

Analysis of Parallel Sorting by Regular Sampling

Bruno Miguel da Silva Barbosa
Universidade do Minho
Departamento de Informática
Braga, Gualtar, 4710-057
a67646@alunos.uminho.pt

RESUMO

Este trabalho prático desenrola-se no âmbito da unidade curricular de Algoritmos Paralelos e tem como principal objetivo efetuar uma análise não só a nível de escalabilidade, mas também ao nível performance e complexidade de um algoritmo: ordenação paralela através de amostragem regular. Assim sendo, esta investigação foi planeada de forma a incluir, primeiramente, uma análise teórica do algoritmo, seguindo depois para a experimentação em MPI (em memória distribuída) e por último, os resultados práticos obtidos bem como algumas conclusões que foram possíveis retirar.

Palavras-chave

Performance, Concorrência, Ordenação, Processos, Segmentos, Algoritmo, Comunicação, Escalabilidade, Complexidade.

1. ANÁLISE TEÓRICA DO ALGORITMO

O raciocínio por detrás do algoritmo de ordenação paralela através de amostragem regular é bem fácil de compreender e pode ser resumido em três simples etapas:

1. Dividir os dados em K segmentos de tamanho igual.
2. Ordenar os segmentos individualmente, no entanto de forma concorrentemente recorrendo a um algoritmo de ordenação, por exemplo, o *QuickSort*.
3. Fundir os K segmentos de maneira a que no final, os dados estejam, na totalmente ordenados.

As Figuras 1, 2 e 3 ilustram cada umas das etapas anteriormente apresentadas.

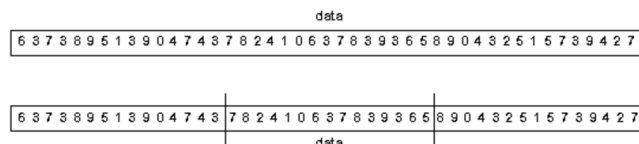


Figura 1. Partição dos dados em K segmentos de tamanho igual

Nesta primeira etapa, são percorridos todos os elementos do *array* inicial e faz-se uma partição do mesmo em concordância com um valor K (número de partições) introduzido pelo utilizador. Deste modo, complexidade nesta etapa é

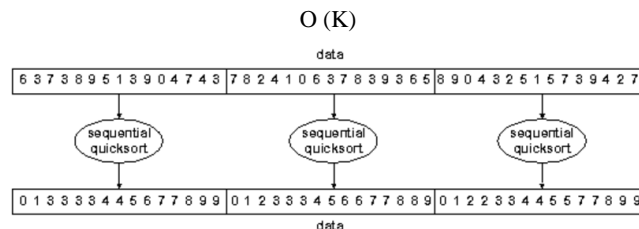


Figura 2. Ordenação concorrente dos vários segmentos através do algoritmo QuickSort

Na segunda etapa, os segmentos definidos na fase anterior são ordenados em paralelo, logo, a complexidade é semelhante à da execução de um *QuickSort* sequencialmente. A diferença é que em vez de termos um conjunto de dados de tamanho N, temos K segmentos de tamanho N/K. Sendo assim, a complexidade deduz-se como se fosse unicamente para um segmento, já que todos eles são processados em paralelo.

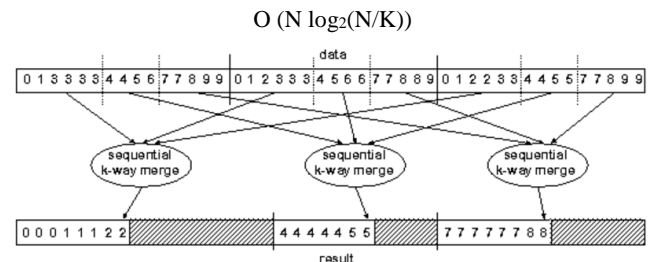


Figura 3. Fundição dos K segmentos já ordenados e divididos em gamas de valores segundo K pivots

