PSRS - Parallel Sort by Regular Sampling

Humberto Vaz e João Dias

***Abstract*—O presente documento apresenta o desenvolvimento do terceiro projeto de *Algoritmos Paralelos* referente ao Mestrado Integrado de Engenharia Informática *(MIEI)* correspondente ao ano letivo *2017/2018*, analisando o problema e as estratégias utilizadas para a sua resolução.**

1. Introdução

A ordenação é um dos problemas mais estudados na ciência da computação devido ao seu interesse teórico e importância prática. Com o advento do processamento paralelo, a ordenação num paradigma paralelo tornou-se uma área importante para a pesquisa de algoritmos.

Embora tenha sido feito um trabalho considerável sobre a teoria da ordenação paralela e implementações eficientes em arquiteturas `SIMD`, representado um bom desempenho paralelo numa variedade de arquiteturas `MIMD` (Multiple Instruction Multiple Data) processadas com um grande número de processadores continua a ser um problema desafiador. Uma dessas implementações é o PSRS, ou seja *Parallel Sort by Regular Sampling*.

2. Considerações Iniciais

2.1. Caracterização do Hardware

Para este projeto recorremos à utilização de nodos **662** do Cluster SeARCH.Tal escolha é motivada pela possibilidade para estes nodos fazer ainda o estudo da utilização de Mylinet (**mx**) em comparação com Ethernet (**eth**). Esta máquina é devidamente caracterizadas na tabela seguinte:

	662
Fabricante	Intel
Microarquitectura CPU	Ivy Bridge
# Cores	24 fis; 48 log
Freq. Relogio	2.4 Ghz
L1	d:32K; i:32K
L2	256K
L3	30720K
RAM	64G

De forma a tirarmos partido da localidade espacial e acedermos de forma mais rápida a dados necessitamos de que estes se encontrem próximos do CPU. Sendo assim e como é impossível manter esta quantidade de dados tão grande em registo, estes deverão estar preenchidos de forma a preencher o segundo nível de memória mais rápida da máquina, os vários níveis de Cache.

- Dataset para a cache L1;
- Dataset para a cache L2;
- Dataset para a cache L3;
- Dataset para a RAM;

Tendo estes requisitos em mente chegámos a uma fórmula para calcular a dimensão do problema de interesse:

$$n \leq \frac{TamanhodaCache}{Array * sizeof(Int)}$$

Uma das precauções que tivemos foi ainda a seleção de um **N** que ocupasse exatamente numa linha de cache, prevenindo assim custos desnecessários no carregamento de ”lixo”. Para tal, tendo em conta que nas máquinas utilizadas, uma linha de cache tem `64B` (16 `Int`) chegamos outra formula restringente:

$$LinhadeCache = \frac{N \times N}{16} \quad (1)$$

Sendo assim, esses tamanhos encontram-se calculados na seguinte tabela:

Memória	N
L1	8192
L2	65536
L3	7864320
RAM	10240000

3. Implementações

3.1. Parallel Sort by Regular Sampling

Trata-se de um algoritmo dividido em 4 fases.
Dado um problema de ordenação com `N` elementos e `p` número de processos, fazemos uma divisão inicial desse conjunto de números por vários processos, sendo que cada um irá ter a sua quota parte do problema:

$$N' = \frac{N}{p} \quad (2)$$

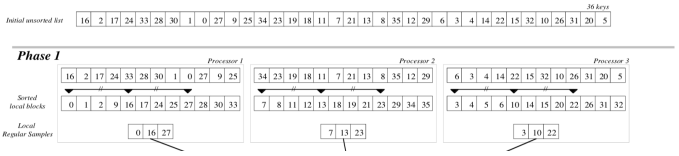


Fig. 1. Fase 1 do PSRS

Na primeira fase cada processo ordena a sua sub-lista de `N'` elementos usando o algoritmo sequencial do *quicksort*.

Na segunda fase, tendo as várias sub-listas/*regularsamples* ordenadas, passamos á escolha de *p-1* pivôs .

