**Relatório**

**Trabalho Prático 2**

***Quicksort***

**Universidade do Minho**

**Mestrado Integrado em Engenharia Informática**

Computação Paralela e Distribuída

Paradigmas de Computação Paralela

Carlos Rafael Cruz Antunes

Nuno André da Silva Oliveira

a67711

a67649

Resumo

O *quicksort* é um algoritmo que ordena um *array* dividindo-o em partes mais pequenas e ordenando-as. Este relatório serve para explicar a implementação paralela no segundo trabalho prático sobre este algoritmo. Serve também para estudar e analisar os resultados obtidos usando um diferente número de *cores* e um diferente número de tamanho de dados a ordenar. Usando estes dados podemos criar gráficos estatísticos e calcular o *speed* *up* desta versão paralela em relação à sequencial, e a escalabilidade do código produzido.

Algoritmo

A versão sequencial implementada foi baseada na apresentação deste trabalho: uma versão do quicksort recursiva que calcula o elemento a meio do array e vai trocando elementos que estejam à esquerda e que sejam maiores do que o elemento do meio por elementos da direita que sejam menores. Este algoritmo utiliza uma técnica “*divide* *and* *conquer*“, isto é, no fim da troca referida anteriormente, o array é dividido em dois, e é chamada a mesma função quicksort nos dois arrays.

A versão paralela foi implementada usando OpenMPI. Neste caso cada processo tem uma parte do *array* e o objetivo é que o *array* fique ordenado não só no próprio processo mas também no array global formado por todos os processos estando ordenado pelo *rank* dos mesmos. Estes processos começam por calcular o máximo e o mínimo do seu array e envia-los para o primeiro processo, que vai calcular o máximo global do *array* e sincronizar todos os máximos locais dos processos para o máximo global. Com esta informação cada um dos processos calcula o valor *step* que vai ser usado no algoritmo que define o intervalo de valores que cada processo vai armazenar e ordenar. De seguida cada processo envia os elementos do *array* que possui que não lhe pertencem e envia para o respetivo processo. Depois da troca efetuada, cada processo já só tem uma parte do array desorganizada mas com o limite máximo e mínimo estabelecido anteriormente. Por exemplo, se para o máximo de um certo *array* for 50, e o mínimo 0, numa execução com cinco processos, o primeiro ficaria com todos os elementos de 0 a 10, e segundo processo com elementos de 11 a 20, e assim sucessivamente. Agora que toda a comunicação necessária está resolvida cada um dos processos corre a versão do quicksort sequencial para o array obtido.

De notar que este algoritmo num caso extremo pode ser equivalente ao sequencial, se o array em causa só tiver elementos de 0 a 10, e apenas um elemento 50. Isto implicaria que o primeiro processo ficaria com o trabalho todo. Como no caso de estudo o array é gerado aleatoriamente a probabilidade de acontecer uma situação como esta é extremamente improvável.

Esta implementação do algoritmo de ordenação teve como objetivo reduzir ao máximo o número de comunicações (visto que num sistema de memória distribuída estas têm um peso significativo), evitar sempre que possível o uso de um processo central, e evitar comunicações na parte “pesada” do algoritmo. Por outro lado este algoritmo peca no balanceamento dos processos o que pode levar a que para *arrays* que tenham os números a tender para uma gama de valores e com valores muito distantes dessa gama tenha fraco desempenho.

Ambiente de Teste

Os testes de desempenho foram executados no *cluster* da Universidade do Minho (*SeARCH*). Foram pedidas duas máquinas, uma com 2 processadores com 8 *cores* em cada (Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz) com *hyper*-*threading* (nodo 641 do *cluster*). Totalizando em 32 *cores*, 16 destes virtuais. A segunda máquina tem 2 processadores com 12 cores em cada (Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz) com *hyper*-*threading* (nodo 662 do *cluster*). Totalizando em 48 *cores*, 24 destes virtuais.

Ambas as máquinas têm 32 *KBytes* de cache nível um para instruções e outros 32 *KBytes* de cache nível um para dados. Tem também 256 *KBytes* de cache nível dois. O nodo 641 tem 20 *MBytes* de cache nível três enquanto o nodo 662 tem 30 MBytes.

Todas as medições foram executadas cinco vezes, e o valor utilizado para estatísticas foi o melhor valor nessas medições, pois é o que se aproxima mais do melhor caso. Foram medidos os tempos de execução para 1, 2, 4, 8, 16, 20, 28 e 32 processos.

Os testes de desempenho foram compilados com o comando *mpicc* (*flags* usadas: -O3). Foram efetuados testes com diferentes tamanhos para que a cache fosse ou não fosse suficiente.

Testes de Desempenho

Os testes da cache nível um (Gráfico 3) revelaram que para tamanhos do *array* muito baixos (5.000 elementos, 19.531 *KBytes* neste caso) o ganho é relativamente pequeno, e a partir de quatro processos o desempenho é cada vez pior. Os testes da cache nível dois (Gráfico 4 - *array* com 50.000 elementos, totalizando em 195 *KBytes*) tiveram ganhos bons até aos oito processos. A partir de 16 processos o desempenho continua melhor de que a versão sequencial, mas começa a piorar em relação à execução com oito processos. Os testes para a cache nível três (Gráfico 5 - *array* com 4.000.000 elementos, totalizando em cerca de 15 *MBytes*) obtiveram resultados cada vez melhores quanto mais se aumentavam os processos no nodo 662. No nodo 641 a partir do vigésimo oitavo processo o desempenho piora acentuadamente.

