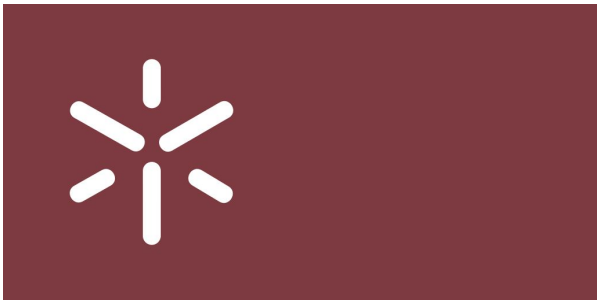


Universidade do Minho

Mestrado Integrado em Engenharia Informática

Luís Pereira A77667

Computação Paralela

Relatório QuickSort

25 Janeiro 2021

Índice

Introdução

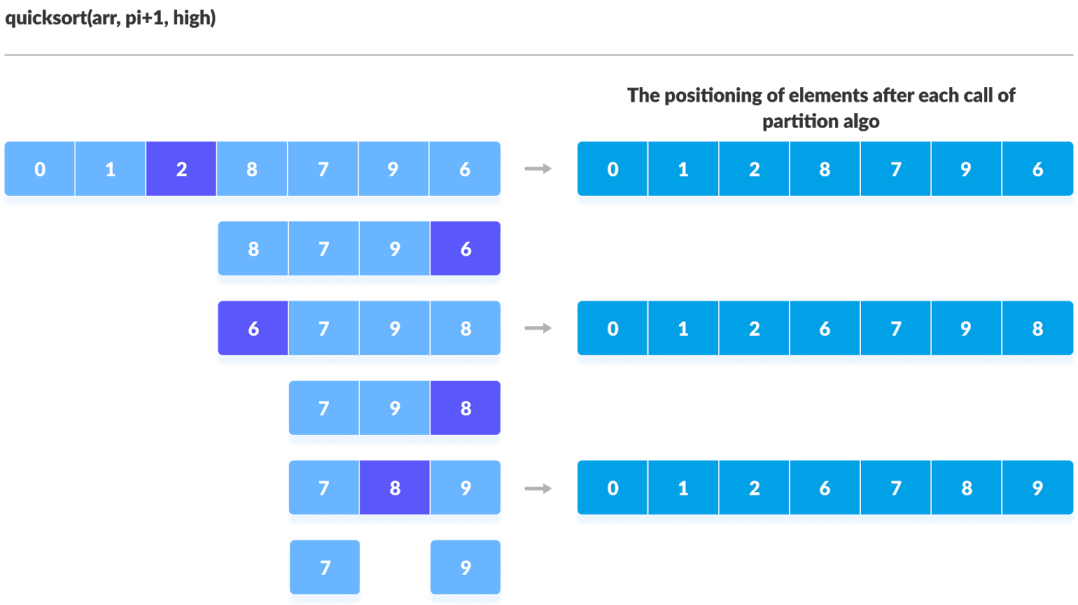
Tradicionalmente, o software tem sido escrito para ser executado sequencialmente. Para resolver um problema, um algoritmo é construído e implementado como um fluxo serial de instruções. Tais instruções são então executadas por uma unidade central de processamento de um computador. Somente uma instrução pode ser executada por vez, após sua execução, a próxima então é executada.

Por outro lado, a computação paralela faz uso de múltiplos elementos de processamento simultaneamente para resolver um problema. Isso é possível ao quebrar um problema em partes independentes de forma que cada elemento de processamento pode executar sua parte do algoritmo simultaneamente com outros. Os elementos de processamento podem ser diversos e incluir recursos como um único computador com múltiplos processadores, diversos computadores em rede, hardware especializado ou qualquer combinação dos anteriores.

O problema concreto em análise no relatório do trabalho prático de Computação Paralela será a implementação de uma versão optimizada e paralela do algoritmo de divisão e conquista QuickSort.

Análise e Implementação do Algoritmo QuickSort

Um pivot do array foi escolhido, neste caso o último elemento. Os elementos do array menores que o pivot são postos à esquerda do mesmo e os maiores à direita dividindo o array em sub-arrays usando o pivot como ponto de partida. As sub-partições são então divididas recursivamente até chegarem a um único elemento, momento em que o array está ordenado.