Para tal, um processo recebe, junta e ordena as *regular samples* e escolhe *p-1* pivôs, sendo *p* o número de processos escolhido inicialmente e os índices dos pivôs seriam a

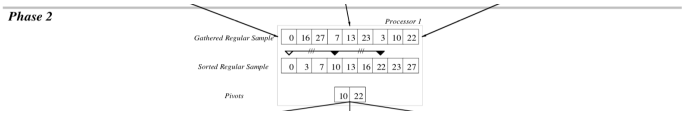


Fig. 2. Fase 2 do PSRS

sequência *p + p, 2p + p, ... (p-1) + p*
De seguida, cada um recebe uma cópia dos pivôs e das *p* partições, sendo que cada partição é um bloco continuo e disjunto de outras partições.

Fig. 3. Fase 3 do PSRS

Na terceira fase, cada processo cria *p* partições consoante os pivôs comunicados pelo processo responsável. Cada um vai conter *p* sub-listas ordenadas. Sendo assim, cada um deles vai ficar com a partição relativa ao seu índice e enviar as restantes *p - 1* sub-listas ordenadas aos restantes processos.

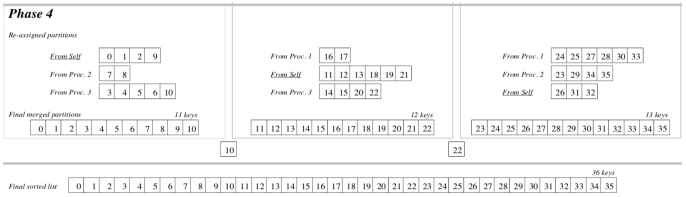


Fig. 4. Fase 4 do PSRS

Na quarta e última fase, cada processo faz paralelamente uma junção das várias sub-listas/partições recebidas com a sublista/partição que não enviou. Sendo que resulta numa sublista ordenada. Isto é, o *i-ésimo* processo irá conter uma lista ordenada com todos os seus elementos menores do que o processo *i+1* e maiores do que o processo *i-1*.

No final existe uma concatenação das listas ordenadas no processo *Master*, sendo esta a lista final ordenada.

4. Problemas da implementação

4.1. Balanceamento de carga

O desempenho da ordenação baseadas em partições depende principalmente de quão bem os dados podem ser particionados de maneira uniforme em subconjuntos menores ordenados. Infelizmente, nenhum método geral e efetivo está disponível atualmente, e é uma questão aberta de como obter um aumento de velocidade linear para classificação paralela em multiprocessadores com um grande número de processadores.

Um ponto relevante de estudo no **PSRS** será o balanceamento de carga.

Na primeira fase, existe uma distribuição equitativa dos elementos a ordenar por todos os processos. A segunda fase é maioritariamente sequencial. A terceira fase está dependente das capacidades de comunicação da máquina.

A fase sensível deste algoritmo é a quarta, por depender dos pivôs globais como se pode ver pela figura 5. Nesta fase, como existe uma receção de mensagens vindas da fase anterior e existir um envio das sub-listas ordenadas para o Master, existe uma sincronização final que pressupõe que todos os processos tenham enviado a sua quota-parte/sub-lista para fazer a junção final no processo Master.

No entanto se existirem sub-listas muito maiores do que outras, isto é, se as mesmas estiverem mal balanceadas, irão existir processos que irão demorar muito, o que no tempo global irá representar um colosso comparativamente à situação das sub-listas estarem balanceadas.

