Engenharia dos sistemas de Computação - Bubble Sort -

autor: Daniel Malhadas

***Resumo*—O presente documento apresenta um estudo aprofundado sobre três diferentes implementações do algorítmo: Bubble Sort. A primeira é a implementação sequêncial clássica, a segunda uma implementação paralela que recorre às diretivas OpenMP para obter o referido paralelismo, a terceira é também uma implementação paralela, mas agora esse paralelismo é obtido com recurso a POSIX threads (Pthreads). Estas três implementações são então comparadas entre si em relação ao tempo de execução para diferentes conjuntos de dados com o intuito de entender quais as que permitem a obtenção de um menor tempo de execução. Os resultados são então organizados em gráficos com intuito também de entender a escalabilidade (relativamente aos conjuntos de dados) para cada implementação.**

***Index Terms*—Bubble Sort, OpenMP, POSIX Threads, Computação Paralela e Distribuída.**

I. Considerações Iniciais

A. O que é o algoritmo Bubble Sort

O algoritmo bubble sort é um algoritmo de ordenação simples que baseia o seu funcionamento e o seu nome no fenómeno físico observável ao deixar cair em água parada vários objetos de diferentes densidades. Cada um desses objetos irá comportar-se de forma diferente conforme a sua densidade. Aqueles que tiverem uma densidade superior à da água irão deslocar-se até ao fundo enquanto que os que tiverem uma densidade menor irão subir à superfície, de forma análoga a uma bolha de água que, pela sua baixa densidade sobe ao longo da água. O algoritmo que se pretende estudar emula este processo sendo a estrutura de dados que se pretende ordenar uma metáfora para a estrutura que é a água e, os elementos da estrutura de dados, são os objetos que se fazem cair na água.

B. Caracterização das Máquinas Utilizadas

Durante este projeto usaram-se duas máquinas principais sendo que a validação da correção dos algoritmos implementados terá sido realizada nas duas máquinas, para assim garantir que a correção se mantinha entre arquiteturas distintas. As máquinas usadas foram: um laptop pessoal e o nodo 652 do Cluster SeARCH, no entanto, os resultados apresentados ao longo deste documento serão todos relativos apenas ao nodo 652 do Cluster, pois o laptop pessoal é bastante mais limitado, tornando assim uma comparação direta pouco interessante. As máquians encontram-se completamente caracterizadas na seguinte tabela:

Caracteristicas	Nodo 652
Manufacturer	Intel
CPU Designation	Dual CPU E5-2670v2
CPU Microarchitecture	Ivy Bridge
Clock Frequency	1.2, 2.50GHz
RAM	64GB
Cores	20 with 40 threads (2 per core)
Peak FP Performance	400GFlops/s
Cache	L1d: 32kB, L1i: 32kB L2: 256kB, L3: 25600kB
Memory Bandwidth	59.7GB/s
Caracteristicas	**Laptop Pessoal**
Manufacturer	Intel
CPU Designation	Dual CPU i5 4200U
CPU Microarchitecture	Haswell
Clock Frequency	1.6, 2.6GHz (with turbo clock speed)
RAM	4GB
Cores	2, 4 threads
Peak FP Performance	41.6GFlops/s
Cache	L1i: 32KB, L1d: 32KB, L2: 0.5MB, L3: 3MB
Memory Bandwidth	2 channels, max 25.6GB/s

Esta informação foi obtida apartir de várias fontes distintas. Sendo elas: - o comando Unix **lscpu**; - o site **cpu-world.com**; - o site **ark.intel.com**.