Para efetuar testes de desempenho no algoritmo quando este tem que usar a memória RAM (Gráfico 6), criou-se um *array* com quarenta milhões de elementos, o que equivale a cerca de 152 *MBytes*. Verificou-se tempos de execução mais lentos do que obtidos nas memórias cache, como esperado, e verificou-se um aumento no desempenho do algoritmo, até um pico de cerca de 12 vezes. Mais uma vez o nodo 641 demonstrou perda de desempenho para um elevado número de processos, por outro lado o nodo 662 manteve ganhos de *performance* constantes.

O Gráfico 1 e o Gráfico 2 representam o tempo de execução do algoritmo em função do número de processos atribuído, para os diferentes tamanhos do array.

# Nodo 641

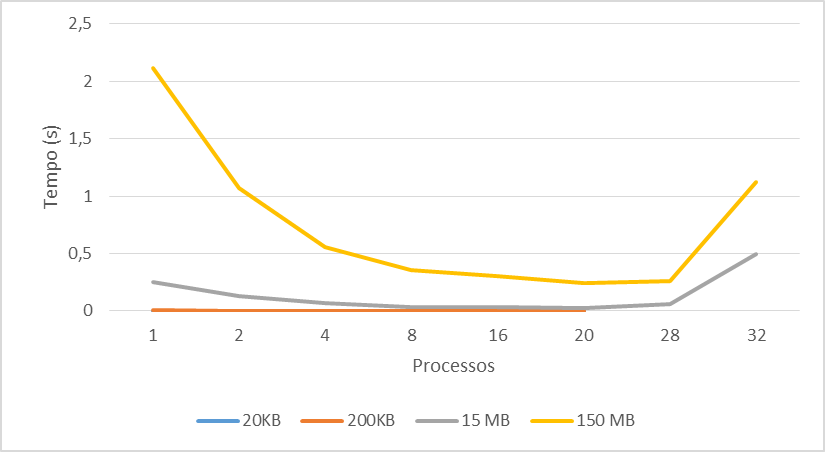


Gráfico 1 - Tempo de execução do quicksort no nodo 641

# Nodo 662

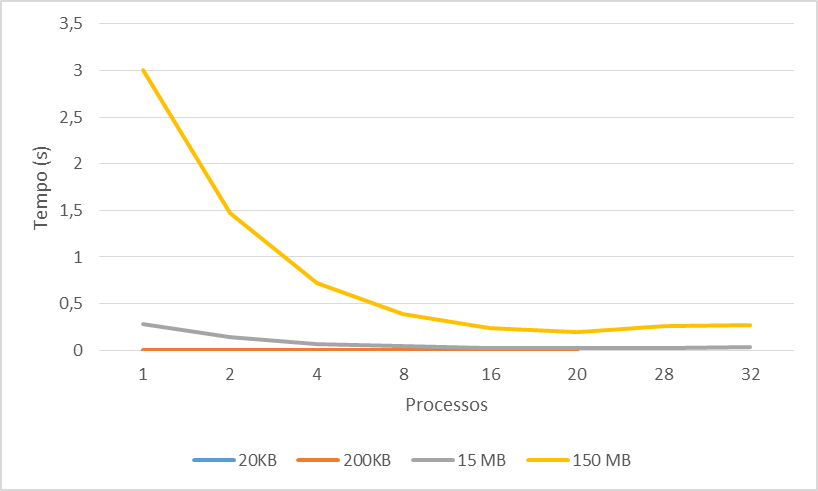


Gráfico 2 - Tempo de execução do quicksort no nodo 662

Análise de Resultados

Com os resultados obtidos podemos concluir que o nodo 662 é mais lento na versão sequencial, mas escala muito melhor do que o nodo 641. Este é mais rápido na versão sequencial mas escala pior, e acaba por ser ultrapassado pelo nodo 662. Isto deve-se ao facto de que o nodo 641 é constituído por *cores* com uma frequência superior ao nodo 662 (melhor na versão sequencial), mas o nodo 662 têm mais *cores* do que o anterior, o que o torna melhor na versão sequencial.

Comparando aos resultados do primeiro trabalho prático verificamos que na implementação em *OpenMPI* atingimos *speed* *ups* de cerca de 12 vezes, enquanto que na implementação em *OpenMP* conseguimos apenas *speed* *ups* de cerca de 4 vezes. Isto deve-se ao facto que depois da comunicação inicial, a implementação em OpenMPI é estritamente paralela, enquanto que a implementação em OpenMP pode ter dependências de acessos à memoria.

Anexos

Gráfico 3 - Tempo de execução do quicksort para um array de 20 KBytes

Gráfico 4 - Tempo de execução do quicksort para um array de 200 KBytes

Gráfico 5 - Tempo de execução do quicksort para um array de 15 MBytes

Gráfico 6 - Tempo de execução do quicksort para um array de 150 MBytes