Programação Paralela e o Modelo Mestre-Escravo

Trabalho 1 - Programação Paralela e Distribuída

Nicolas Pereira do Nascimento

Estudante de Engenharia de Computação

Pontifícia Universidade Católica do Rio Grande do Sul

Porto Alegre, Brasil

nicolas.nascimento@acad.pucrs.br

Gabriel Chieza Chiele

Estudante de Engenharia de Computação

Pontifícia Universidade Católica do Rio Grande do Sul

Porto Alegre, Brasil

gabriel.chiele@acad.pucrs.br

*Abstract*— This paper presents the implementation of a parallel algorithm using the master-slave paradigm and the MPI libraries. It also compares the performance of algorithm when modifying the number of processes used to execute.

Keywords—Parallel Programming; MPI; Distributed Programming, Master-Slave Paradigm;

# Introdução

A construção de sistemas atuais se baseia em diferentes frentes. Dentre elas estão aspectos econômicos, legais e performáticos. Esta última, especificamente, é bastante relevante e é alvo de diversos estudos. Uma das alternativas que visa o aumento de desempenho é a aplicação de um modelo de programação paralela, que objetiva a realização de tarefas através de múltiplos processos executando em múltiplos processadores. Cabe ressaltar que o paradigma paralelo pode ser utilizado em sistemas com um único processador, contudo a execução das tarefas torna-se concorrente e o paralelismo é apenas aparente.

# Funcionamento

## Algoritmo

A fim de garantir a fidelidade dos resultados e visando medir o ganho de performance da mudança de paradigma exclusivamente, o algoritmo utilizado foi o Rank Sort. Este algoritmo é um algoritmo de ordenação de vetor e caracteriza-se por ter uma performance de ordem quadrática (O(n²)).

## Paradigma Paralelo

A programação paralela pode ser realizada de muitas formas. Especificamente a esse trabalho, usou-se o paradigma mestre-escravo. O modelo especifica uma hierarquia entre os processos de maneira que exista um processo “mestre” e diversos processos “escravos”. O processo “mestre” fica responsável pela divisão da tarefa total em sub-tarefas, pela distribuição das mesmas entre os processos “escravos” e pela integração dos resultados provenientes dos escravos. Aos processos escravos cabe realizar as sub-tarefas e, ao finalizarem-las, enviar as sub-tarefas completas ao mestre.

# Implementação

O trabalho foi desenvolvido em ambiente Linux e implementado em linguagem C, utilizando a biblioteca MPI para a distribuição dos processos e a criação das comunicações necessárias.

O programa implementado realiza a ordenação, em ordem crescente, de um vetor, que é populado com valores contidos em um arquivo. O caminho do arquivo deve ser passado por parâmetro, juntamente com o tamanho desejado para o vetor, além do parâmetro “np” (representando o número de processos a ser utilizados) utilizado pela biblioteca MPI.

Caso o valor “np” seja igual a um, o programa irá executar sua versão sequencial, caso contrário, será criado um processo mestre, que nesta implementação é o processo cujo rank é igual a zero, os demais processos criados serão os escravos.

O processo mestre é encarregado de separar o vetor em diversas partes e entrega-las aos escravos, através da chamada MPI\_Send(), que serão encarregados de realizar a ordenação da parte que lhes foi entregue.

O número de partes em que o vetor será dividido é definido pela equação:

n° de partes = tamanho do vetor / ( n° de processos \* 4 )

# Execução

Todos os testes da aplicação foram executados no LAD(Laboratório do Alto Desempenho) da PUCRS. O cluster utilizado, chamado atlântica, tem sua especificação detalhada abaixo:

“O cluster Atlântica é composto por 16 máquinas Dell PowerEdge R610. Cada máquina possui dois processadores Intel Xeon Quad-Core E5520 2.27 GHz Hyper-Threading e 16GB de memória, totalizando 8 núcleos (16 threads) por nó e 128 núcleos (256 threads) no cluster. Os nós estão interligados por 4 redes Gigabit-Ethernet chaveadas.