De referir que entre os estados das Figuras 2 e 3 existem outros procedimentos subentendidos onde é efetuada uma pivotação em cada segmento, de acordo com o número de partições, que vai determinar K pivots essenciais para permitir que a fundição seja realizada em paralelo. Portanto, restringido ao estágio apresentado na Figura 3, a sua complexidade é

$$O(N \log_2(K))$$

Na sua totalidade, o algoritmo de ordenação paralela através de amostragem regular tem uma complexidade de

$$O(N / p \log(N)), \text{ para } N > p^3$$

Onde N representa o tamanho dos dados (número de inteiros do *array*) e p representa o número de processos pelos quais os dados serão repartidos.

Segundo a Referência 1, existem alguns valores de referência em termos de ganhos de performance, contudo, testados no contexto de memória partilhada.

Speedup(100 million, 2) = 1.9 (96%),
Speedup(100 million, 4) = 3.5 (88%),
Speedup(100 million, 8) = 5.7 (71%),
Speedup(100 million, 16) = 8.1 (51%),
Speedup(100 million, 32) = 10.2 (32%),
Speedup(100 million, 64) = 11.6 (18%).

Figura 4. Valores de referência para ganhos de performance em memória partilhada

Pela Figura 4 conseguimos verificar que, pelo menos, até 64 fios de execução, é possível obter ganhos de *performance* mantendo o tamanho (considerável) dos dados fixo. O maior ganho registado foi de 11.6 e servirá, inicialmente, como um ponto de referência a

atingir. A partir desta informação e da natureza do algoritmo (com poucas dependências de dados), espera-se que a escalabilidade seja quanto melhor quanto maior for o número de processos.

Em termos de custos de comunicação, o maior impacto está associado ao facto de se enviarem os vários segmentos para os respectivos processos e depois fundi-los novamente, mas de forma a garantir que, no final, o *array* inicial esteja ordenado. Será uma boa oportunidade para verificar como se desenrola relação entre o número de processos criados e o tempo de comunicação gasto à medida que se aumenta o número de processo. Por exemplo, quando o número de processo é baixo, serão enviados segmentos de dados maiores. Da mesma forma que quando estiverem a ser usados vários processos os segmentos serão menores, mas está-se a lidar com mais processos. Nas próximas secções vamos poder concluir qual será a melhor relação entre estas duas variantes.

No que diz respeito ao balanceamento de carga não devem haver grandes desequilíbrios dado que a partição é feita de modo a que cada processo receba um segmento de tamanho uniforme, isto é, de igual tamanho. Quanto muito, o que pode ter algum impacto no balanceamento de carga neste caso, é o facto de existir um segmento, que por sorte (pois a inicialização do *array* de entrada é um procedimento aleatório), ficou na sua maioria, já ordenado, precisando assim de menos iterações para corrigir a ordenação.

2. EXPERIMENTAÇÃO MPI

Antes de começar com a experimentação propriamente dita, vai-se definir o ambiente onde os testes vão ser realizados. Ou seja:

- Os testes são realizados nas máquinas 641, reservando 2 nós com 32 processadores cada.
- O tamanho do *array* de entrada será de 100 000 000 elementos (gerados aleatoriamente).
- O número de processos pertence ao seguinte conjunto {1,2,4,8,12,16,20} (justificação nas Figuras 7 e 8).
- Os módulos carregados foram
 - Sequencial – gnu/4.9.0
 - MPI (*Ethernet*) – gnu/openmpi_eth/1.8.4
 - MPI (*Myrinet*) – gnu/openmpi_mx/1.8.4
- O código foi compilado com otimização de nível 3 quer para o código sequencial quer para a versão paralela.
- Os tempos de execução/comunicação foram calculados a partir da mediana de 5 medições.

