**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* ….

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

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 12 *cores* em cada (Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz) com *hyper*-*threading* (nodo XxXX do *cluster*). Totalizando em 48 *cores*, 24 destes virtuais.

Esta máquina tem 32 *KBytes* de cache nível um para instruções e outros 32 *KBytes* de cache nível um para dados. Tem ainda 256 *KBytes* de cache nível dois e 30 *MBytes* de cache nível três.

A segunda máquina tem Xx processadores com Yy cores em cada () com *hyper*-*threading* (nodo XxX do *cluster*). Totalizando em Zz *cores*, Uu destes virtuais.

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

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, 32, 40 e 48 *threads*.

???

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

Testes de Desempenho

Os testes de desempenho foram compilados com o comando XXXXX. 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 5.000.000 elementos, totalizando em cerca de 19 *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

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