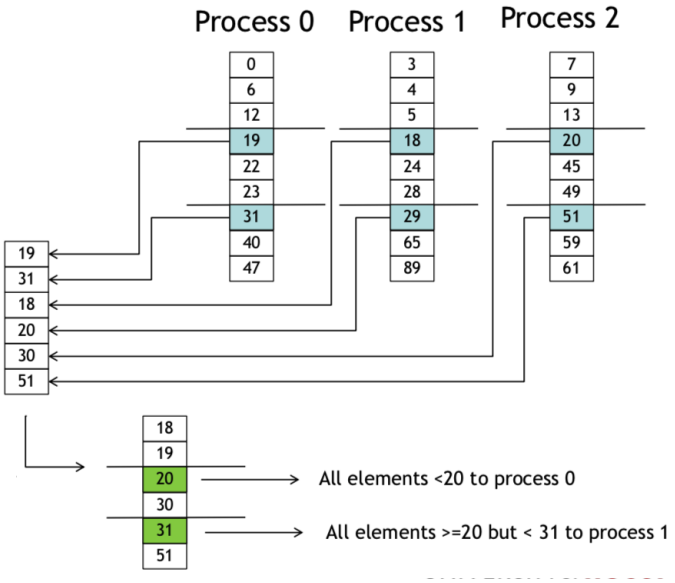


Fig. 5. ”Balanceamento - Balanceado”

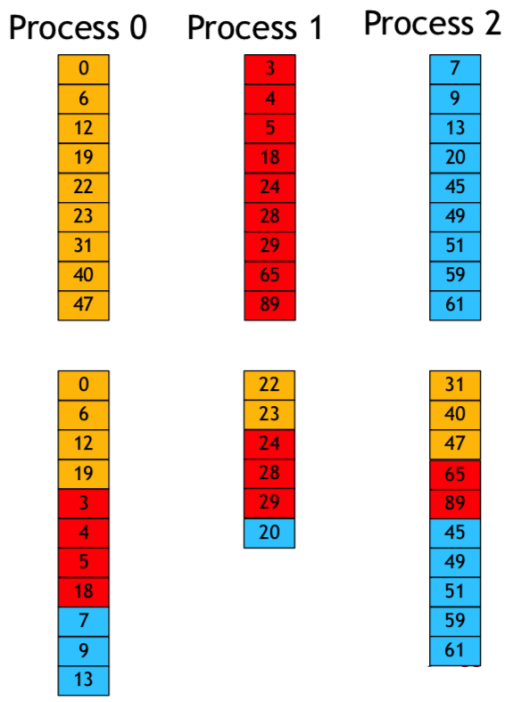


Fig. 6. "Balanceamento - Desbalanceado"

5. Análise da Complexidade

Nas fases um e dois do PSRS, todos os processadores têm aproximadamente a mesma quantidade de trabalho a ser feito.

Na fase três, não é óbvio o quão uniformemente o trabalho é dividido porque isso depende de quão bem os dados foram particionados. No entanto, pode ser mostrado que existe um limite superior na quantidade de trabalho que um processador deve fazer.

Seja y_i o i-ésimo pivô, sendo que $1 \leq i \leq p-1$.

Existem três casos: $i = 1$ - Todos os dados a serem processados pelo processador 1 devem ser menores ou iguais ao pivô 1. Como existem $p^2 - p - \frac{p}{2}$ elementos do mesmo regular sample/sublista que são maiores do que o y_1, existem no mínimo $n - (p^2 - p - \frac{p}{2})$ elementos que são maiores ou iguais ao y_1.

$i = p$ - Todos os elementos a serem processados pelo processador i, terão de ser maiores do que o y_{i-1} . Existem $(p^2 - 2p + \frac{p}{2})$ elementos do regular *sample* que são menores ou iguais do que o pivô $p - 1$, logo, existem no mínimo $n - (p^2 - 2p + \frac{p}{2})$ maiores do que o y_{p-1}.

$1 < i < p$ - Todos os dados a serem processados pelo processador p têm de ser maiores do que o y_{i-1} e menores ou iguais ao pivô y_{i+1}. Existem (i-2)p + $\frac{p}{2}$ elementos do *regular sample* que são menores ou iguais do que o pivô $i - 1$ o que implica que existam $(p-i)\ p - \frac{p}{2}$ elementos do *regular sample* que são maiores do que y_i.

