da paralelização por blocos com Pthreads e espera ativa

Como podemos observar na Figura 5, não há muita informação que se possa retirar da curva de ganhos, pois ao contrário de outras versões, esta segue um comportamento padrão, escalando para um baixo número de *threads* (de acordo com o tamanho dos dados) e começando depois a perder performance à medida que estas aumentam.

No entanto, esta implementação vai ser relevante na media em que permite dar uma ideia da capacidade de paralelização do OpenMP, o qual também é traduzido para Pthreads, comparando os tempos anteriores com os obtidos com a implementação manual [Ver Secção II-B5].

3) ***OpenMP Mutex****:* O objetivo da implementação desta versão é perceber de que forma o algoritmo é influenciado quando se substituiu o mecanismo de espera ativa pelas implementações de exclusão mútua do OpenMP. Para tal, o vetor auxiliar que continha a identificação das *threads* que possuiam o lock de cada bloco passa a conter variáveis do tipo *omp_lock_t*, sobre as quais podem ser aplicadas as operações *omp_set_lock* e *omp_unset_lock*.

A principal diferença quando comparada com as implementações anteriores é que deixa de ser necessária a existência de um escalonamento circular, passando a ser aleatória a *thread* que adquire o primeiro *lock*, mantendo-se no entanto, a ordem das *threads* ao longo do resto da iteração.

Os ganhos obtidos estão representados no gráfico seguinte:

A análise da Figura 6 volta a confirmar que o algoritmo escala melhor para *inputs* de dimensões superiores, sendo conseguidos ganhos na ordem das dezoito vezes.

No entanto, observa-se novamente uma quebra anormal de performance das 22 para as 24 e das 44 para as 46 *threads*. Foram levantadas várias hipóteses para explicar esta variação. A primeira prende-se ao facto que que quando se executa o algoritmo com blocos de tamanho fixo, há uma fase da execução em que passam a existir mais *threads* do que blocos para computar. Isto provoca um grande desbalanceamento de carga, que pode fazer com que o ganho conseguido não seja suficiente para cobrir o custo de manutenção das *threads*.

A outra hipótese está mais ligada às maquinas onde foram feitos os testes. Pensa-se que ao gerar o vetor, este fique guardado apenas na cache L3 do *socket* onde executa a *thread* principal. Assim, uma vez que estamos perante uma arquitetura NUMA, quando as *theads* que executam no outro *socket* tentarem aceder aos dados, vão ter de usar um barramento específico, que pode estar a ser saturado a partir das 22 *threads*.

4) ***Posix Threads Mutex****:* A última versão implementada corresponde à fusão das novidades introduzidas nas duas versões anteriores, sendo elas a exploração do Pthreads e a utilização de exclusão mútua. Como tal, esta versão foi implementada para garantir que nenhum comportamento inesperado pudesse acontecer. Obtiveram-se os seguintes resultados:

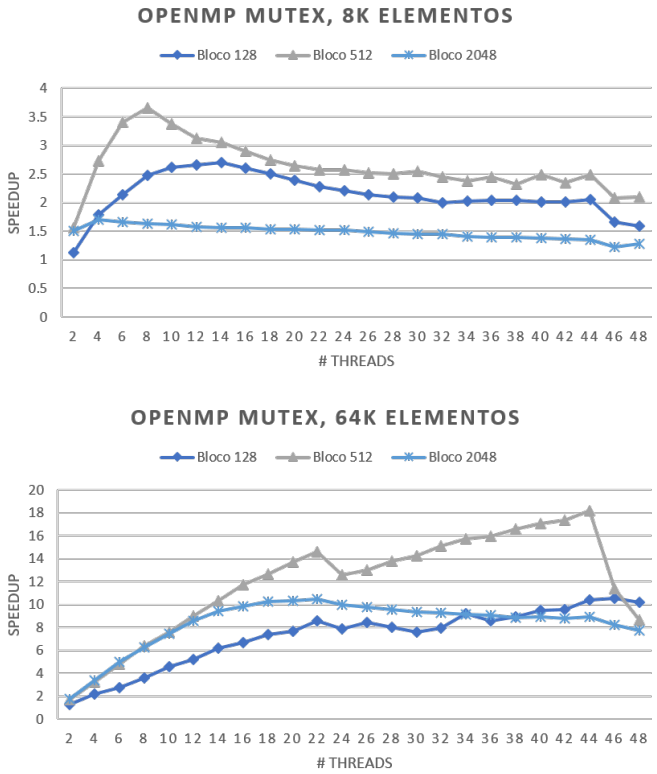


Figura 6. Ganhos da paralelização por blocos com OpenMP e Mutex

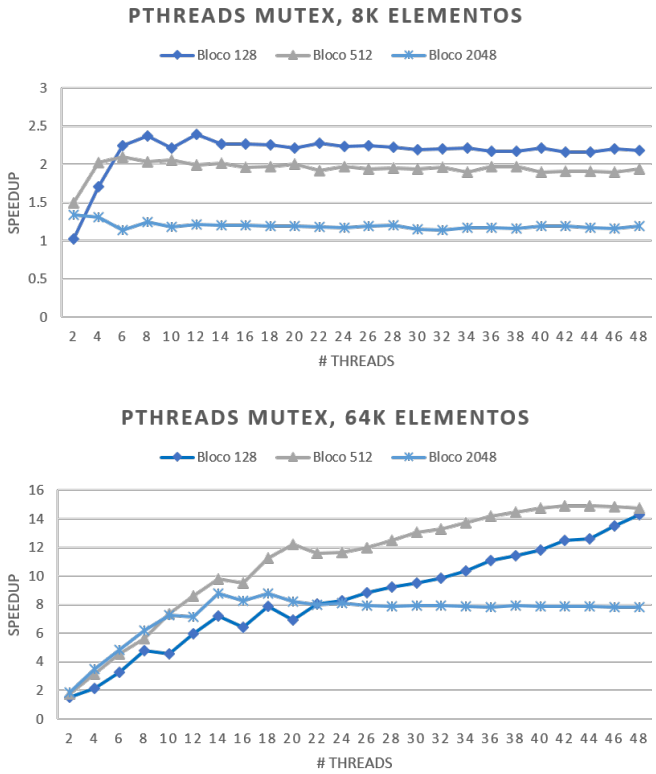


Figura 7. Ganhos da paralelização por blocos com Pthreads e Mutex

*5) **Comparação entre implementações**:* Tendo sido explicadas as várias implementações do algoritmo, bem como a forma como cada uma delas se comporta, resta agora efetuar um termo de comparação que permita perceber quais delas seriam mais adequadas a cada caso de teste.

Para tal, juntaram-se ar curvas correspondentes aos testes com blocos de 512 elementos, por apresentarem (no geral) um melhor desempenho, obtendo-se os seguintes resultados:

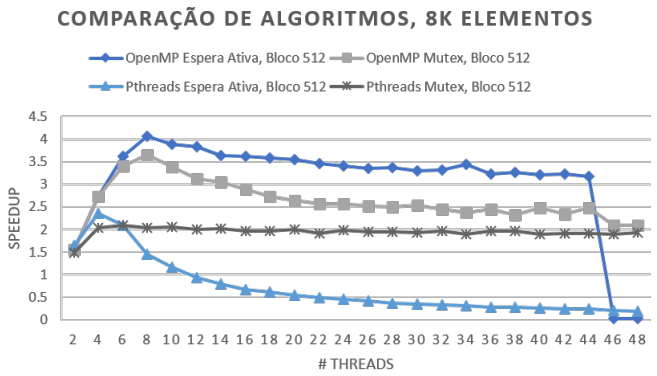


Figura 8. Comparação das várias implementações (volume de dados reduzido)

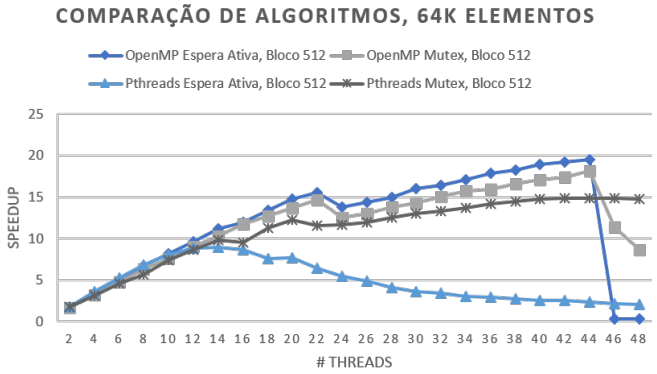


Figura 9. Comparação das várias implementações (maior volume de dados)

Como podemos observar na Figura 8 e na Figura 9, as implementações em OpenMP comportam-se bastante melhor do que as desenvolvidas com o Pthreads.