C. Conjuntos de Dados

O algoritmo bubble sort aplica-se a uma lista de valores que se entende estarem de alguma forma desordenados pretendendo-se então ordená-los. Para a obtenção das análises de performance apresentadas no decorrer deste documento teve então de se pensar em conjuntos de dados para teste que fossem relevantes. Considerou-se que os valores em si a ordenar não são de todo o fator relevante, por esse motivo, todos os testes foram realizados com valores aleatórios. No entanto o espaço que estas listas ocupam em memória é já relevante e, por esse motivo, decidiu-se que a diferença entre cada conjunto de dados deveria ser o seu tamanho em memória. Foram então escolhidos os seguintes quatro conjuntos de dados:

1 - Um Conjunto de dados que encha por completo a Cache L1

O nodo 652 do Cluster descrito na sub-secção anterior tem L1d: 32kB e L1i: 32kB, apenas nos interessa dados, por isso temos um total de 32kB na Cache L1 para os nossos dados. Cada valor na lista será representado por um inteiro de 32 bits (4Bytes), logo este nível da cache poderá conter 32000B/4B=8000 valores. Mas não terminamos por aqui, podemos ainda ter em atenção o tamanho de uma linha da cache. Neste caso cada linha terá 64B, por isso cabem 16 valores por linha, logo o número total de valores na nossa lista a ordenar deve ser um múltiplo de 16 de forma a maximizar o proveito tirado da localidade espacial. 8000/16 = 500, logo

a dimensão ideal para a lista a ordenar é 8000 valores.
Os restantes conjuntos de dados foram calculados de maneira análoga.

2 - Um Conjunto de dados que encha por completo a Cache L2

a dimensão ideal para a lista a ordenar é 64000 valores.

3 - Um Conjunto de dados que encha por completo a cache L3

a dimensão ideal para a lista a ordenar é 6400000 valores.

4 - Um Conjunto de dados que não caiba em nenhum dos níveis de Cache anteriores e force o CPU a carregá-lo da DRAM

Para este conjunto de dados precisamos somente de uma lista que seja significantemente maior que 6400000 elementos. Encontrou-se então um valor relevante que é também múltiplo do número de valores que cabem numa linha da cache. Uma boa dimensão é 7200000.

II. Diferentes Implementações

A. Implementação Sequencial

```
void Bubble_sort(int a[], int n) {
  int list_length, i, temp;
  for (list_length = n; list_length >= 2;
      list_length--)                          (2)
    for (i = 0; i < list_length-1; i++)
      if (a[i] > a[i+1]) {
        temp = a[i];                          (1)
        a[i] = a[i+1];
        a[i+1] = temp;
      }
} /* Bubble_sort */
```

O algoritmo bubble sort sequencial clássico consiste em comparar cada par de elementos adjacentes na lista a ordenar e trocá-los se estiverem na ordem errada, esta troca está visível no ponto 1 circundado no código. Esta passagem pela lista é repetida até que mais nenhuma troca seja necessária. Uma possível otimização, presente no ponto 2, é diminuir o número de elementos a ordenar por iteração do ciclo exterior. Isto porque no fim da primeira iteração é garantido que o maior elemento fique na posição do fim da lista, na segunda iteração é garantido que o segundo maior elemento fique na posição antes da última, e por aí adiante. Logo entende-se que os elementos do fim já estão ordenados a cada iteração e podem portanto ser ignorados nas iterações posteriores.
Este algoritmo tem complexidade média e de pior caso (n^2), sendo n o número de elementos a ordenar. Quando a lista já está ordenada (melhor caso), a complexidade é somente O(n). Note-se também que este algoritmo é particularmente ineficiente, comparado com outros, quando n é demasiado grande ou quando a lista está completamente desordenada, sendo vantajoso mais quando o número de elementos fora de ordem é bastante reduzido.