Quanto à complexidade de cada fase, assumindo que $\mathbf{w = \frac{n}{p}}$, para a fase 1, temos que o *quicksort* inicial tem o custo de $O(w\ log\ n)$ quando $n \geq p^3$, sendo que existem w ou $\frac{n}{p}$ elementos para serem ordenados em cada processador.

Na fase 2, dado que são escolhidos p regular *samples* locais em cada processador, e existe um processador a receber esses $p-1$ regular *samples* dos restantes processadores, a complexidade é $O(p^2 \times log\ p^2 + p \times log\ w)$.

Quanto ás fases 3 e 4, estas apresentam uma complexidade de $O(2w(p-1))$ dado que existe a troca de $p-1$ partições de tamanho w e o envio para o Master também de $p-1$ partições de tamanho w para a formação da lista final ordenada.

De seguida apresentamos uma tabela com a complexidade do algoritmo sequenciam bem como soma da complexidade das várias fases para o algoritmo PSRS em paralelo.

Implementação	**Complexidade**
Sequencial	$\Omega(n \times logn)$
Paralela	$O(n/p \times logn), n \geq p^3$

Dado que em arquiteturas MIMD a informação é enviada através de mensagens, durante as várias fases existe troca de mensagens entre processos. Apesar da troca mensagens pressupor o envio por uma entidade e a receção por outra, a seguinte análise tem por base apenas contempla o envio de mensagens, sendo que a cardinalidade das mensagens enviadas é igual á cardinalidade das mensagens recebidas.

- Fase 1 - p mensagens de tamanho $p-1$ causado pelo envio das regular *samples*
- Fase 2 - $p-1$ mensagens de tamanho $p-1$ causado pela comunicação dos pivôs globais aos $p-1$ processadores
- Fase 3 - $p-1$ processos a enviar 1 mensagem de tamanho $O(\frac{n}{p})$ causado pelo envio das partições aos $p-1$ processos
- Fase 4 - $p-1$ processadores a enviar 1 mensagem de tamanho $O(\frac{n}{p})$ para o processador *Master*

Nas quatro fases, cada tarefa acede apenas a uma pequena parte dos dados (sempre menor que $2\frac{n}{p}$) e esses acessos são altamente localizados. Em última instância reduzirá também a latência média de memória. Por estas razões isto em mente o algoritmo é bastante adequado para ambientes de memória distribuída.

Na **segunda fase**, apenas p (p - 1) elementos/**regular samples** precisam ser "recebidos" da **primeira fase**, sendo que $p - 1$ elementos devem devem ser enviados para todos os restantes processos $(p-1)$.

Nas **duas últimas fases**, cada processo tem que enviar sub-listas p - 1 para os outros processadores p - 1.

O número total de mensagens necessárias nesta fase é, portanto, $(p - 1)(p - 1)$ uma vez que o processador Master não precisa de enviar para si mesmo a sua quota parte ordenada caso contrário seriam $p\ (p - 1)$ mensagens. E o tráfego total de dados por sua vez não é maior que n.

Dois pontos finais precisam ser abordados. Primeiro, o algoritmo e sua análise baseiam-se em não haver elementos duplicados na lista a serem ordenados. Caso existam esta análise não será válida. No entanto, mesmo se houver um grande número de chaves, cada uma com um pequeno número de duplicadas, na prática não haverá problema. Se houver chaves para as quais há um grande número de duplicadas, o desempenho paralelo será degradado. Uma possível solução seria a alteração do algoritmo para cobrir esse caso (por exemplo, adicionando uma chave secundária).

Em segundo lugar, é possível usar mais do que elementos *p (p - 1)* na *regular sample* para escolher os pivôs. Porém uma amostra maior implica mais sobrecarga na determinação dos pivôs, embora isso também possa ser feito em paralelo. Se n for grande o suficiente, o balanceamento de carga mais efetivo pode compensar esse custo.

6. Analise empírica

Denotando os resultados teóricos abordados na secção anterior são encorajadores. Nesta secção analisamos de forma experimental as secções abordadas anteriormente. É de referir que quando nos referimos a processos, estamos a fazer uso de vários processadores em máquinas/nós `662` diferentes.