Implementação Sequencial

```c
void quickSort(int array[], int low, int high)
{
    if (low < high)
    {
    /* pi is partitioning index, array[p] is now
       at right place */
        int pi = partition(array, low, high);
        // Separately sort elements
        quickSort(array, low, pi - 1);
        quickSort(array, pi + 1, high);
    }
}
```

```c
int partition (int array[], int low, int high)
{
    int pivot = array[high];// pivot
    int i = (low - 1);//Index of smaller element
    for (int j = low; j <= high- 1; j++)
    {
        if (array[j] <= pivot)
        {// increment index of smaller element
            i++;
            swap(&array[i], &array[j]);
        }
    }
    swap(&array[i + 1], &array[high]);
    return (i + 1);
}
```

Uma recriação do código inicial fornecido em tentativa de criar um código mais vectorizavel do que o original.

Vetorização AVX2

Analisando a parte mais dispendiosa do processo de QuickSort, o ciclo for da função partition podemos observar que não se encontra a ser vetorizado automaticamente devido a problemas de **control flow** derivados da comparação que cada ciclo tem de fazer.

```
seq.c:24:2: missed: couldn't vectorize loop
seq.c:24:2: missed: not vectorized: control flow in loop.
```

Possível Vetorização e melhor desempenho AVX2

```c
#include <immintrin.h>
__m256d A0;
__m256d M0;
__m256d PIVOT;
PIVOT = _mm256_broadcast_sd(&pivot); // Criaçao do vector de pivot
A0 = _mm256_loadu_pd(array + 0 * 4);//load de 4 doubles para vector
M0 = _mm256_cmp_pd(A0, PIVOT, 2);   //comparaçao <=
```

Uma possível abordagem para vetorizar o problema de partição seria utilizar intrinsics.
Neste exemplo com doubles utilizo intrinsics da intel para efetuar a comparação de 4 elementos simultaneamente recorrendo a dois vectores, um formado somente pelo elemento pivot e outro com os primeiros 4 elementos a ser comparados. Efetuando 4 comparações simultaneamente.