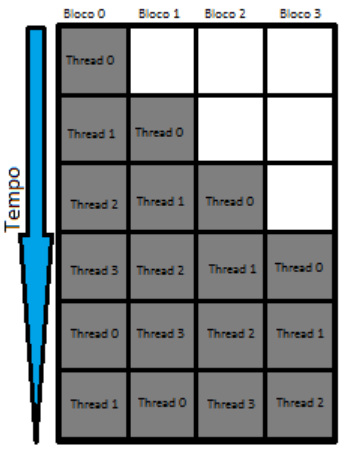
B. Implementação Paralela - OpenMP

```
void Bubble_sort_OMP(int a[], int n) {
  int list_length, block_count, i, temp;
  int block_size, lower, upper, block;
  block_count = num_threads;
  omp_lock_t m[block_count];
  block_size = n/block_count;

  for(i=0;i<block_count;i++)
      omp_init_lock(&m[i]);

  omp_set_num_threads(num_threads);
  #pragma omp parallel private(list_length,
      block, lower, upper, i, temp)
  {
     for(list_length=block_size;
        list_length>=1;list_length--)
        for(block=0;block<block_count;block++){
           lower=block_size*block;
           if(block==block_count-1) upper = n;   (2)
           else upper = block_size*(block+1);
           if(block==0) omp_set_lock(&m[0]);     (1)
           for(i=lower;i<upper-1;i++)
              if(a[i]>a[i+1]) swap(a+i,a+i+1);
           if(block==block_count-1){
              omp_unset_lock(&m[block]);         (3)
              break;
           }
           omp_set_lock(&m[block+1]);
           if(a[i]>a[i+1]) swap(a+i,a+i+1);
           omp_unset_lock(&m[block]);
        }
  }
} /* Bubble_sort_OMP */
```

Na explicação do subcapítulo anterior referimos que o algoritmo bubble sort consiste em dois ciclos, sendo um interior ao outro. Foi então necessário decidir qual dos ciclos paralelizar. Dois critérios foram tidos em conta: já que estamos a usar paradigmas fork&join que funcionam com base em "pools"de threads, o ciclo a paralelizar, idealmente, deve ser aquele que cause o mínimo de overhead relativo ao fork&join das threads; E o ciclo a paralelizar deve, idealmente, ser aquele cujas iterações dependem menos umas das outras. Quanto ao primeiro critério é fácil entender que o ciclo a paralelizar deveria ser o exterior visto que as threads serão lançadas somente uma vez, antes do ciclo, e juntas de novo somente no fim do ciclo ao invés de serem lançadas e juntas a cada iteração. Desta forma o overhead que o fork&join poderia causar torna-se praticamente nulo e deixa de ser um problema. Quanto ao segundo critério, observamos que as iterações de ambos os ciclos são dependentes das iterações anteriores. O ciclo exterior depende da iteração anterior para, por exemplo, saber que o último elemento da iteração anterior já se encontra ordenado (otimização referida na subsecção anterior). No entanto a dependência no ciclo interior é ainda mais forte, porque numa dada iteração tem-se a intenção de pegar no elemento que trocou de posição na iteração anterior (se tiver trocado) e verificar se deve ser trocado novamente para a posição seguinte. Se este seguimento se quebrar, o efeito da bolha que sobe pela água mencionado no primeiro capítulo deixa de existir e já não se pode garantir também que o elemento do fim

fique ordenado a cada iteração do ciclo exterior. Ambos os ciclos são problemáticos, mas já vimos em cima vantajens de paralelizar o exterior.