6.1. Resultados experimentais

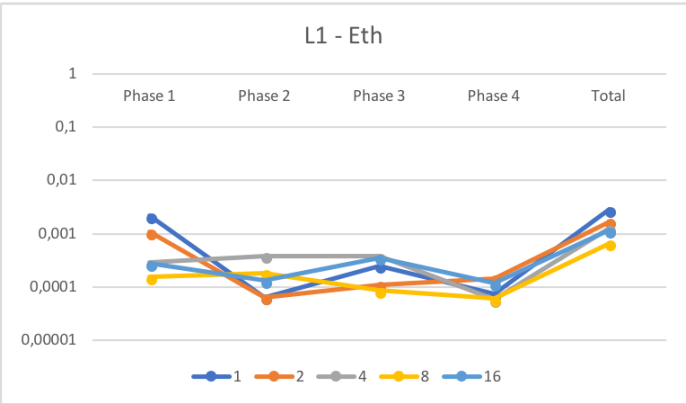


Fig. 7. Tempos L1

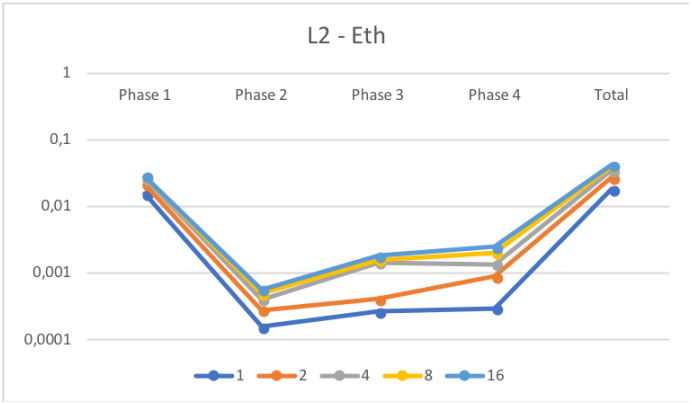


Fig. 8. Tempos L2

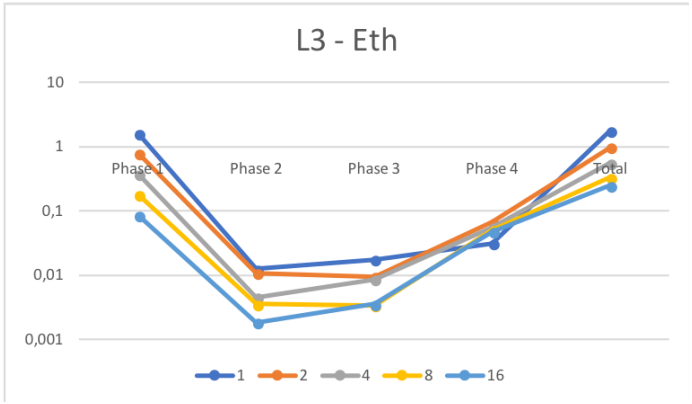


Fig. 9. Tempos L3

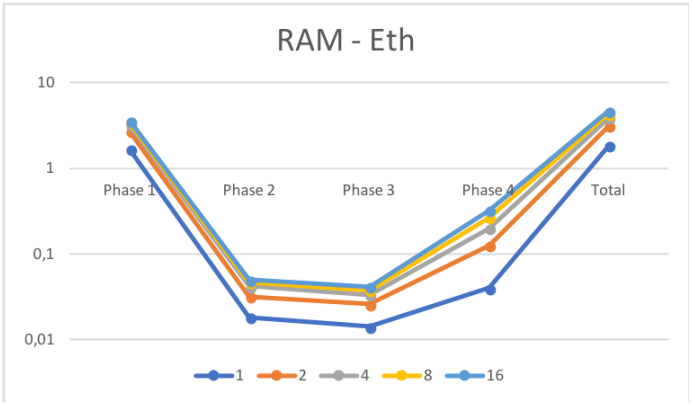


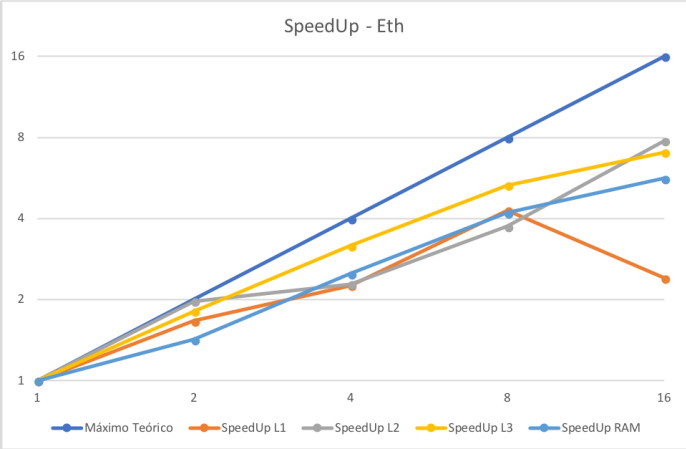
Fig. 10. Tempos ram

Speedup reflete o quão mais rapida uma tarefa pode ser executada comparativamente a apenas um processador/processo.

$$S_p = \frac{T_1}{T_p} \tag{3}$$

S - Speedup; T - Tempo; p - Número de processos;

O algoritmo PSRS garante que se comporta bem em qualquer tipo de distribuição de carga desde que o número de dados duplicados seja relativamente pequeno. Como



VI-A

Fig. 11. speedups associados

