Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Paralela - OpenMP

Algoritmo BucketSort

Autores:

João Vieira	A76516
Manuel Monteiro	A74036

11 de Dezembro de 2018

Conteúdo

1 Introdução

No âmbito da Unidade Curricular de Computação Paralela, durante as aulas laboratoriais, foi-nos lecionado o paradigma de memória partilhada, tendo por base as ferramentas de paralelização de código fornecidas pelo *OpenMP*. Também foram analisados eventuais ganhos nos algoritmos com a utilização desta interface. Posto isto, foi-nos proposto a realização deste trabalho prático, que tem como objetivo a análise de um algoritmo sequencial e a consequente paralelização do mesmo, de modo a atingir soluções mais eficientes e mais rápidas. O algoritmo escolhido pelo nosso grupo para este trabalho foi o algoritmo de ordenação **BucketSort**.

2 Descrição do Algoritmo Sequencial

O algoritmo BucketSort trata de dividir os elementos, distribuídos uniformemente, de um array inicial por vários buckets, em que os elementos menores ficarão em posições de buckets menores e o contrário também. Para tal criámos uma *struct bucket*, que nos serve para guardar os elementos do array, assim como um array auxiliar que é utilizado para inserir e ordenar os elementos de cada bucket. Primeiramente, o algoritmo define o tamanho que cada bucket terá, percorrendo o array original. De seguida calcula o início dos mesmos. Depois faz a travessia no array original para inserir os elementos em cada bucket (array a ser ordenado), e por fim é utilizada a função *qSort* para ordenar cada um.

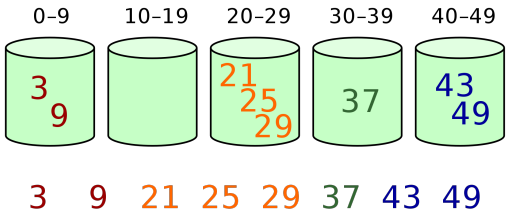

Figura 1: Ilustração do BucketSort

3 Paralelização do Algoritmo

Para a paralelização do algoritmo deparamo-nos com algumas dificuldades, pois este efetua muitas operações nos mesmos dados, que quando são paralelizadas provocam disparidades nos resultados que não são desejadas. Tendo isto em conta, o cálculo do tamanho, início e a inserção de elementos em cada bucket é ainda feita sequencialmente. A diferença é que agora é utilizada mais do que uma thread para ordenar os buckets.

4 Testes e Análise de Resultados

Antes de efetuarmos a análise de resultados, queremos referir que foram realizados vários testes que nos ajudaram a perceber melhor a causa dos resultados obtidos. Mas devido à pequena extensão do relatório, decidimos expor apenas os resultados dos testes mais relevantes.
Em primeiro lugar, os testes foram realizados no nó 652 do *cluster Search*, que tem as seguintes propriedades:

Processador:	
Modelo	Intel® Xeon® Processor E5-2670v2
Arquitetura	Ivy Bridge
#cores	10
#Threads	20
Frequência de Clock	2.5 GHz
Memória:	
Cache L1 (*per core*)	32KB de Instruções, 32KB de Dados
Cache L2 (*per core*)	256KB
Cache L3 (*shared*)	25MB
RAM	64GB

Tabela 1: Especificações do Nó.

Nos nossos testes foram utilizados os seguintes tamanhos de *array*, de modo a ocupar os vários níveis de memória:

Tamanho	**Nível de Memória**
3.000	Cache L1
30.000	Cache L2
300.000	Cache L3
3.000.000	Cache L3
30.000.000	RAM

Tabela 2: Tamanho dos *inputs*.

Para **todos** os testes tivemos em conta os seguintes fatores:

- A medição do tempo foi feita em micro segundos.
- A execução do algoritmo é realizada 20 vezes para cada teste.
- A cache foi limpa entre as execuções do algoritmo.

No início calculamos o máximo ganho teórico através da *lei de Amdahl*. Notámos que quanto maior o volume de dados, menor é a percentagem sequencial do algoritmo, indicando logo o aumento de *speed-up* possível.
Para os testes com o número de threads igual ao número de buckets, e com uma gama de valores até 99 999, obtivemos os seguintes resultados:

Figura 2: Gráfico de ganhos.

4.1 Análise

Iremos agora apresentar a análise dos resultados para o tamanho do *input* de dados, feita até 32 threads:

4.1.1 Array de tamanho 3.000

Para um *array* com o tamanho menor, grande parte da execução do programa é ainda sequencial, o que limita imenso o *speed-up*. Verificamos assim que o ganho obtido é mínimo, sendo que até é mais lento na execução com 2 e 32 threads. Também existe o fator de que o overhead da paralelização é superior aos ganhos da mesma.

4.1.2 Array de tamanho 30.000

O ganho da paralelização para este tamanho, é também ainda muito limitada pela fraca paralelização do programa, apesar disso já podemos ver um aumento no ganho com o número de threads, apesar de ser muito reduzido.

4.1.3 Array de tamanho 300.000

A partir deste valor notámos que a parte paralela já tem mais *workload* o que se transmite no gráfico com um ganho logo nas 2 threads. Com maior número de threads e consequentemente buckets existe uma melhor distribuição de valores nos buckets.