De forma a reduzir a dependência entre iterações no ciclo exterior, dividiu-se a lista em T partes, sendo este T igual ao número de threads a utilizar. Desta forma cada thread poderá trabalhar em cada uma destas partes/blocos de forma independente. Estes blocos devem ser operados por somente uma thread de cada vez de forma a eliminar a possibilidade de "data races". Para evitar que outra thread entre no mesmo bloco enquanto este está a ser computado por outra, a thread que entrou primeiro deve recorrer ao uso de um mutex para bloquear acesso ao bloco atual, vemos isto no ponto 1 circundado no código. Reparamos então que há a necessidade de um diferente mutex para cada diferente bloco. Note-se ainda que, para manter a correção do algoritmo, cada thread deve avançar de bloco para bloco por ordem, percorrendo assim ordeiramente o array, indo libertando o lock que faz para cada bloco no momento em que acaba de o percorrer e adquire o lock para o próximo bloco. O facto de apenas libertar um bloco quando adquire o próximo permite que, caso seja necessário, realize um swap entre os elementos nas fronteiras dos blocos. Desta forma os maiores elementos são realmente levados para o fim a cada iteração, sendo que enquanto uma thread percorre um bloco mais avançado, as outras estão ocupadas nos anteriores. Temos então uma estratégia que funciona como um pipeline onde, por cada bloco computado, uma iteração é terminada. O fator limitante desta paralelização torna-se então o tamanho de cada bloco, tendo em conta que cada bloco diminui com o aumento de threads estima-se que este algoritmo seja bastante escalável ao nível das threads. Segue-se uma imagem ilustrativa do algoritmo:



Como se vê na imagem, quando uma thread termina de percorrer o último bloco deve adquirir o primeiro e voltar lá. Note-se que agora é importante que a thread liberte de imediato o lock mal acabe o último bloco de forma a evitar um deadlock, caso contrário a thread no primeiro bloco nunca seria capaz de avançar e por isso nunca o libertaria. Desta vez não tem problema libertar o lock de imadiato, pois como é o bloco não precisamos de nos preocupar com o elemento na fronteira. Isto está visível no ponto 3 circundado no código

É ideal que cada bloco tenha o mesmo número de elementos, para assim haver um bom balanceamente de computação entre threads (embora por vezes alguns blocos possam ser menos custosos, como em situações onde há muito poucos swaps), mas por vezes o número de elementos no array não é divisível pelo número de threads disponível. Uma possível solução para este problema encontra-se no ponto 2 circundado no código. Nesse ponto vemos que, nessa situação, os valores "extra"são computados no último bloco. Não é expectável que este pormenor seja muito negativo na performance pois o número de elementos "extra"será sempre menor que o número de threads e por isso demasiado pequeno para realmente se considerar que quebra o balanceamento da computação entre threads.

Esta estratégia permite que o ciclo interior tenha uma complexidade complexidade de O(1) no caso em que temos n ou mais threads. Como o ciclo exterior continua com uma de O(n) vemos que este algoritmo tem uma complexidade média de O(n).

C. *Implementação Paralela - POSIX Threads*

```
void* sort_thread(void *param) {
   int list_length, block_count, i, temp;
   int block_size, lower, upper, block;
   block_count = num_threads;
   block_size = num/block_count;
   pthread_mutex_t *m = thread_mutexes;

   for(list_length = block_size; list_length
       >= 1; list_length--)
      for(block=0;block<block_count;block++){
         lower = block_size *block;
         if(block==block_count-1) upper = num;
         else upper = block_size*(block+1);
         if(block==0)
             pthread_mutex_lock(&m[0]);
         for(i=lower;i<upper-1;i++)
            if(arr[i]>arr[i+1])
               swap(arr+i,arr+i+1);
         if(block==block_count-1){
            pthread_mutex_unlock(&m[block]);
            break;
         }
         pthread_mutex_lock(&m[block+1]);
         if(arr[i]>arr[i+1])
             swap(arr+i,arr+i+1);
         pthread_mutex_unlock(&m[block]);
      }
   return NULL;
} /*sort_thread*/

void Bubble_sort_pthreads() {
   int ts[num_threads];
   for(thread=0;thread<num_threads;thread++){
      ts[thread]=thread;
      pthread_create(&thread_handles[thread],NULL,
                     sort_thread,NULL);
   }
   for(thread=0; thread<num_threads;thread++)
      pthread_join(thread_handles[thread],NULL);
   for(thread=0; thread<num_threads; thread++)
      pthread_mutex_destroy(&thread_mutexes[thread]);
}/*Bubble_sort_pthreads*/
```

a implementação usando POSIX Threads com recurso à biblioteca pthreads.h mantem exatamente a mesma estratégia de paralelizar o ciclo exterior e de dividir a lista em blocos. Ao observar o código note-se que todas as váriáveis não definidas no excerto apresentado são globais, podendo assim