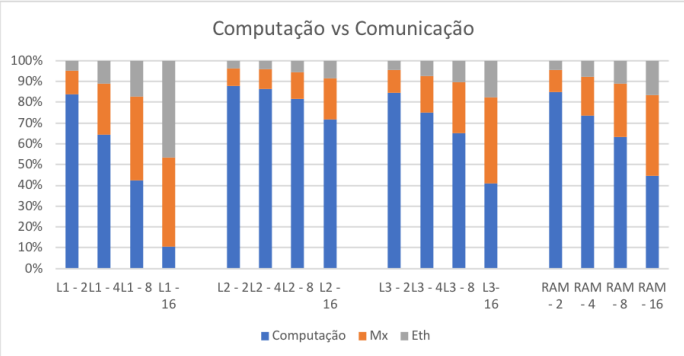


Fig. 12. Custos de comunicação/computação - percentagem acumulada

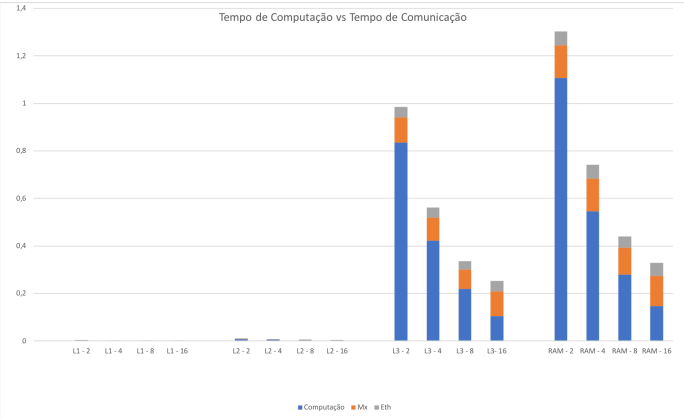


Fig. 13. Tempos Totais - valores acumulados*

podemos ver na figura VI-A, o dataset para qual este obtém maior *speedup* trata-se do dataset gerado para L3, sendo que corresponde `7 864 320` números inteiros ou `31 457 280` bytes a serem ordenados. Pelo que denota ser o número ideal do compromisso **tempo de comunicação - tempo de computação**. Este dataset é o que obtém melhor *speedups* com o aumento do número de processos à exceção de quando fazemos uso de 1, 2 e 16 processos, sendo nestes casos o Dataset que faz frente a trata-se do dataset gerado para L2 .