O último nó (atlantica16), dispõe de uma NVIDIA Tesla S2050 Computing System, com 4 NVIDIA Fermi computing processors (448 CUDA cores cada) divididos em 2 host interfaces e 12GB de memória (3GB por GPU) ”[1].

# Testes

Os testes de aplicação foram realizados utilizando-se um arquivo contendo 100 mil números inteiros desordenados. Os parâmetros alterados a cada teste eram o número de processo e número de elementos no vetor. Ao final de cada teste, um arquivo chamado "output.txt" era gerado e continha o vetor ordenado.

# Resultados

Os resultados encontrados se baseiam em medir o tempo de execução da aplicação com diferentes cargas e números de processos, foram executados testes com 10k, 50k e 100k elementos, para 1, 4, 8 e 16 processos, totalizando 12 amostras.

Com estas medidas obtemos os valores de speed up e eficiência para a implementação. Estas medidas informam a melhoria em tempo e em utilização do processador resultante do paralelismo.

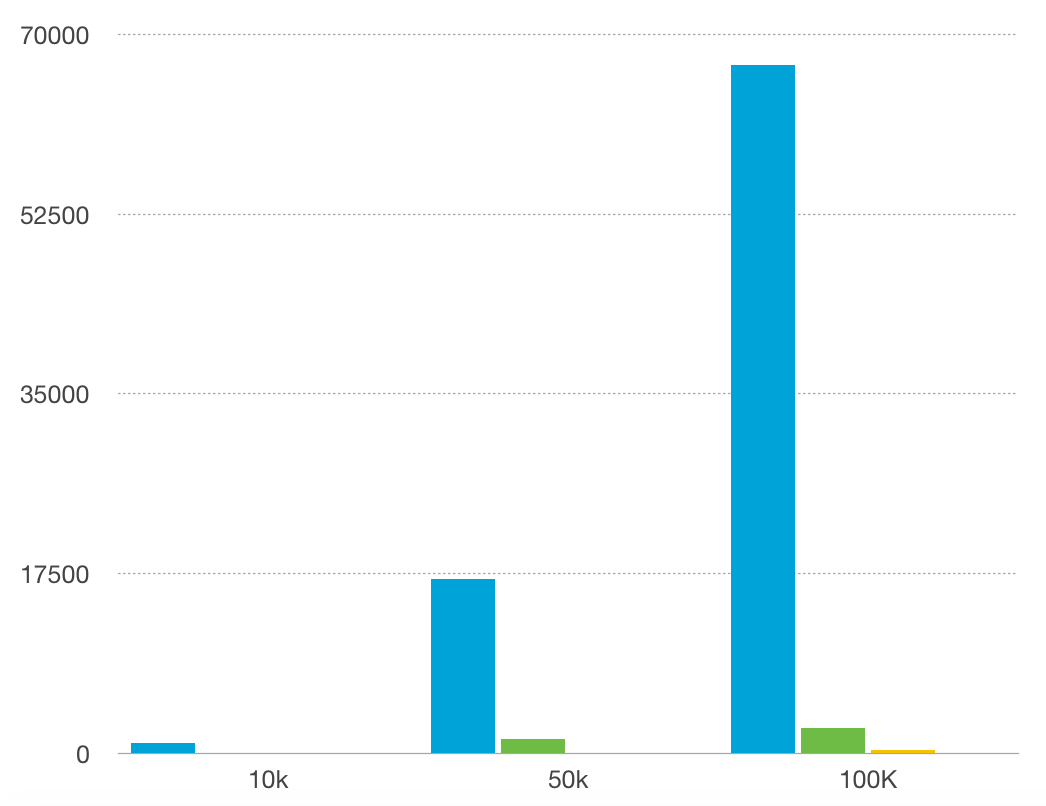
Estas medidas, apresentadas abaixo, são calculados da seguinte forma:

* Speed up = tempo de execução sequencial / tempo de execução para n processos
* Eficiencia = speed up / n° de processos