Implementação da Solução

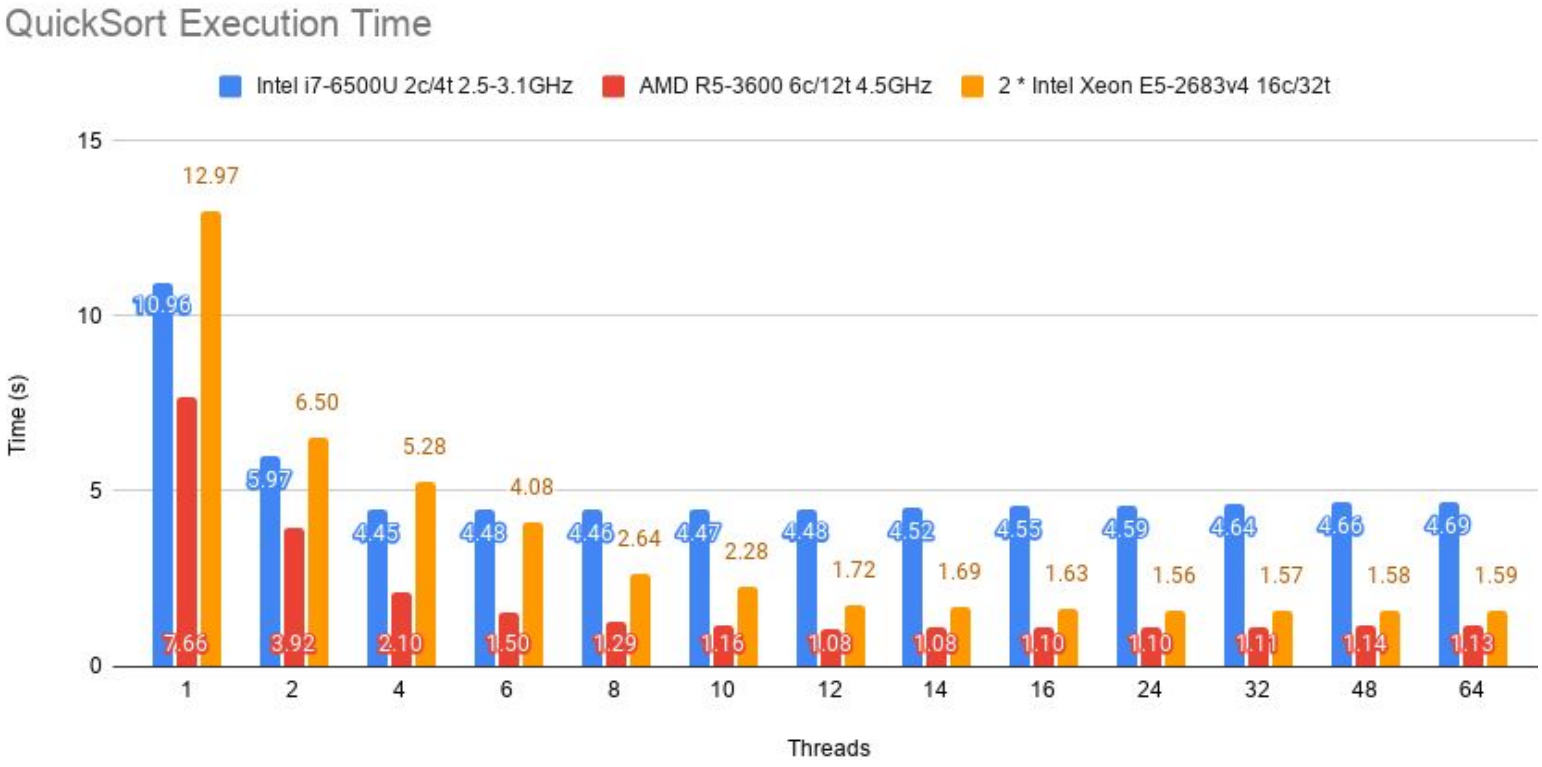
Realizada a análise do algoritmo sequencial, e a identificação das zonas críticas do algoritmo na qual o acesso ao array e as variáveis dos índices podem sofrer concorrência de escrita, foi Implementada a solução recorrendo à clause inicial **shared** do array e size para especificar a partilha desta memória pelos vários threads e à clause **firstprivate** fazendo o array e as variáveis dos índices, privadas e já inicializadas para cada thread.
Nesta implementação cada thread irá executar cada uma das chamadas recursivas a cada sub-partição recorrendo à clause **#pragma omp task.** Após análise de uma primeira tentativa, foi averiguado que para um número relativamente baixo de elementos a implementação sequencial apresentava um melhor resultado. Efetuando as mudanças necessárias para que a segunda implementação seja mista entre paralela e sequencial, o programa atingiu um nível de desempenho esperado.

```
#pragma omp parallel num_threads(64) default(none) shared(array,SIZE)
{
    #pragma omp single nowait
    {
    quickSort(array, 0, SIZE-1);}
}
void quickSort(int array[], int low, int high)
{
    if (low < high) {
        int pi = partition(array, low, high);
        /* pi is partitioning index, array[p] is now at right place */
        if (high - low < 10000) {
            quickSort(array, low, pi - 1);
            quickSort(array, pi + 1, high);
        } else {
            #pragma omp task default(none) firstprivate(array,low,pi)
            {
                quickSort(array, low, pi - 1);
            }
                #pragma omp task default(none) firstprivate(array,high,pi)
                {
                        quickSort(array, pi + 1, high);
                }
            }
    }
}
```

