CAMPUS ALEGRETE
CIÊNCIA DA COMPUTAÇÃO

Relatório Técnico Científico

Trabalho de Introdução ao Processamento Paralelo
Merge Sort

Marcos Vinícius Treviso
marcosvtreviso@gmail.com

Alegrete, Agosto, 2013

RESUMO

O trabalho refere-se à paralelização do algoritmo de ordenação *merge sort*, que por sua vez consiste em dividir o problema em vários sub-problemas e resolver esses sub-problemas usando recursividade. O objetivo do trabalho é conseguir paralelizar o algoritmo de ordenação, e após isso, comparar o resultado paralelizado com o serial. Como o algoritmo é recursivo, foram utilizadas *tasks* para paralelizar as chamadas de funções e a programação foi realizada utilizando OpenMP. É importante ressaltar que por usar recursividade e um vetor extra para organização, é usado muita memória e tempo de execução. O resultado paralelizado foi, inicialmente, melhor que o serial, com o tamanho de vetor igual a um milhão e *chunk* igual a quinhentos, é possível obter resultados diferentes mudando essas configurações. Para obter um bom desempenho com a paralelização, é necessário que o *chunk* seja igual ao tamanho do vetor divido por duzentos, o que significa que a partir de 0,5% do tamanho do vetor, a paralelização passa a ser uma boa opção para ganho de desempenho em tempo de execução. Os ganhos só foram relativemente bons ao usar uma baixa quantidade de threads.

Palavras-chave: Merge Sort, OpenMP, Processamento Paralelo.

Sumário

1 Introdução

O *merge sort* é uma ordenação por mistura, algoritmo de ordenação do tipo dividir-para-conquistar criado por John von Neumann em 1945. Basicamente ele consiste em dividir o problema em sub-problemas e unir as resoluções dos sub-problemas. Como o *merge sort* usa recursividade, acaba não sendo muito eficiente devido ao alto consumo de memória e tempo de execução. A intenção desse trabalho é paralelizar o *merge sort,* e após isso, comparar os resultados obtidos paralelamente com os sequencialmente. Será feito uma breve explicação do funcionamento do *merge sort*. Depois a análise do algoritmo através do *gprof* , verificando o tempo que cada função leva para ser executada, para poder então saber como paralelizar o algoritmo.

1.1 Objetivos

Esse trabalho tem como por objetivo avaliar o desempenho do algoritmo de ordenação *merge sort*, mostrando os resultados e as etapas até conseguir paralelizar o código.

2 Revisão Bibliográfica

"O *merge sort* é definido como muito previsível [1]". Segue abaixo propriedades do mesmo.

- Estável.
Ou seja, tanto no pior como no melhor caso, a demora para a conclusão da ordenação será a mesma.

- Preciso no mínimo o dobro de memória.
Devido ao vetor auxiliar utilizado.

- O(n.log(n)) em tempo de execução.
No pior caso.

- Não é adaptativo.
Ou seja, suas decisões de rotas são tomadas na inicialização e não sofrem alteração após isso.

- Não precisa de acesso aleatório a dados.
Acesso em tempo constante à uma posição de memória.

Seguindo esses pré supostos, podemos analisar que o algoritmo é uma boa escolha para uma variedade de situações. Como quando é preciso uma estabilidade na ordenação, ou quando estamos trabalhando com listas encadeadas. Porém requer um espaço extra de memória para realizar sua aplicação.

3 Desenvolvimento do Trabalho

Após o algoritmo ser entregue, ele foi compilado utilizando o *gprof* e analisado para verificar aonde deveria ser feita a paralelização, nas primeiras tentativas foi tentado ser paralelizado foi a função *intercala*, que por sua vez conquista e divide os sub-problemas. Porém como essa função é chamada recursivamente, foi visto em aula juntamente com a professora que o melhor para esse problema seria resolvê-lo usando *tasks*, no começo foi difícil de ser implementado, pois mesmo usando *tasks*, não havia ganho de desempenho em comparação com a execução serial. Por fim foi corrigido usando um *chunk* para verificar quando deveria ser chamada as funções em área paralela. O número de threads utilizados foram igual ao número de núcleos do processador, no caso foi utilizado um Intel® Core™ i5 CPU 650 @ 3.20GHz × 4,. Ou seja, 4 threads foram utilizadas.

O ganho de desempenho com programação paralela para esse problema é relativo ao tamanho do vetor e ao *chunk* utilizado.

No Funcionamento, é mostrado o funcionamento do algoritmo e como ele pode ser paralelizado.

Na análise é mostrado os resultados obtidos pelo *gprof* e também os códigos utilizados para o processo de paralelização. É importante ressaltar que o código foi testado até trinta vezes e os resultados foram os esperados. Praticamente uma constante diminuição do tempo de execução do paralelo para o serial.

