

**Mestrado em Engenharia Informática**

*Algoritmos Paralelos - 2014/2015*

BubbleSort Paralelizado

10 de Março de 2015

**Fábio Gomes** pg27752

# **Índice**

[**Índice** 2](#_Toc413723457)

[Introdução 3](#_Toc413723458)

[Caracterização do Sistema 4](#_Toc413723459)

[Nodo do Cluster obtido 4](#_Toc413723460)

[Explicação do Problema 5](#_Toc413723461)

[Análise do Código Fornecido 6](#_Toc413723462)

[Paralelização com OpenMP 7](#_Toc413723463)

[Código BubbleSort Paralelizado 9](#_Toc413723464)

[Testes e Análise de Resultados 10](#_Toc413723465)

[Conclusão 11](#_Toc413723466)

# Introdução

O problema que nos foi apresentado está relacionado com o algoritmo de ordenação BubbleSort normalmente conhecido como o pior algoritmos de todos do género. É nosso trabalho tentar paraleliza-lo.

Há que ter em atenção pois não se trata de uma simples paralelização com pragma omp parallel tradicional, temos que tratar de casos de sincronismo e concorrência de dados.

# 

# Caracterização do Sistema

## Nodo do Cluster obtido

Para utilização deste projeto tivemos que recorrer ao SEARCH, que tem a seguinte especificação.

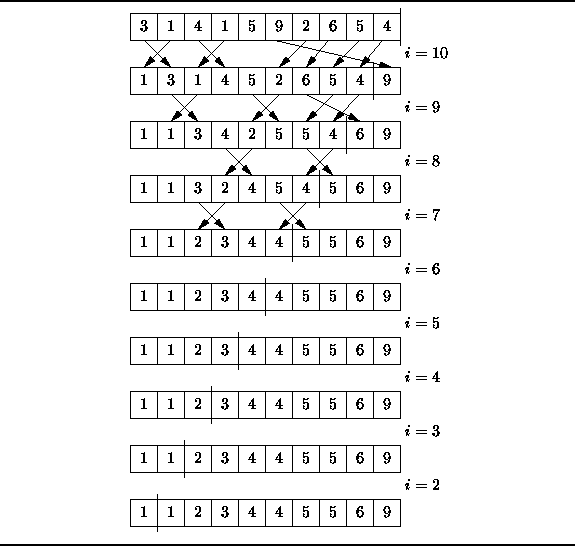
|  |  |
| --- | --- |
|  | **Cluster Node** |
| **Manufacturer** | intel |
| **Model** | e5-2695v2 |
| **Clock speed** | 2.4 GHz |
| **Architecture** | x86-64 |
| **µArchitecture** | Ivy Bridge |
| **#Cores** | 12 |
| **#Threads per Core** | 2 |
| **Total Threads** | 24 |
| **Peak FP performance** | 38.4 GFLOP/s |
| **L1 cache** | 768kB (32kB L1 Data per Core) |
| **L2 cache** | 3MB (256kB per Core) |
| **L3 cache** | 30,720 kB (Shared) |
| **Main Memory** | 66,068,588 kB |
| **Memory Channels** | 4 |
| **Max Memory** | 786,432 MB |
| **Max Bandwidth** | 59,733.32 MB/s |

# 

# Explicação do Problema

O BubbleSort é uma técnica básica em que se comparam dois elementos do vetor/*array* e trocam-se as suas posições se o primeiro elemento é maior do que o segundo. São feitas várias passagens pela tabela e em cada uma, comparam-se dois elementos adjacentes tal que se estes elementos estiverem fora de ordem, eles são trocados segundo o critério previamente explicado. Podendo ser trocado conforme se pretenda de forma Ascendente ou Descendente.

Este método tem como vantagem a sua simplicidade e a estabilidade mas peca por ser muito lento. Deve ser usado para tabelas muito pequenas ou quando se sabe que a tabela está quase ordenada. O nome vem do método da troca em que os elementos menores (mais “leves”) vão aos poucos “subindo” para o início da tabela, como se fossem bolhas. Exemplificado na figura seguinte.



Exemplo para um array de 10 elementos -

# Análise do Código Fornecido

O programa é bastante simples, é gerado um vetor a de tamanho n e para cada posição dele é percorrido até ao fim fazendo as trocas. Terminando apenas quando todos os índices estiverem percorridos.