Demonstração e Análise dos Resultados

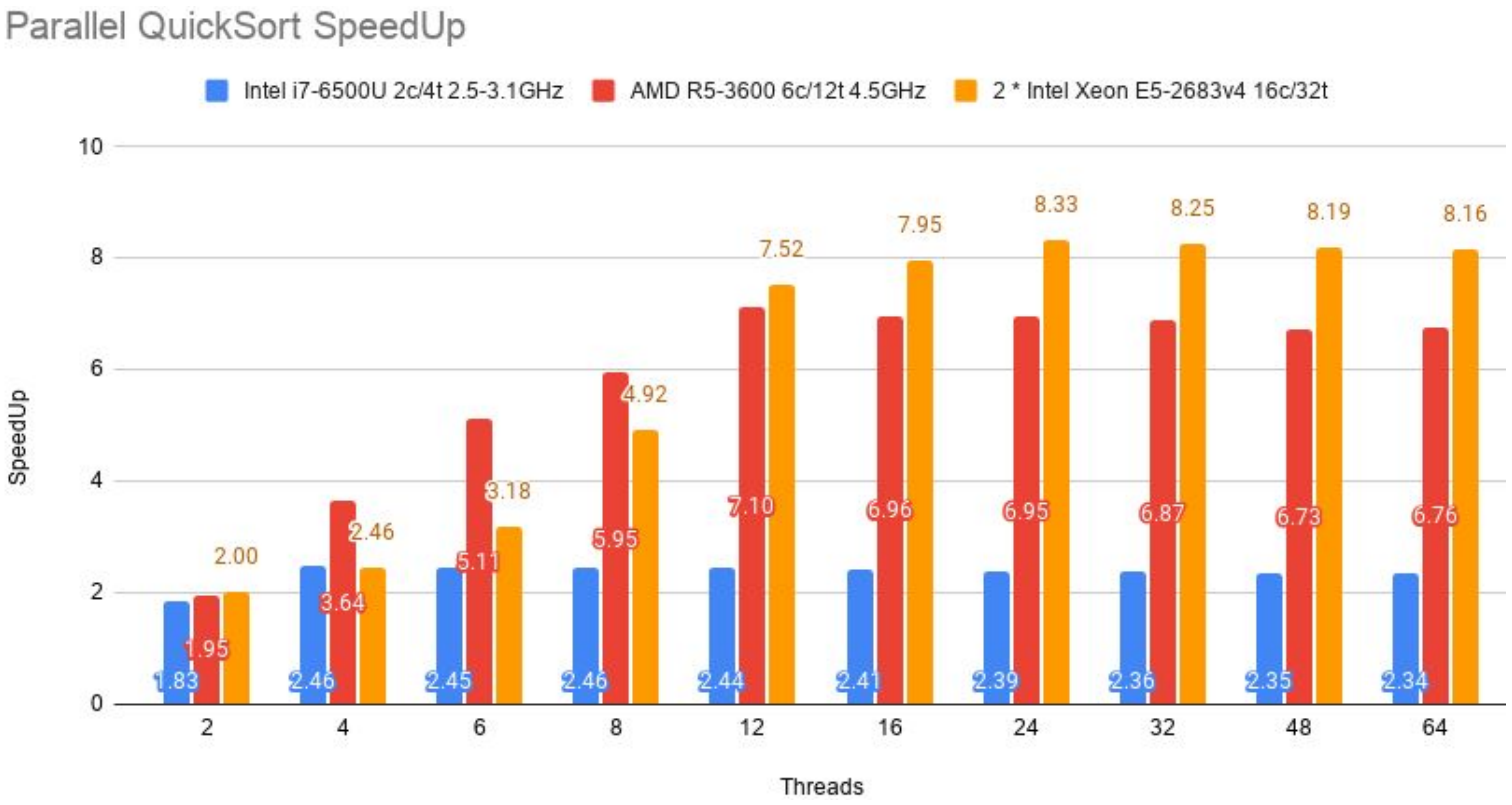
Desenvolvida a implementação final da solução do problema será então iniciada a fase de testes e análise de resultados. Para isso não só foram analisados resultados do node 781 do cluster mas também recorrendo a uma máquina com um Intel i7-6500U que dispõe de 2 cores físicos e 4 threads disponibilizando hyperthreading com base clock de 2.5Ghz e boost clock de 3.1Ghz. Uma outra máquina com um AMD R5-3600 que dispõe de 6 cores físicos e 12 threads disponibilizando simultaneous multithreading com overclock estático de 4.5Ghz. Para cada máquina e número de threads foram retiradas 10 amostras e efetuado o cálculo da mediana.