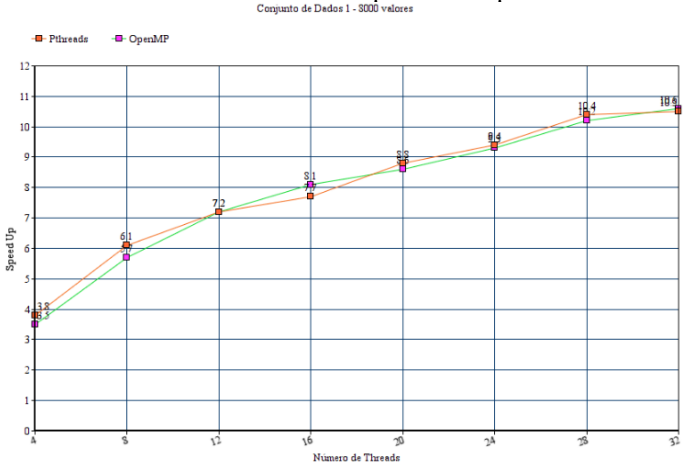
ser partilhadas entre diferentes threads. Basta olhar para o tamanho desta implementação para se notar que a natureza de baixo nível da biblioteca pthreads.h que implementa as POSIX threads nos permite um controlo muito maior sobre o que de facto está a acontecer, mas ao mesmo tempo aumenta bastante a complexidade do código em si. Nesta implementação temos de criar explícitamente uma "pool"de threads e temos também de arranjar um mecanismo para controlo de acesso à memória partilhada que, assim como na versão OMP, foi um conjunto de mutex. No código são visíveis duas funções. A função **Bubble_sort_pthreads** é responsável pela criação e posteriar junção das threads que irão executar paralelamente a função **sort_thread** sendo que esta implementa os ciclos do nosso algoritmo em si. O ponto circundado 1 mostra esta criação e o 2 a junção. Note-se ainda que, como se vê no ponto circundado 3, optou-se por não passar nenhum argumento à função que as threads irão executar. Poder-se-ia ter optado por passar um apontador para uma estrutura com todos os dados necessários ao invés de ter que manter várias variáveis globais e essa opção seria bastante mais "limpa"e agradável visualmente, no entanto entendeu-se que o impacto na performance inerente da necessidade de alocar memória para essa referida estrutura e de a operar em cada thread não se justificava.

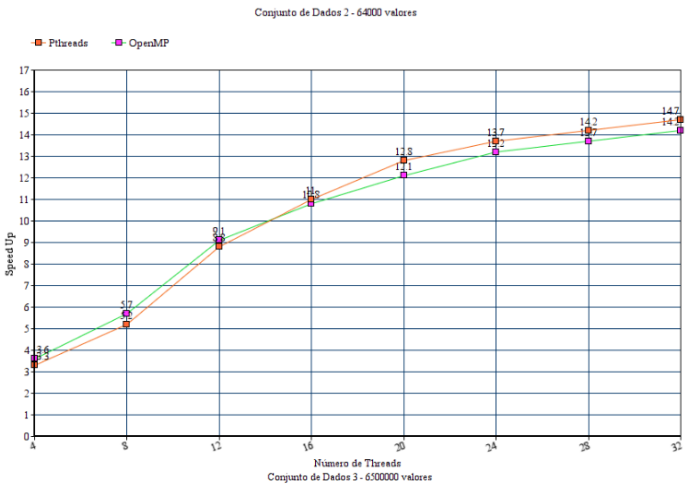
Como o algoritmo é exatamente igual ao anterior com OMP é possível observar no código as linhas que foram marcadas com pontos circundados no subcapítulo anterior e notar a forma como se comportam exatamente da mesma forma com o mesmo raciocínio implícito. Esta semelhança mantêm-se também na complexidade que, pelas mesmas razões que na versão anterior, se mantêm com complexidade média de O(n).