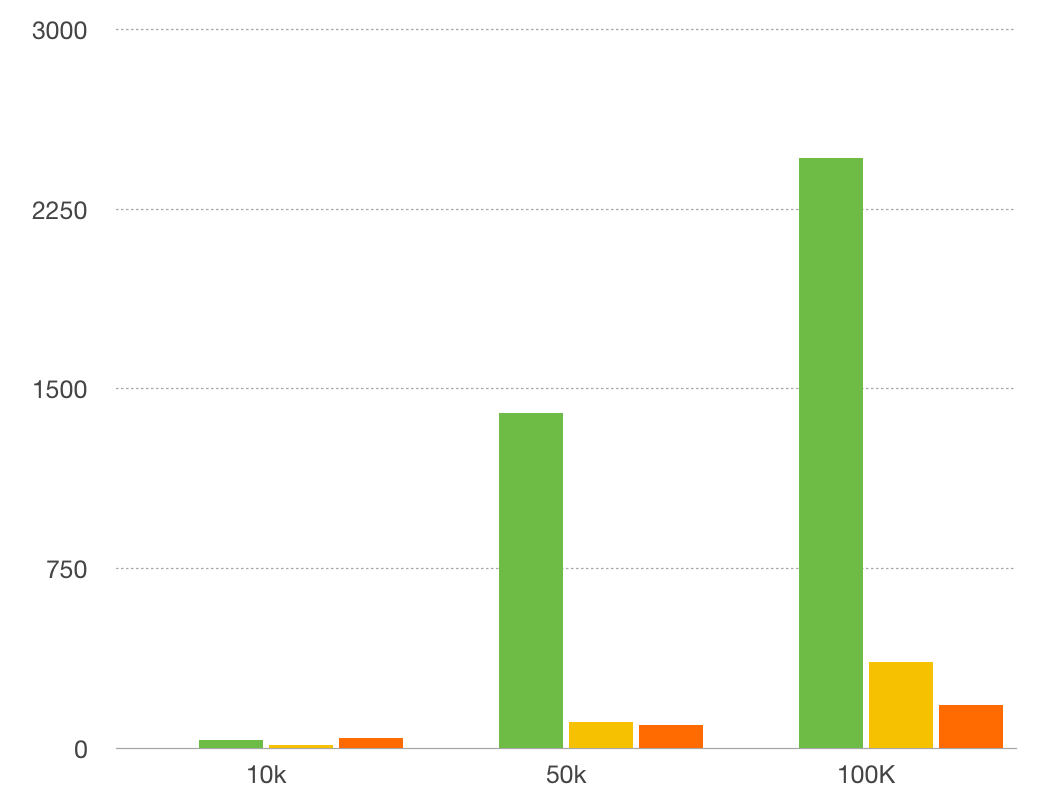
Tabela I

| Tempo de Processamento (milisegundos) | | | |
| --- | --- | --- | --- |
| N° de processos | N° de elementos do vetor | | |
| 10K | 50K | 100K |
| 1 | 1000 | 17000 | 67000 |
| 4 | 34 | 1400 | 2463 |
| 8 | 14 | 109 | 360 |
| 16 | 43 | 100 | 183 |