A primeira versão desenvolvida corresponde a uma versão simplificada do algoritmo genuíno, uma vez que a parte de fundição dos segmentos vindos dos vários processadores é feita em sequencial quando na verdade, devia ser feita em paralelo.

A nível de pseudo-código o ficheiro está organizado segundo a seguinte estrutura:

- Inicialização do *array* de entrada.
- Cálculo do número de elementos por processo.
- Dispersão dos segmentos de dados pelos processos (com a função *MPI_Scatter*).

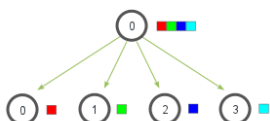


Figura 5. Ilustração da função *MPI_Scatter*

- Ordenação local do segmento de dados respetivo ao processo. Todos os processos executam esta operação em paralelo.
- Reunião de os segmentos enviados para os processos (através da função *MPI_Gather*).

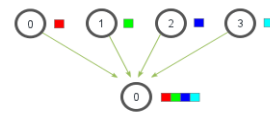


Figura 6. Ilustração da função *MPI_Gather*

- Fusão dos segmentos de forma a que, no final, os dados fiquem todos ordenados. (realizado em sequencial)

Uma experimentação realizada inicialmente permitiu compreender como é que a escalabilidade se comporta à medida que se aumenta o número de processos. Nestes primeiros testes foram utilizadas as *flags* de mapeamento por *core* e por nó. O número de processos variou entre 2 a 64 com números de base 2. Daqui conclui-se que não se tira qualquer partido com a opção de mapeamento por nó pois, os tempos de execução são extremamente mais altos (devido à comunicação) em comparação com o mapeamento por *core*. Outra das conclusões a retirar é que já a partir de 16 processos já começam a aparecer algumas perdas de performance, isto é, o pico de performance está compreendido entre os 8 e os 16 processos. Foi por este motivo que se decidiu ajustar a gama de valores para o número de processos a usar. As Figuras 7 e 8 vêm justificar as declarações previamente enunciadas, destacando as colunas azuis, que simbolizam o tempo total de execução, e as colunas laranja, que representam o tempo despendido em comunicação.

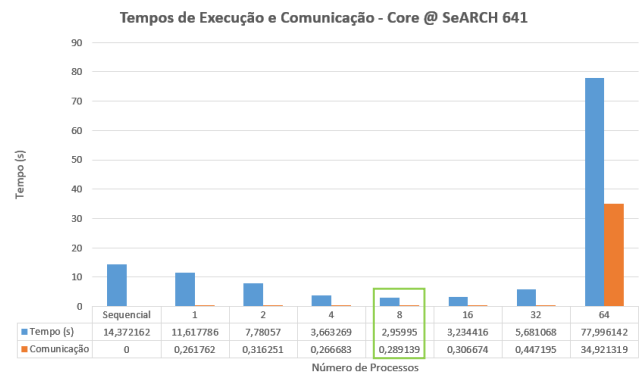


Figura 7. Comparação entre os tempos de execução e comunicação com mapeamento por *core*

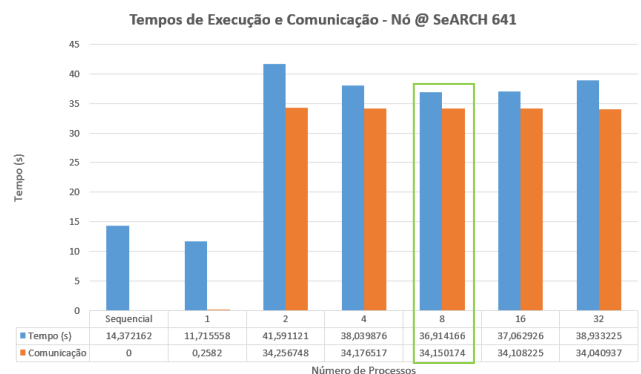


Figura 8. Comparação entre os tempos de execução e comunicação com mapeamento por nó