Como é possível ver, nas figuras 7, 8, 9 e 10, a fase que demora mais é a primeira porque existe um elevado overhead no que diz respeito à ordenação nos *p* processos e no envio de *p-1* mensagens com *p* elementos, sendo estas mensagens relativas à comunicação dos regular *samples* da primeira para a segunda fase.

A seguinte fase que leva mais tempo trata-se da **quarta fase** provavelmente devido à receção de mensagens vindas da **terceira fase** bem como a junção ordenada dos elementos previamente recebidos numa sublista local. No final existem *p - 1* processos a enviar as suas sub-listas ao *Master*. Em suma existe alguma comunicação e algum overhead no que diz respeito à ordenação nos *p* processos.

As seguintes imagens são representativas da comunicação e computação, sendo que os tempos e percentagens relativos à **Ethernet** (eth) devem ser acumulados com os valores referentes à da **Myrinet** (mx), que apresenta sempre valores inferiores à primeira referida.

Note: A implementação destas medições foram feitas de modo *naive* eliminando apenas a computação do programa e correndo-o. Sendo o seu resultado posteriormente subtraído tempo ao total.

Como tinha sido previamente referido anteriormente a escolha da máquina **662** deveu-se em parte à capacidade de utilizar tanto **eth** (ethernet) como **mx** (myrinet).

O problema de ordenação distribuída é para colocar alguns dos elementos de modo que cada sub-lista em cada estação seja ordenado e cada elemento na estação i seja menor ou igual a qualquer elemento na estação i + 1.

Com isto em mente verificamos que existe um grande custo de comunicação, a sua complexidade é geralmente medida em termos do número necessário de mensagens e do tráfego total de dados, sendo que, vai haver um acréscimo do numero de mensagens com o acréscimo de processos em atividade. Verificamos no caso de **16** processos o custo de comunicação é superior ao de computação.

Outra questão que verificamos foi que apesar da utilização de **Myrinet** apresenta ganhos razoáveis apesar de não estarem na mesma ordem de grandeza da superioridade apresentada em relação à **Ethernet**. Sendo que a **Ethernet** tem uma velocidade de **10 Mb/s** e a **Myrinet 1 Gb/s** esta diferença de comparativamente com a **ethernet** que apresenta velocidades

de 10^3 esta diferença não é traduzida nos tempos, o que nos leva a concluir que o custo superior da comunicação seria resultante da processo de alocação e definição de destino das mensagens e não a taxa de envio.

7. Conclusão e Trabalho futuro

No final, observando todas as conclusões da seção anterior, entendemos que esse algoritmo é bom em diferentes arquiteturas e reconhece sua força no equilíbrio de carga de trabalho.

Em relação ao trabalho futuro, acreditamos que seria relevante um estudo mais aprofundado relativamente à existência de um maior numero de duplicados e a forma que o PSRS se comporta a lidar com esses casos e a sua possível degradação de performance.

Como conseguimos ver pela figura VI-A, consegue-se perceber que existe um elevado peso percentual na comunicação em relação à computação para Datasets muito pequenos com 8 e 16 processos, o que é expectável pois trata-se de pouco trabalho computacional para muitos processos. Á medida que avançamos no que diz respeito ao tamanho do Dataset, esta diferença percentual vai se ”diluindo” nos tempos de comunicação efetiva.

Ao analisarmos a figura VI-A em conjunto com VI-A dá para concluir que o ponto ideal entre comunicação e computação trata-se do Dataset correspondente à RAM (`40 960 000` bytes) e `16` processos, o que torna este algoritmo bastante atrativo no que diz respeito ao *High Performance Computing*.

Em conclusão, conseguimos compreender os prós e contras do algoritmo com detalhes suficientes para fazer o julgamento final das melhores situações para utilizá-lo em detrimento de outros.

8. Bibliografia

- LI,Xiao et al. *On the Versatility of Parallel Sorting by Regular Sampling*
- SHI,Hanmao; SCHAEFFER, Jonathan. *Parallel Sorting by Regular Sampling*