1. Tabela comparativa entre os tempos de execução em milissegundos.

Figura I.a

1. Comparação entre os tempos de execução. *(azul - np 1; verde - np 4; amarelo - np 8; laranja- np 16).*

Figura I.b

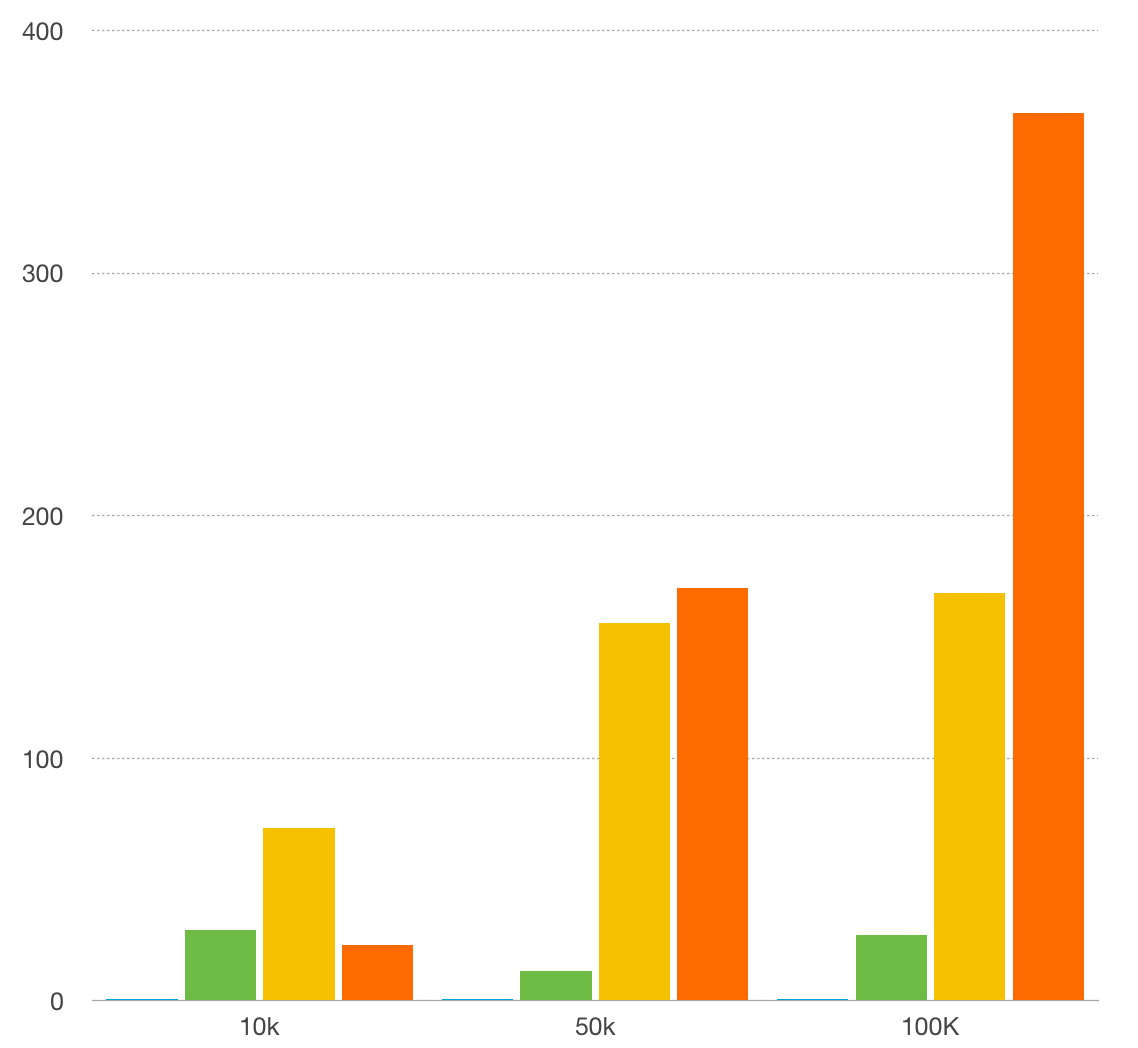
1. Comparação entre os tempos de execução excluindo-se a versão sequencial. *(verde - np 4; amarelo - np 8; laranja- np 16).*

