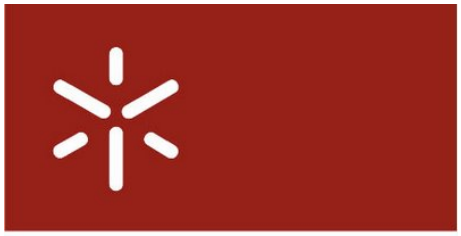
Universidade do Minho

Departamento de Informática

Engenharia de Sistemas de Computação

Heap Sort



Universidade do Minho

António Sérgio Alves Costa A78296
José Pedro Moreira Resende A77486

10 de Junho de 2019

Conteúdo

1 Introdução

Algoritmos de ordenação sao usados para reorganizar e ordenar uma dada lista ou array de acordo com parametros fornecidos por um comparador de elementos.

Existem varios algoritmos de ordenação com diferentes perfis de execucação e memoria, neste artigos vamos falar em especifico do heapsort ,da sua paralelização e da sua caracterização.

O heapsort é um algoritmo de ordenação da familia dos algoritmos de ordenacao por seleção.

Este algoritmo usa uma estrutura de dados heap para os ordenar a medida que os insere de forma a tornar a ordenacao estavel.

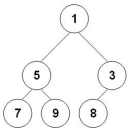
O seu tempo de execução sobre conjuntos ordenados aleatoriamente é muito bom, tem um uso de memória bem comportado e o seu desempenho em pior caso é praticamente igual ao desempenho em caso médio.

2 Algortimo: Heapsort

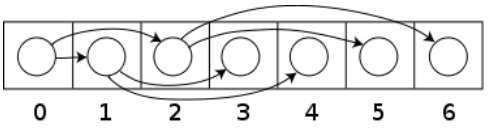
2.1 Sequencial

Nesta implementação do algoritmo heap sort sequencial utilizamos o codigo disponibilizado no enunciado.

O funcionamento deste algoritmo consiste em duas fases. Primeiramente e construido uma heap apartir do array que nos e dado, em segundo é construido o array ordenado retirando o maior numero (root atual) e inserindo no array ordenado. A heap e entao atualizada de forma a manter os invariantes deste tipo de estrutura e eleger um novo root.



Node	1	5	3	7	9	8
Index	0	1	2	3	4	5



Este array ordenado e na verdade feito no mesmo array da heap e simplesmente e alterado os limites final da heap, reaproveitando a memoria, sem ser necessario alocar um novo array para guardar o resultado final.

2.2 Paralelo

2.2.1 Com Locks

Com o intuito de paralelizar e acelarar este algoritmo depois de alguma pesquisa e experimentação chegamos a uma implementação na qual dividimos o array em varias hheaps mais pequenas de tamanho constante (sempre que possivel).

Assim, utilizamos o simples algoritmo sequencial mas com ajuda de locks e de threads paralelizamos o algoritmo. Basicamente, em vez de termos apenas um fio de execução, iremos ter N fios de execução.

Todavia esta solução trouxe-nos bastantes problemas visto que havia data race, e nem sempre a ordem pelo que os siftDowns eram feitos, eram os corretos, acabando assim por se construir um array ordenado diferente do correto.

Para resolver este problema, decidiu-se utilizar locks para termos a certeza que nenhuma thread estava a mexer em algo ao mesmo tempo que outra, e, para além disto, utilizamos schedule static para garantir que a ordem pela qual os siftDowns eram feitos, era a correta, gerando assim o array ordenado correto.

2.2.2 Sem Locks

Após concluirmos a versão anterior, percebemos que os locks e o schedule static poderiam influenciar a performance do algoritmo e decidimos tentar criar um algoritmo de paralelização que não utilizasse estas "ferramentas".

Assim tanto as heaps como os arrays ordenados associados a cada uma destas estruturas podem ser calculados e ordenados de forma independente e paralela.

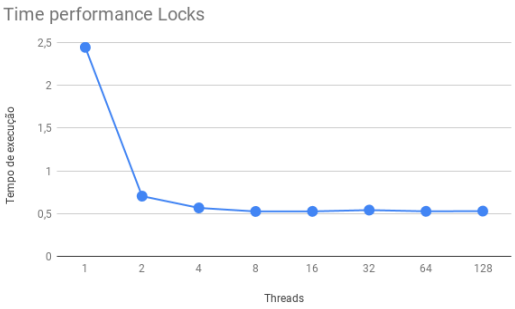
Utilizamos aqui o scheduler dinamico do omp de forma a distribuir melhor a carga de trabalho entre as threads.

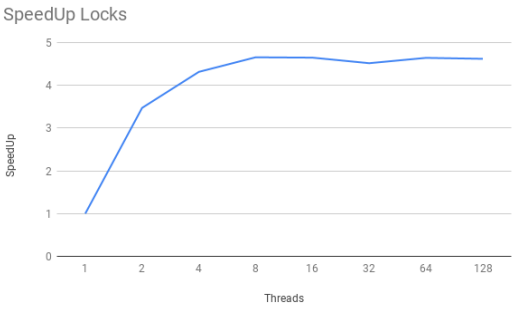
Apos obtermos varios segmentos ordenados e necessario correu o processo de fundir estes segmenetos de forma a obter um unico array ordenado.

Para este trabalho foi usado uma heuristica de redução em arvore o que torna todo o processo de merge extremamente efeciente e possivel de executar de forma paralela, uma vez que temos blocos sem dependencias entre si, tirando partido tambem de um scheduler dinamico de forma a que a carga de trabalho seja distribuida pelas threads.