/\*-----------------------------------------------------------------

\* Function: Bubble\_sort

\* Purpose: Sort list using bubble sort

\* In args: n

\* In/out args: a

\*/

void Bubble\_sort**(**

int a**[]** /\* in/out \*/**,**

int n /\* in \*/**)** **{**

int list\_length**,** i**,** temp**;**

**for** **(**list\_length **=** n**;** list\_length **>=** 2**;** list\_length**--)**

**for** **(**i **=** 0**;** i **<** list\_length**-**1**;** i**++)**

**if** **(**a**[**i**]** **>** a**[**i**+**1**])** **{**

temp **=** a**[**i**];**

a**[**i**]** **=** a**[**i**+**1**];**

a**[**i**+**1**]** **=** temp**;**

**}**

**}** /\* Bubble\_sort \*/

# Paralelização com OpenMP

A Paralelização não é tão trivial como poderá parecer porque há concorrência nos dados. Para tal é necessário usar um novo mecanismo do OpenMP chamado omp\_lock\_t. Como o nome indica é feito um lock a um elemento de forma a que apenas 1 thread o consiga ter em sua posse. Assim, crio um array de locks chamado lock de tamanho igual ao número de threads.

lock **=** **(**omp\_lock\_t**\*)** malloc**((**nthreads**)\*sizeof(**omp\_lock\_t**));**

Com ele já teremos mais controlo no vetor para ordenarmos sem problemas.

É preciso ainda definir uma subdivisão do vetor, chamada de bloco. O tamanho de cada bloco block\_size é igual ao tamanho do vetor a dividir pelo número de locks (igual ao número de threads). Cada bloco é associado um lock, assim apenas 1 thread poderá fazer as trocas à vontade sem se preocupar com concorrência pois é apenas ela que lá está.

Com o decorrer do tempo o lock será libertado e o seguinte será adquirido, o que foi libertado será apanhado por uma thread que estava à espera para continuar a sua iteração. Desta forma haverá uma sequência de threads ao longo do vetor sem nunca se cruzarem causando conflitos.

Em termos de código, é iniciado o vetor de locks, algumas variáveis de condição e o tamanho do bloco.

double start **=** omp\_get\_wtime**();**

int list\_length**=**n**,** i**,**x**,**ind**,** temp**,**ltemp**,**max**;**

block\_size **=** **(**int**)(**list\_length**/**nthreads**);**

**for(**i**=**0**;**i**<**nthreads**;**i**++)**

omp\_init\_lock**(&(**lock**[**i**]));**

int ordered **=** 0**,**changed **=** 0**;**

Depois começa o ciclo principal com um pragma omp parallel para criar a região paralela com a particularidade de partilhar, obviamente, o vetor a ser ordenado, o tamanho da lista que vai decrescendo ao longo das iterações, o vetor de locks e a condição ordered que nos vai parar o ciclo principal quando 1 thread chegar ao fim sem fazer uma troca sugerindo que está o vetor ordenado poupando tempo.

#pragma omp parallel shared(a,lock,n,list\_length,ordered) num\_threads(nthreads) private(temp,i,ltemp,ind) firstprivate(block\_size,changed)

**while(!**ordered**){**

changed **=** 0**;**

Já dentro do ciclo while temos que percorrer os blocos do vetor e em cada um fazer lock. Depois apenas iteramos elementos naquelas posições até ao fim do bloco, ou do vetor quando estamos no último bloco. Por fim libertamos o lock e verificamos se fizemos alterações para parar o ciclo while e decrementamos 1 ao tamanho do vetor para que a próxima thread não vá até ao fim de todo pois é inútil.

**for(**ind**=**0**;**ind**<**nthreads**;**ind**++){**

omp\_set\_lock**(&(**lock**[**ind**]));**

max **=** list\_length**>(**ind**+**1**)\***block\_size **?** **(**ind**+**1**)\***block\_size **:** list\_length**;**

**for** **(**i **=** ind**\***block\_size**;** i **<** max**;** i**++){**

**if** **(**a**[**i**]** **>** a**[**i**+**1**])** **{**

temp **=** a**[**i**];**

a**[**i**]** **=** a**[**i**+**1**];**

a**[**i**+**1**]** **=** temp**;**

changed **=** 1**;**

**}**

**}**

omp\_unset\_lock**(&(**lock**[**ind**]));**

**}**

**if(**changed**==**0**)**

ordered **=** 1**;**

list\_length**--;**

**}**

Para finalizar destruímos o vetor de locks e contabilizamos o tempo.

**for(**i**=**0**;**i**<**nthreads**;**i**++)**

omp\_destroy\_lock**(&(**lock**[**i**]));**

double end **=** omp\_get\_wtime**();**

printf**(**"%3.2gn"**,**end**-**start**);**

# Código BubbleSort Paralelizado

void Bubble\_sort**(**

int a**[]** /\* in/out \*/**,**

int n /\* in \*/**)** **{**

double start **=** omp\_get\_wtime**();**

int list\_length**=**n**,** i**,**x**,**ind**,** temp**,**ltemp**,**max**;**

block\_size **=** **(**int**)(**list\_length**/**nthreads**);**

**for(**i**=**0**;**i**<**nthreads**;**i**++)**

omp\_init\_lock**(&(**lock**[**i**]));**

int ordered **=** 0**,**changed **=** 0**;**

#pragma omp parallel shared(a,lock,n,list\_length,ordered) num\_threads(nthreads) private(temp,i,ltemp,ind) firstprivate(block\_size,changed)