Tempo de execução



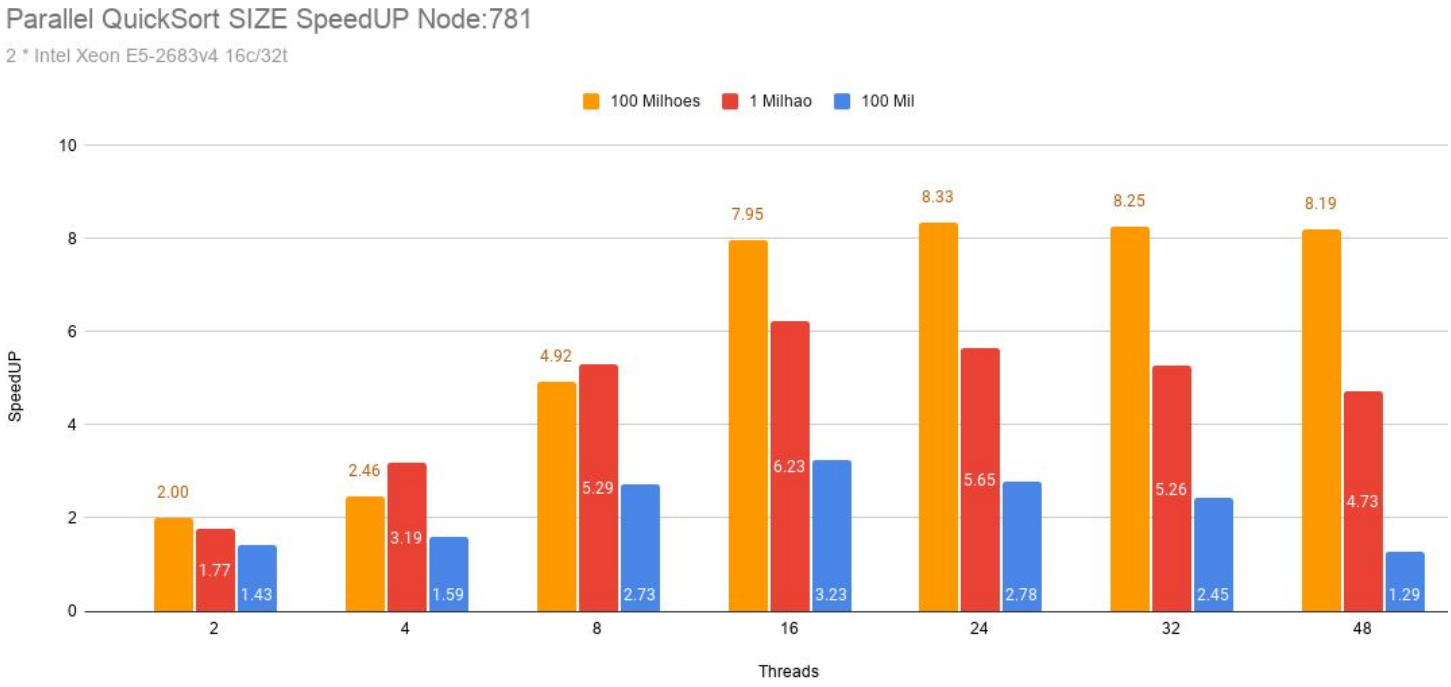
Analisando o tempo de execução em cada número de threads em diferentes máquinas para um array com 100 Milhões de elementos podemos observar que o programa apresenta uma boa escalabilidade até 24 threads, a partir da qual o overhead torna ineficiente o uso de um número superior de threads. Também se pode observar os efeitos de uma velocidade de clock superior, apresentando um tempo de execução menor mesmo utilizando menos threads.

SpeedUp



Após o cálculo da mediana dos tempos de execução foi feita a divisão entre o tempo sequencial e o tempo em cada número de threads. Como se pode verificar o ganho pelo PU i7-6500U e o R5-3600 é estagnado no momento em que o pu alcança o seu número de threads, diminuindo devido ao overhead que cada thread adicional apresenta. Neste momento podemos verificar como anteriormente que para 100 Milhões de elementos, o melhor SpeedUP é representado por 24 threads com um valor de SpeedUp 8.33x comparado com o sequencial.