3.1 Funcionamento

Funcionamento do algoritmo:

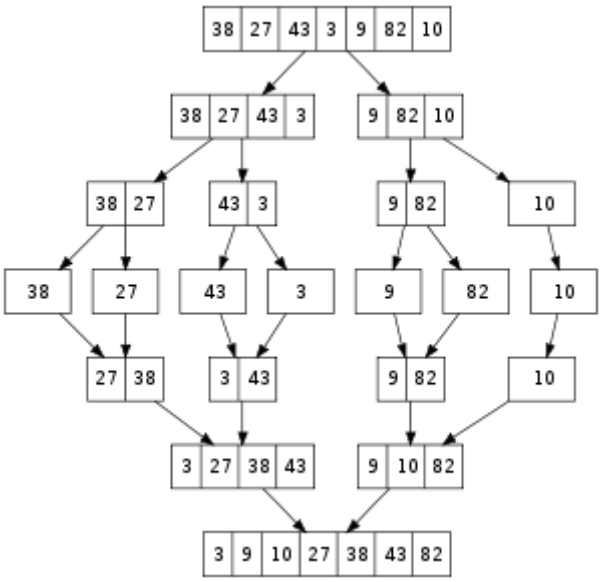


Figura 1. Funcionamento

Podemos analisar que primeiramente ele divide o problema em sub-problemas recursivamente, e após todos estarem separados em sub-problemas diferentes, ele então os vai conquistando e combinando até deixar o vetor organizado.

3.2 Análise

Resultado obtido pelo *gprof* em serial.

% Time	Cumulative Seconds	Self Seconds	Calls	Self ms/call	Total ms/call	Name
100.00	0.02	0.02	99999	0.00	0.00	intercala
0.00	0.02	0.00	1	0.00	0.00	imprimi
0.00	0.02	0.00	1	0..00	20.00	mergesort

Tabela 1. Resultado gprof

Maior tempo na função intercala, pois ela é a que divide, conquista e combina os sub-problemas.

É possível analisar que é criado uma *task* para cada chamada recursiva da mergesort e também uma *task* que precisa esperar o resultados das outras para a função de intercala. Além disso, é visto que só é realizada em área paralela, caso respeite a condição de que o tamanho do vetor seja maior que o *chunk*, sendo assim, possui ganho de desempenho em relação à serial.

4 Resultados

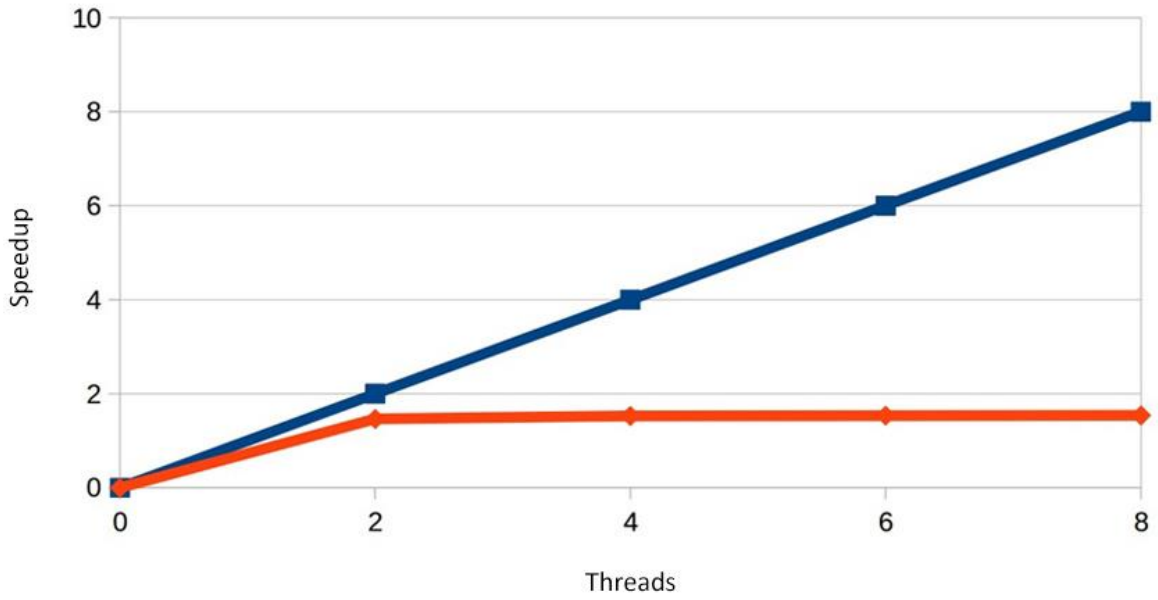
Com o uso de um processador Intel® Core™ i5 CPU 650 @ 3.20GHz × 4, com 4 threads, foram obtidos os resultados abaixo:

Tempo de computação em serial: 0.246850 s

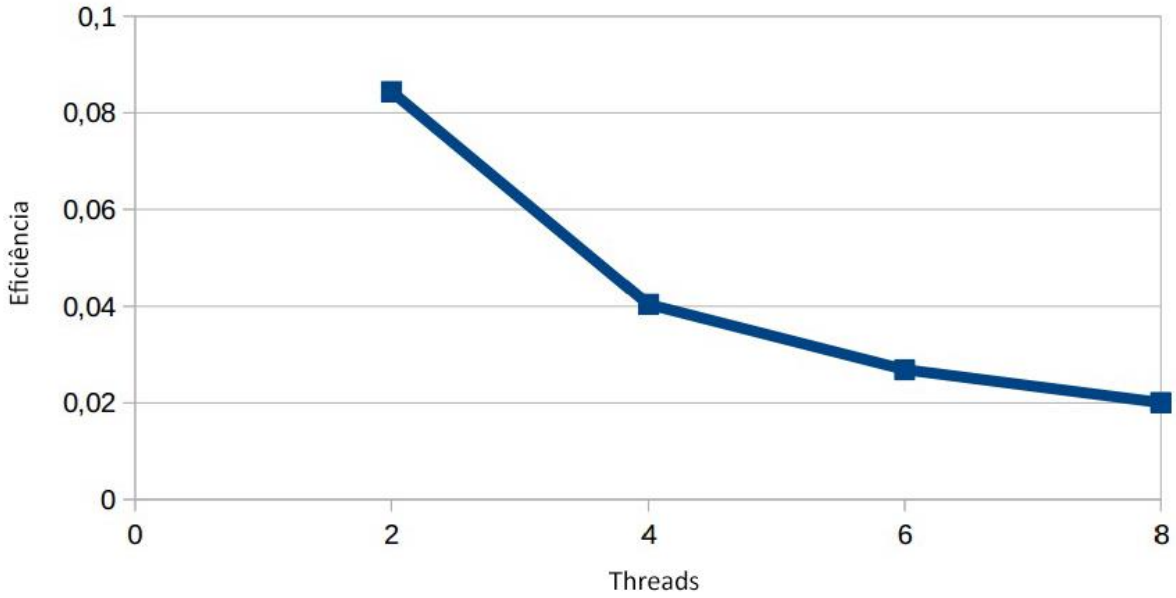
Tempo de computação em paralelo: 0.166151 s

4.1 Speedup

Chunk igual a 500 e tamanho do vetor igual a 1000000.



4.2 Eficiência



O algoritmo apresenta inconsistência em relação ao tamanho do vetor/*chunk*, por causa disso, após X número de threads, não é mais obtido uma eficácia na computação. A partir desse ponto, o algoritmo se torna praticamente constante.

5 Conclusão

Como visto nos resultados obtidos, o desempenho da paralelização para esse algoritmo é relativo ao tamanho do vetor x *chunk*. O *speedup* deve ser levado em consideração no momento em que deve-se paralelizar o *merge sort*, pois por ser um algoritmo recursivo e que necessita de um vetor extra de memória, acaba se tornando inviável. Porém deve-se ressaltar que para poucas processadores é significante a eficácia obtida.

De fato, inicialmente, o resultado obtido pelo algoritmo paralelo foi melhor que o sequencial em tempo de computação, atingindo um tempo em torno de 0,8 segundos a menos que o serial.

A paralelização do código foi feita após cerca de duas tentativas, onde foi tentando paralelizar os laços na função intercala, e foi tentada utilizar *sections* para cada chamada de função.

Após desapontamentos no desempenho em relação ao sequencial, o subterfúgio foi utilizar *tasks* para resolver o problema, que de fato, serviu adequadamente em conjunto com o uso de um *chunk*.

Em contrapartida, o trabalho ajudou a fortalecer o entendimento da programação paralela utilizando OpenMP, além disso, foi de suma importância o auxílio da professora durante a realização do mesmo, onde foi explicado o funcionamento de uma arquitetura paralela e métodos de programação para memória compartilhada.

6 Referências

SORTING-ALGORITHMS. **Merge Sort**. http://www.sorting-algorithms.com/merge-sort

Apêndice A – Código Fonte

```
void mergesort (int p, int r, int v[]) {
    if (p < r) {
        int q = (p + r)/2;
        mergesort (p, q, v);
        mergesort (q+1, r, v);
        intercala (p, q+1, r, v);
    }
}

void mergesort_parallel (int p, int r, int v[]) {
    if (p < r) {
        int q = (p + r)/2;
        #pragma omp task if(q-p > CHUNK)
        mergesort_parallel (p, q, v);
        #pragma omp task if(r-(q+1) > CHUNK)
        mergesort_parallel (q+1, r, v);
        #pragma omp taskwait
        intercala (p, q+1, r, v);
    }
}
```