Quando olhamos para as diferenças entre os mecanismos de exclusão mútua, vemos que a implementação com espera ativa do OpenMP tem ganhos superiores aos conseguidos com as chamadas ao sistema operativo (operações *lock* e *unlock*). No entanto, as primeiras implicam que as *threads* estejam permanentemente ativas, reduzindo drasticamente a sua eficiência comparativamente à versão com mutex. Isto pode ter várias implicações, nomeadamente uma quebra de desempenho quando se passam a executar duas instâncias do mesmo programa, ou mesmo o sobreaquecimento da máquina em com vetores de dimensões superiores, que levaria a uma redução da velocidade de processamento.

III. Condições de teste

Todos os testes necessários à criação dos gráficos apresentados foram executados no cluster SeARCH [2], em máquinas Intel Xeon E5-2695v2 (nodos 662) Porém, apesar de não terem sido registadas as medições, observaram-se comportamentos idênticos nas máquinas Intel Xeon E5-2670v2 e Intel Xeon E5-2650v2, correspondentes aos nodos 652 e 641 do SeARCH.

Cada pondo dos gráficos foi calculado através da média dos dois melhores testes em cinco medições, respeitando uma tolerância mínima de 5%. Foram repetidos todos os testes com blocos de dimensão 128,256,512,1024 e 2048 elementos, sendo apresentados apenas os mais representativos. As dimensões dos vetores foram selecionadas de acordo com o tempo de computação, utilizado-se para tal um vetor de dimensões reduzidas (8192 elementos) e um vetor de dimensões ligeiramente superiores (com 65536 elementos), que já permite uma melhor percepção dos comportamentos. Um estudo mais exaustivo ao algoritmo passaria por utilizar vetores de dimensões consideravelmente maiores, talvez na ordem dos milhões de elementos.

IV. Conclusão

Apesar de o Bubble Sort parecer à partida um algoritmo bastante simples, a sua implementação paralela é ainda bastante complexa. Os vários objetivos do projeto foram analisados e implementados. Porém, não sendo o principal propósito do trabalho a máxima otimização, algumas (senão todas) as versões podem ainda ser melhoradas.

Ainda que se tenham adiantado possíveis justificações, ficaram por investigar alguns aspetos mais relacionados com a(s) máquina(s) de teste, como por exemplo a quebra de desempenho entre as 22-24 e as 44-48 *threads*.

Pode também ser analisado o impacto da variação do tamanho dos blocos durante a execução, de forma a que o número de blocos nunca (ou quase nunca) seja inferior ao número de *threads*, obtendo então um melhor balanceamento de carga, sob a penalização de aumentar o número de *locks*.

Poderiam ainda ser analisados diversos comportamentos como a resposta do algoritmo quando várias instâncias do programa estão a ser executadas na mesma máquina e em simultâneo, a eficiência computacional em cada uma das *threads*, entre outros.

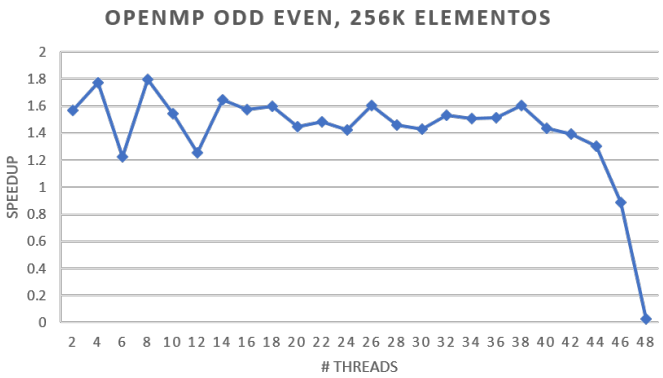
A análise do comportamento em vetores de tamanho superior seria também bastante importante, na media em que permitiria eliminar algum ruído de medição. De notar que foram incluídos no Anexo A gráficos correspondentes aos testes explicados neste documento, para vetores com 256k elementos. Contudo, não foram produzidos a tempo de ser devidamente analisados.