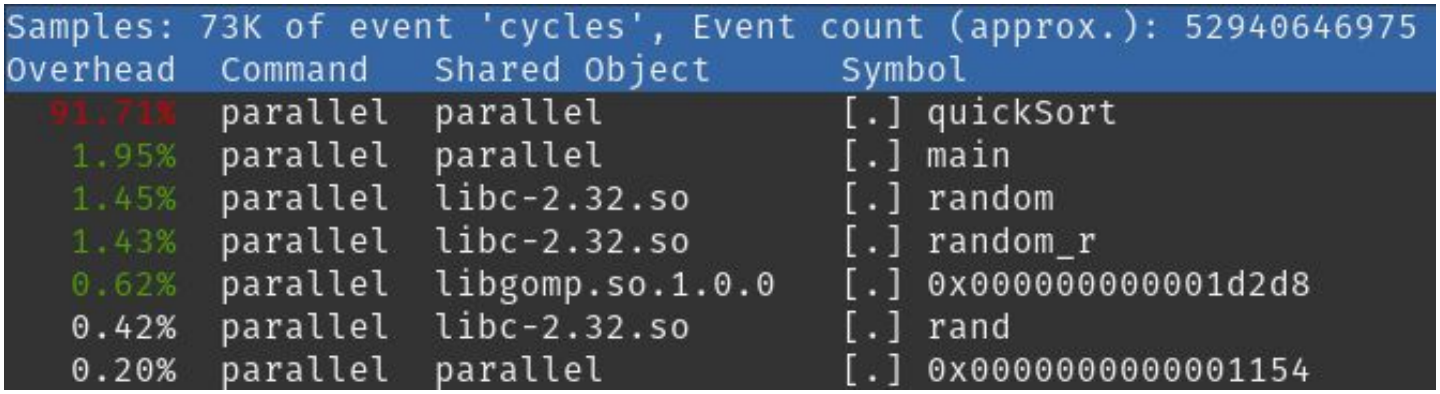
SpeedUp por Tamanho de Array



Escalabilidade

Para uma melhor compreensão da escalabilidade do programa, foi feita uma análise do SpeedUp para várias dimensões de array utilizando sempre o mesmo nodo 781 do cluster que apresenta 32 cores físicos e 64 threads. Como podemos verificar pela figura à medida que a quantidade de elementos no array diminui, a escalabilidade do programa também diminui, apresentando o melhor SpeedUp cada vez em número de threads menor. Isto deve-se à quantidade de sub-partições diminuir com a diminuição do tamanho do array. Por sua vez, a utilização de tamanhos superiores ao testado poderão efetuar ainda mais sub-partições que por sua vez podem utilizar em paralelo um número superior de threads.

Percentagem em área paralela



```
Samples: 73K of event 'cycles', Event count (approx.): 52940646975
Overhead  Command   Shared Object      Symbol
  91.71%  parallel  parallel           [.] quickSort
   1.95%  parallel  parallel           [.] main
   1.45%  parallel  libc-2.32.so       [.] random
   1.43%  parallel  libc-2.32.so       [.] random_r
   0.62%  parallel  libgomp.so.1.0.0   [.] 0x000000000001d2d8
   0.42%  parallel  libc-2.32.so       [.] rand
   0.20%  parallel  parallel           [.] 0x0000000000001154
```

Utilizando o comando perf analisou-se a percentagem de tempo que cada parte do programa utiliza durante a sua execução, chegando a uma conclusão positiva de que efetivamente a maior parte do tempo (~=91.7%) é gasto dentro da secção paralela do código.

Conclusão

Termino este trabalho sobre os conceitos de computação paralela utilizando OpenMP, podendo verificar ganhos significativos em performance com uma implementação paralela do algoritmo de divisão e conquista QuickSort.
Após uma análise da sua escalibilidade foi observado que o número de threads mais eficiente aumenta à medida que o número de partições aumenta, o que por sua vez aumenta com o tamanho do array a ordenar.
A implementação de uma versão vetorizada aplicando o conceito acima referido poderia levar a um menor tempo de execução aproveitando a capacidade de operações vetoriais do PU.