Para estes testes o código foi compilado com o compilado de tecnologia de comunicação *Ethernet* através do seguinte comando

```
$ mpicc -O3 code.c -o psrs -fopenmp -std=c99
```

E foi executado tal como é apresentado no comando em baixo

```
$ mpirun -mca btl tcp,sm,self --map-by [core  
| node] -np $nprocs psrs
```

3. RESULTADOS PRÁTICOS

A secção dos resultados práticos está dividida em 2 subsecções, uma para a tecnologia de comunicação *Ethernet* (1 Gbps), outra para a tecnologia *Myrinet* (10 Gbps) e por último, uma secção para comparação das duas. Sem dúvida que uma das coisas a comparar será, obviamente, o tempo gasto em comunicação entre as duas tecnologias. Dado que a *Myrinet* tem maior débito, o mais provável é que consiga menores tempos em comunicação.

Os valores inseridos nos gráficos de ganhos de desempenho foram calculados segundo a seguinte fórmula

$$\text{Ganho} = \text{Tempo}_{\text{sequencial}} / \text{Tempo}_{\text{paralelo}}$$

3.1 Ethernet

Os resultados da Figura 9 comprovam que grande parte do tempo de execução é dedicado a processamento dos dados. Para além disso, o menor tempo de execução acontece com 8 processadores.

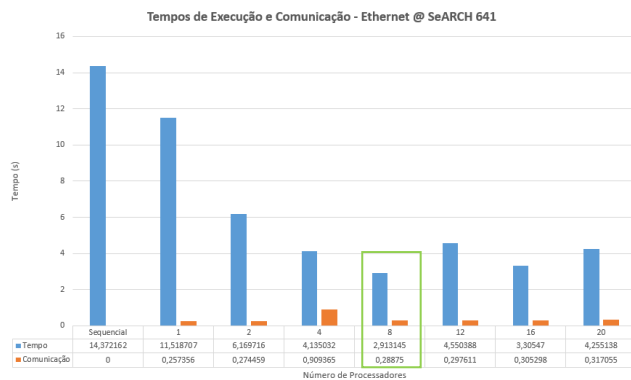


Figura 9. Comparação entre os tempos de execução e comunicação para Ethernet

A Figura 10 vem complementar os resultados obtidos na Figura 9. Aqui, tem-se mais em conta o peso da comunicação durante a execução do programa, registando o maior impacto quando foram usados 4 processadores.

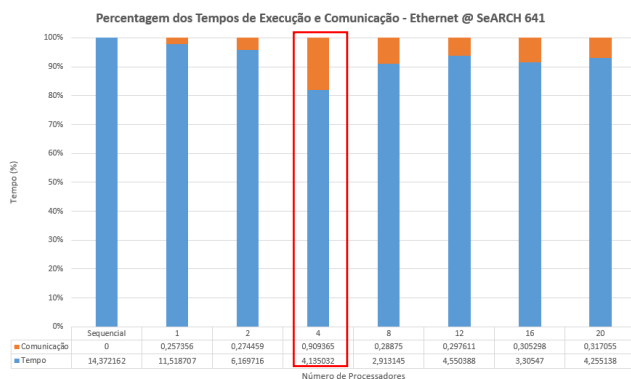


Figura 10. Percentagem dos tempos de execução e comunicação para Ethernet

Em termos de ganhos de *performance*, consegue-se retirar que o pico de desempenho ocorre quando se usam 8 processadores, obtendo um ganho de, aproximadamente, 5 vezes.