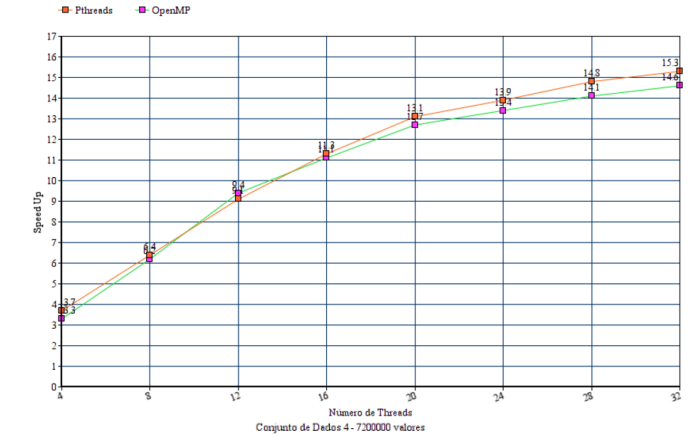
III. Medições de Performance

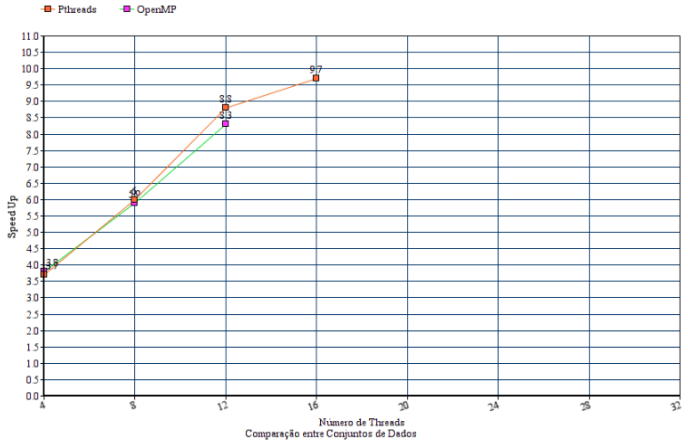
De forma a obter os resultados apresentados ao longo deste capítulo o código foi compilado com recurso a **gcc** com a flag de optimização **-O3** e a flag relativa ao OMP **-fopenmp**. Com o intuito de medir tempos de execução utilizou-se a biblioteca **time.h** e usou-se a expressão **(stop-start)/(CLOCKS_PER_SEC / 1000)** permitindo visualizar o tempo em milisegundos, sendo esse valor guardado sobre a forma de um double, permitindo uma precisão de 64 bits.

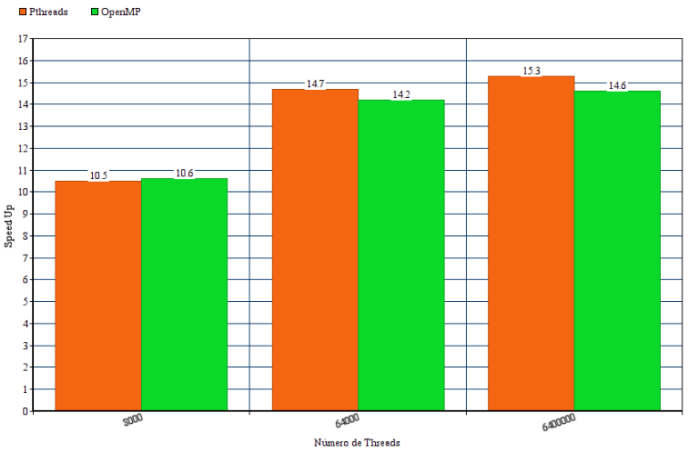
Apresentam-se agora gráficos que nos permitem observar os speed ups obtidos:











Cada um destes gráficos representa os speed ups obtidos com as implementações paralelas (em relação à implementação sequencial). Os primeiros 4 gráficos são relativos a um só conjunto de dados e podem ser usados para comparar as implementações usando esse específico conjuntos de dados. O último gráfico é um gráfico de barras que apresenta os melhores speed ups para cada conjunto de dados com cada implementação, permitindo assim uma comparação direta entre os três.