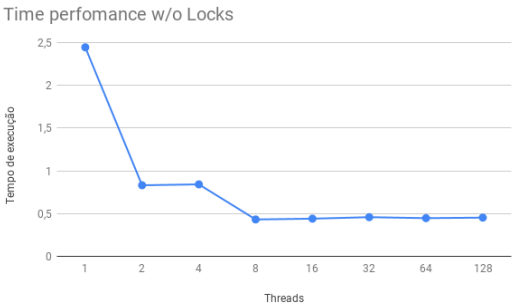
3 Apresentação Resultados

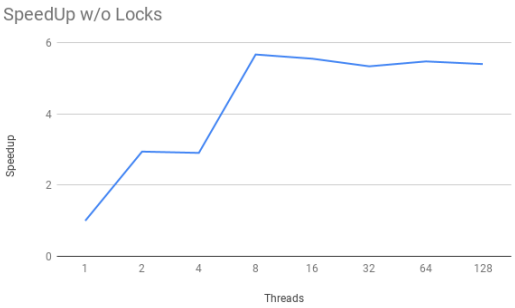
3.1 Locks





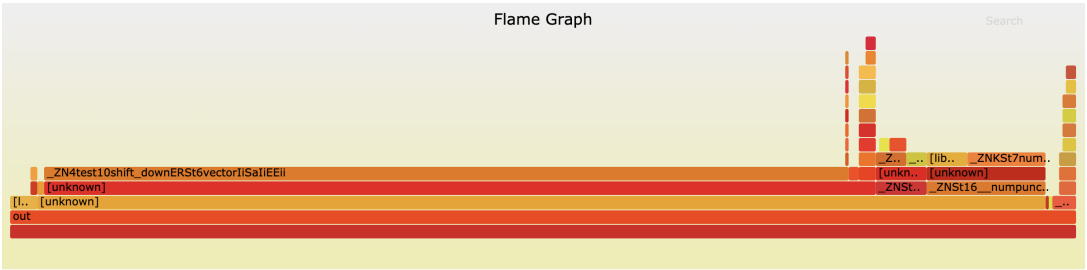
3.2 Sem Locks



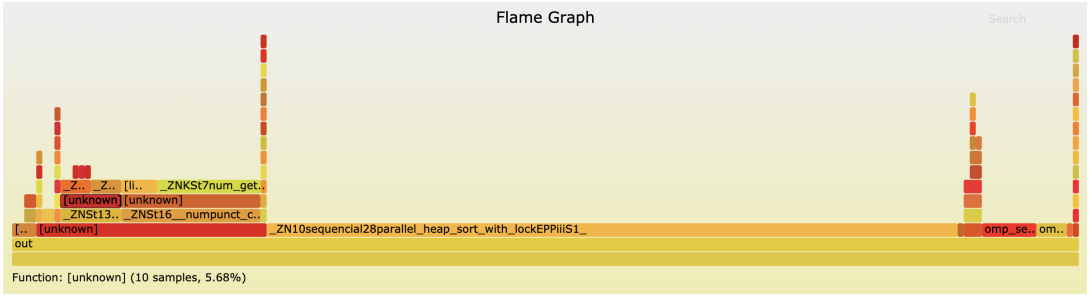


3.3 FlameGraph

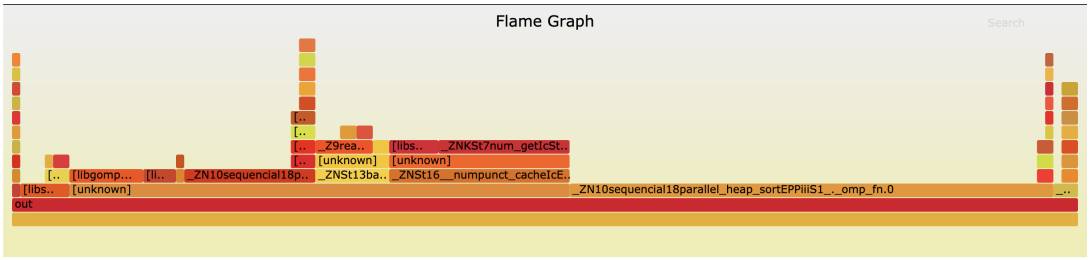
3.3.1 Sequencial



3.3.2 Com Locks



3.3.3 Sem Locks



4 Análise dos Resultados

Após ver os gráficos anteriores conseguimos perceber claramente que a paralelização resulta, em ambos os casos. No entanto, percebemos que o algoritmo sem locks é bastante melhor.

Todavia também se pode reparar que chegando às 8 threads o SpeedUp estabeliza em ambos os casos, o que pode ser explicado por estarmos a utilizar o nosso computador pessoal que apenas tem 4 cores.

No que toca aos FlameGraphs, percebe-se claramenteque a implementação sequencial tem um fio de execução muito monótono quando se compara aos outros dois.

5 Conclusão

A paralelização de algoritmos nem sempre é fácil, e neste trabalho tivemos um exemplo disso.

Heapsort é um algoritmo de ordenação extramamente difícil de paralelizar tendo em conta as suas características. Primeiro, é bastante difícil paralelizar a carga de trabalho de um siftDown sem aumentar bastante a complexidade do código e do algoritmo, por outro lado a paralelização é impossível realizar vários siftDowns ao mesmo tempo sem corroer o resultado correto do algoritmo.

Com o perf conseguimos perceber o comportamento dos vários algoritmos implementados, e perceber as suas diferenças. Conseguimos também perceber o que cada algoritmo utiliza do sistema. Por exemplo, com o perf percebe-se claramente qual é o algoritmo que utiliza locks, visto que procurando pelas funções chamadas após a realização do record, há funções que chamam locks.

Em suma, este projeto serviu para percebemos a complexidadede paralelizar algoritmos que às vezes nos parecem bastante básicos, e ao mesmo tempo para ter mais alguma experiência e perceber a importância de ferramentas de profiling como o perf.