**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 uma a 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. Estes processos começam por calcular o seu máximo e o seu mínimo e sincronizam com o primeiro processo, que vai calcular os pedaços do array que vão ser trocados entre cada processo. Depois da troca efetuada, cada processo já só tem uma parte do array desorganizada mas com um limite máximo e mínimo estabelecido para o seu processo. 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.

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.

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 2, 4, 8, 16, 20, 28, 32, 40 e 48 processos.

Testes de Desempenho

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.

Os testes neste nível de cache revelaram que para tamanhos do *array* muito baixos (5.000 elementos, 19.531 *KBytes* neste caso)

Os testes neste nível de cache (*array* com 50.000 elementos, totalizando em 195 *KBytes*)

Os testes para este nível de cache (*array* com 4.000.000 elementos, totalizando em cerca de 15 *MBytes*) obtiveram resultados

Para efetuar testes de desempenho no algoritmo quando este tem que usar a memória RAM, criou-se um *array* com vinte milhões de elementos, o que equivale a cerca de 76 *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

>>>>>>>>>>>>>>>>>>>>>>> UPDDATE <<<<<<<<<<<<<<<<<<<<<<

# 641

>>>>>>>>>>>>>>>>>>>>>>> Inserir gráficos e bla bla bla <<<<<<<<<<<<<<<<<<<<<<

# 662

>>>>>>>>>>>>>>>>>>>>>>> Inserir gráficos e bla bla bla <<<<<<<<<<<<<<<<<<<<<<

Análise de Resultados

>>>>>>>>>>>>>>>>>>>>>>>

comparar os 2 resultados

Falar um bocado sobre o que correu mal/pior

A influência do custo de comunicação

Pode-se meter aqui um gráfico comparativo se tivermos espaço

<<<<<<<<<<<<<<<<<<<<<<