Os valores apresentados foram obtidos da seguinte forma: primeiro mediu-se o tempo da versão sequencial, chamar-lhe-emos **Tseq**, de seguida, para o número de threads pretendido (imaginemos que seja **NumT**), mediu-se o tempo da versão OpenMP um total de 10 vezes. Desses 10 tempos fez-se a média aritmética de todos os valores que não se desviassem mais do que 10% do melhor tempo medido, chamaremos a esta média **Tomp**. O mesmo se repetiu para a versão Pthreads e ao tempo resultante desta versão chamaremos **Tpth**. Temos então que o speed up para **NumT** threads da versão OpenMP = **Tomp/Tseq** e para a versão Pthreads = **Tpth/Tseq**. este processo repetiu-se para cada número de threads pretendido e para cada conjunto de dados (definidos no primeiro capítulo). Nota-se olhando para os gráficos que apenas se mede os speed ups até um máximo de 32 threads. Tal deve-se ao facto de a máquina usada ter apenas 20 cores físicos e, não haver grande relevância em ver os resultados obtidos com recurso a hyperthreading, pois é expectável que estes sejam piores. Apenas se mediu alguns valores depois das 20 threads com o intuito de verificar se realmente se notava alguma descida de speed up. Nota-se ainda que nos gráficos relativos ao último conjunto de dados nem todos os speed ups foram calculados. Isto acontece porque o tempo de execução com essas características se torna tão elevado que não foi possível obter resultados em tempo útil para assim fazer os cálculos necessários.

Observando agora os resultados obtidos notamos que o conjunto de dados sobre o qual se obteve um menor speed up foi o conjunto de dados 1 que contêm 8000 valores. Por ser um conjunto de dados demasiado pequeno estima-se que o overhead causado pela espera das threads por um bloco que ainda não tenha sido libertado por outra foi demasiado elevado, isto é, os cálculos efetuados por thread não compensaram o tempo que teve que esperar antes de poder avançar para um novo bloco. com 8000 valores, para 4 threads teremos 4 blocos com 2000 valores cada. Sabendo que ao iterar por estes blocos em muitas iterações não irão ocorrem swaps (ou seja, são praticamente instantâneas) podemos concluir que cada thread faz um trabalho mínimo, mas que mesmo assim ainda compensa um pouco já que se obtém algum speed up comparativamente com a versão sequencial. Há medida que o número de threads aumenta vemos que o speed up é cada vez menos significativo. Isto é normal, já que, por exemplo, com 20 threads cada thread fará à volta de 8000/20 = 400 iterações por bloco, onde muitas serão sem swap. É expectável que se se continuasse a aumentar significantemente o número de threads continuariamos a observar perdas graduais de speed up podendo até chegar a um ponto onde as threads causam um overhead tão escusado que teriamos uma perda de speed up em relação à versão sequencial.