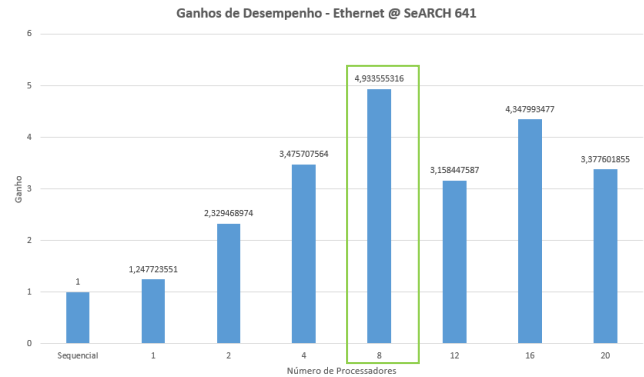


Figura 11. Ganhos de desempenho para Ethernet

3.2 Myrinet

A experimentação com *Myrinet* não correu como seria esperado pois resultou em alguns erros durante a execução do programa. Neste sentido, retirou-se tempos de execução e comunicação até 8 processos e com base neles vão efetuar-se as comparações que forem possíveis.

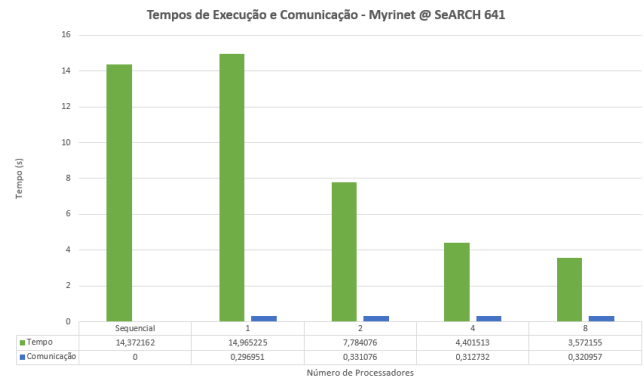


Figura 12. Comparação entre os tempos de execução e comunicação para Myrinet

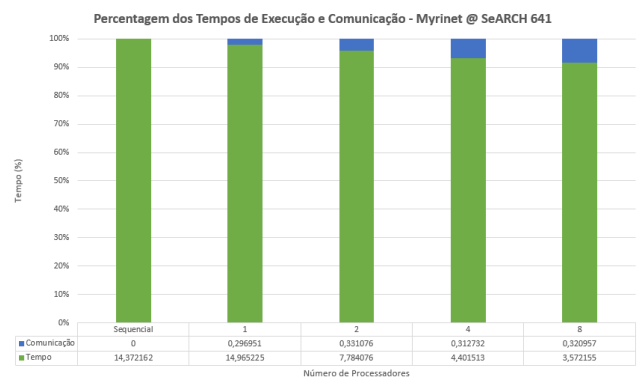


Figura 13. Percentagem dos tempos de execução e comunicação para Myrinet

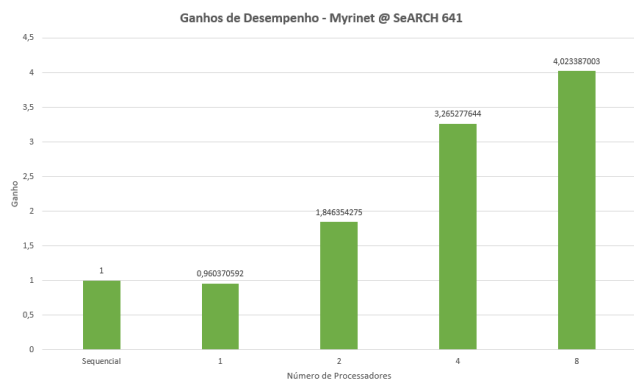


Figura 14. Ganhos de desempenho para Myrinet

3.3 Ethernet VS Myrinet

4. CONCLUSÕES

5. REFERÊNCIAS

- [1] <http://users.cms.caltech.edu/~cs284/lectures/29oct97/sld003.htm>
- [2] <http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/>
- [3] https://github.com/wesleykendall/mpitutorial/blob/gh-pages/tutorials/mpi-scatter-gather-and-allgather/code/all_avg.c
- [4] <https://www.open-mpi.org/faq/?category=myrinet>