**while(!**ordered**){**

changed **=** 0**;**

**for(**ind**=**0**;**ind**<**nthreads**;**ind**++){**

omp\_set\_lock**(&(**lock**[**ind**]));**

max **=** list\_length**>(**ind**+**1**)\***block\_size **?** **(**ind**+**1**)\***block\_size **:** list\_length**;**

**for** **(**i **=** ind**\***block\_size**;** i **<** max**;** i**++){**

**if** **(**a**[**i**]** **>** a**[**i**+**1**])** **{**

temp **=** a**[**i**];**

a**[**i**]** **=** a**[**i**+**1**];**

a**[**i**+**1**]** **=** temp**;**

changed **=** 1**;**

**}**

**}**

omp\_unset\_lock**(&(**lock**[**ind**]));**

**}**

**if(**changed**==**0**)**

ordered **=** 1**;**

list\_length**--;**

**}**

**for(**i**=**0**;**i**<**nthreads**;**i**++)**

omp\_destroy\_lock**(&(**lock**[**i**]));**

double end **=** omp\_get\_wtime**();**

printf**(**"%3.2gn"**,**end**-**start**);**

**}** /\* Bubble\_sort \*/

# Testes e Análise de Resultados

Com tudo concluído, é tempo de fazer medições de tempos de execução e analisar os resultados.

Os testes foram feitos para uma divisão de blocos igual ao número de threads apenas e com número de threads e de tamanho variado como sugere a seguinte tabela.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | #Threads | | | | | | | | | | |  |
| Size | **Serial** | **2** | **4** | **6** | **8** | **10** | **14** | **16** | **20** | **30** | **40** | Min |
| **100** | **0,002** | 0,0002 | 0,0003 | 0,0003 | 0,0004 | 0,0007 | 0,0006 | 0,0006 | 0,0008 | 0,0012 | 0,0065 | 0,0002 |
| **500** | **0,004** | 0,0008 | 0,0018 | 0,0018 | 0,0019 | 0,003 | 0,0022 | 0,002 | 0,0021 | 0,0026 | 0,0079 | 0,0008 |
| **1000** | **0,008** | 0,0035 | 0,0043 | 0,0047 | 0,0054 | 0,0086 | 0,0047 | 0,005 | 0,0048 | 0,0047 | 0,016 | 0,0035 |
| **5000** | **0,092** | 0,069 | 0,071 | 0,073 | 0,068 | 0,067 | 0,062 | 0,05 | 0,044 | 0,041 | 0,036 | 0,036 |
| **10000** | **0,365** | 0,18 | 0,28 | 0,27 | 0,25 | 0,22 | 0,18 | 0,16 | 0,14 | 0,11 | 0,13 | 0,11 |
| **50000** | **8,905** | 7,5 | 6,7 | 6,6 | 6,5 | 6,4 | 3,2 | 3 | 2,4 | 2,3 | 1,9 | 1,9 |
| **100000** | **30,135** | 17 | 27 | 26 | 24 | 14 | 13 | 11 | 9,9 | 7,9 | 7,5 | 7,5 |
| **200000** | **122,07** | 90 | 110 | 120 | 90 | 67 | 56 | 44 | 42 | 31 | 21 | 21 |

A Verde estão representados os tempos mais baixos (melhores) e a Vermelho os maiores (piores).

Conseguimos traçar um padrão, a versão Serial fornecida consegue ser sempre pior para estes casos de teste mas prevejo que para valores mais baixos de input isto não ocorra.

Como seria de esperar quanto maior o problema em termos de tamanho do vetor mais tempo irá durar, obrigando a um aumento do número de threads para que o seu tempo de execução não aumente também de forma explosiva.

Todos os tempos medidos são abaixo do Serial o que me satisfaz mas muito provavelmente será possível obter tempos muito melhores porque há algumas lacunas na minha implementação que não foram corrigidas, nomeadamente quando uma thread está à espera de um lock para um bloco seguinte mas esse bloco já não é preciso pois o tamanho da lista (virtual que decresce a cada thread concluída) já é mais baixo que esse. Fazendo com que seja uma perca de tempo esperar pelo lock que não é necessário de todo.

# Conclusão

Para além de ser um caso prático e cativar por si só foi também um trabalho muito bom na medida em que permitiu usar novos conhecimentos adquiridos nesta Unidade Curricular que não foram abordados anteriormente. Um trabalho que apesar de parecer simples engloba em si vários aspectos que tiveram de ser devidamente considerados para que o algoritmo funcionasse corretamente.

No final dou o trabalho por concluído mas sem deixar a nota que poderá ser melhorado como já referi. Já tínhamos utilizado locks em Sistemas Distribuídos mas nesta aplicação prática revelaram-se muito mais críticos e obrigaram a uma maior preparação e análise do problema.