Nota-se depois que o conjunto de dados com o qual se obteve o maior speed up foi o terceiro, com 6400000 valores. É expectável que, com este algoritmo se obtenham melhores speed ups para maiores conjuntos de dados (e este conjunto de dados é o maior sobre o qual se conseguiu fazer todas as medições) já que são os que compensam mais o overhead da gestão da memória partilhada. Observamos então que estas implementações paralelas são **escaláveis a nível dos dados** sendo pior para menores conjuntos de dados e, o facto de que realmente se obteve melhores speed ups com maiores conjuntos de dados prova que houve um bom balanceamento da computação entre threads, sendo que um mau balanceamento iria puxar os speed ups para mais perto do valor sequencial. Como se disse no capítulo anterior, o fator limitante deste algoritmo é o tamanho dos blocos, quanto menores mais rapido será o algoritmo e maior o speed up, ou seja, o algoritmo também é **escalável a nível das threads**, no entanto há que entender que deve haver também um limite para quão pequenos estes blocos devem ser, visto que, como vemos por observação dos speed ups obtidos com o conjunto de dados mais pequeno, blocos demasiado pequenos irão causar overhead muito mais difícil de compensar. Portanto aumentar o número de threads será vantajoso na maior parte dos casos, mas apenas uma análise com mais conjuntos de dados e um maior número de threads poderia realmente identificar um ratio interessante entre valores a ordenar e threads que permitissem alcançar o pico do speed up. De qualquer forma, qualquer ratio encontrado seria sempre uma estimativa crassa pois nunca temos forma de dividir de maneira justa a computação entre threads, já que não temos maneira de à priori saber que iterações são custosas (fazem swap) e quais não o são. Por isso é perfeitamente possível que haja uma thread que não faça nenhum swap e por isso termine de imediato enquanto outra encontre swaps em todas as posições por onde passa e por isso demore vários milisegundos com computação que deveria ser dividida pelas threads igualmente. Esta é uma segunda limitação deste algoritmo e uma provável razão para que haja speed ups mais acentuados com menor número de threads já que, quanto maior o número de threads maior é a probabilidade de que alguma delas não faça nada que seja realmente relevante num bloco que percorre. Inputs grandes são os únicos capazes de disfarçar esta falha já que mesmo com muitas threads os blocos serão grandes. O facto de com um elevado número de threads se estar a usar hyperthreading (a partir das 20 threads) também poderá influenciar e ser uma razão pela qual a curva de speed up deixa de ser tão acentuada por volta das 20 threads.

Por último vemos que das duas implementações paralelas, aquela que forneceu os maiores speed ups foi a implementação que usa Pthreads. Isto era o que se esperava visto que o uso da biblioteca pthreads.h permite a criação de um código mais "low-level"do que o uso de diretivas OpenMP. Isto permite-nos fazer escalonamento das threads diretamente sendo que o programa não terá que incorrer num overhead de ter de determinar o escalonamento das mesmas em tempo real.

IV. Conclusões

Ao longo deste documento explicaram-se três possíveis implementações para o algoritmo bubble sort e estudaram-se as limitações do mesmo comparando-se as diferentes implementações. Afirma-se que os speed ups obtidos foram bons e aquilo que se esperava, mas pecando apenas talvez por um pequeno número de conjuntos de dados para teste. Com mais tempo talvez fosse relevante estudar mais inputs intermédios (nem grandes nem pequenos) que permitissem melhor concluír algo sobre as implementações nessa fase até porque afirmar sem experimentar é em si só uma grande falácia e como tal mais experimentação permitir-nos-ia fazer afirmações mais certas e fundamentadas. poder-se-ia fazer uma maior quantidade de testes até ao ponto em que se podesse prever uma ligação entre o número de valores a ordenar e o número de threads disponível de forma a poder obter o pico de speed up. Para já é expectável que essa relação seja possível de se encontrar e, não fosse este um algoritmo pior que outros com o mesmo propósito (como o quicksort, mergesort, etc...), poderia até ser algo de interessante a fazer.

Por fim, entende-se que as estimativas dos resultados foram bem sucedidas e confirmadas na sua maioria com os testes realizados. Pensa-se também que, embora poucos, os conjuntos de dados cubriram um terreno grande o suficiente para se ter no mínimo um panorama geral do comportamento do algoritmo implementado. Por estas razões considera-se que este estudo foi completado com sucesso.