4.1.4 Array de tamanho 3.000.000

Para este tamanho já se verifica que o *workload* de cada thread é suficiente para haver aumentos mais significativos com aumento de threads. Para 32 threads já existe o *overhead* de paralelismo, daí o *speed-up* diminuir.

4.1.5 Array de tamanho 30.000.000

Aqui continua a haver uma boa distribuição de trabalho por thread e isso nota-se no aumento contínuo de ganho. Porém reparamos que para o tamanho acima referido, até 16 threads, o ganho é superior. Isto pode-se dever ao facto de haver muitos mais números para processar. Mas existe outro fator a ter em conta, o número de acessos à memória. Desta forma faz todo o sentido que o ganho seja inferior, para um tamanho superior vai haver mais iterações e consequentemente mais acessos à memória.

4.1.6 Teste do #buckets com 32 Threads

Também realizamos testes em que fizemos variar o **número de buckets**, numa gama de valores até 1024 buckets.
Num primeiro teste, que diz respeito ao primeiro gráfico que se encontra na secção Anexos, fixamos o número de threads em 32.
Como se pode verificar, excetuando para um tamanho de 3000, até 32 buckets há sempre um aumento de ganho similar aos ganhos verificados no teste principal, que variava até as 32 threads.
A partir desse número até 1024 buckets, apesar de continuar a haver ganhos, estes vão diminuindo. Isto deve-se ao facto de usarmos um *loop scheduling* estático, que poderá dar origem a casos em que há threads que trabalham buckets com muitos elementos e outras que poderão ter buckets com poucos elementos ou até vazios, provocando um desbalanceamento de carga nas threads.
Também podemos concluir que, quanto maior o número de buckets, maior será a percentagem de trabalho sequencial realizado pelo algoritmo, já que como há um grande número de buckets, a sequência paralela não vai ser aproveitada pois maior parte dos buckets já vão estar ordenados.

5 Otimização

6 Conclusão

Com o finalizar deste trabalho prático, verificámos que a gestão do paralelismo da memória num elevado número de cores é desafiante, pois enfrentamos dificuldades na otimização da versão paralela do algoritmo. Apesar disso, podemos dizer que devido à fraca escalabilidade do programa, este é vantajoso apenas para *array's* de elevadas dimensões.
Em suma, estas máquinas requerem uma maior atenção por parte do programador, pois existem vários fatores que influenciam o *speed-up*, sendo este maior quanto melhor for a implementação dos programas.

7 Anexos

7.1 Teste do #buckets com 32 threads

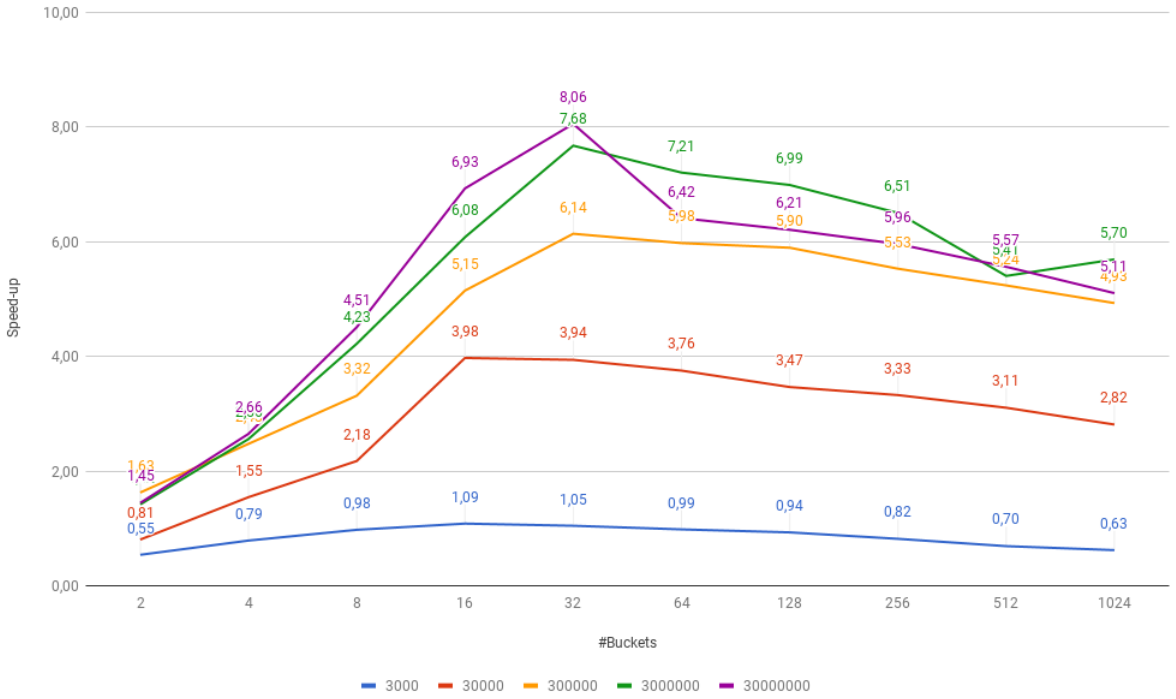

Figura 3: Gráfico de ganhos por #buckets.