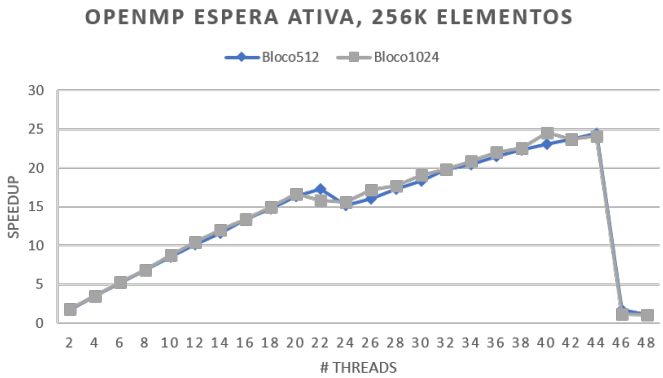
Referências

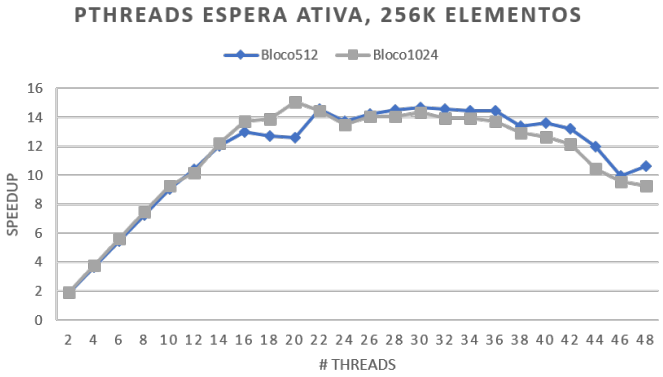
[1] Algorithms and Data Structures [Online]. Consultado a 17/02/2017. Disponível em: http://www.algolist.net/Algorithms/Sorting/Bubble_sort

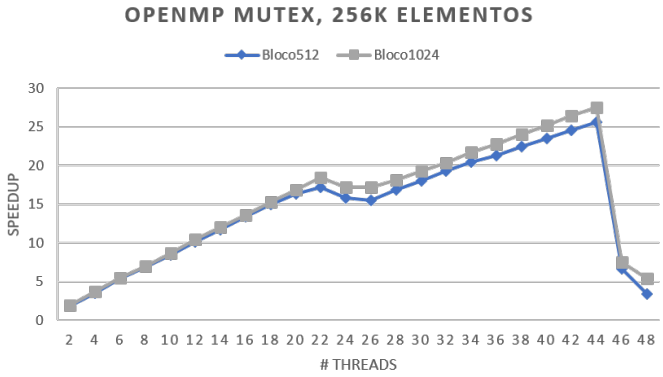
[2] Search-ON2: Revitalization of HPC infrastructure of UMinho, (NORTE-07-0162-FEDER-000086), co-funded by the North Portugal Regional Operational Programme (ON.2-O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF).

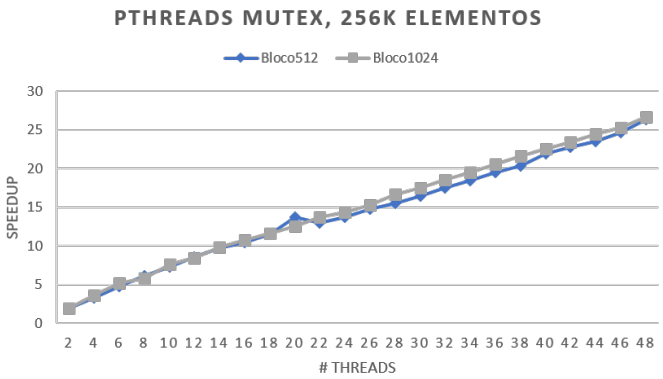
APÊNDICE A
MEDIÇÕES DE DESEMPENHO COM 256K ELEMENTOS

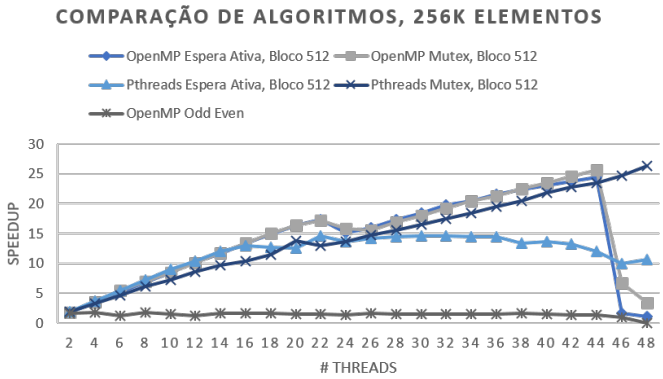












APÊNDICE B
ANÁLISE DE ESCALABILIDADE EM FUNÇÃO DO VOLUME DE DADOS