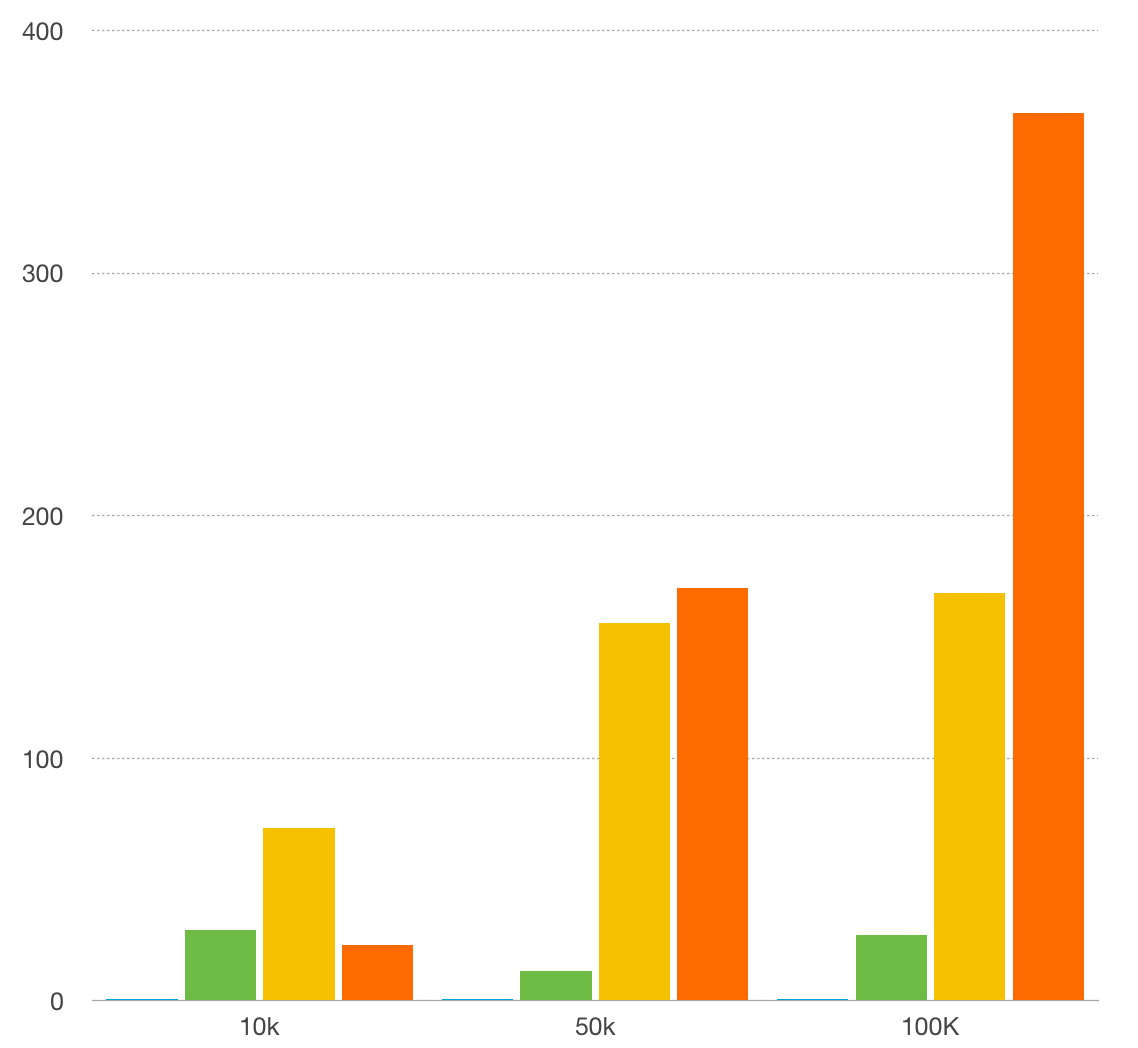
Tabela II

| Speed Up | | | |
| --- | --- | --- | --- |
| N° de processos | N° de elementos do vetor | | |
| 10K | 50K | 100K |
| 1 | 1 | 1 | 1 |
| 4 | 29.4 | 12.14 | 27.20 |
| 8 | 71.4 | 155.96 | 168.11 |
| 16 | 23.2 | 170 | 366.12 |

1. Tabela dos valores calculados de speed up.

Figura II





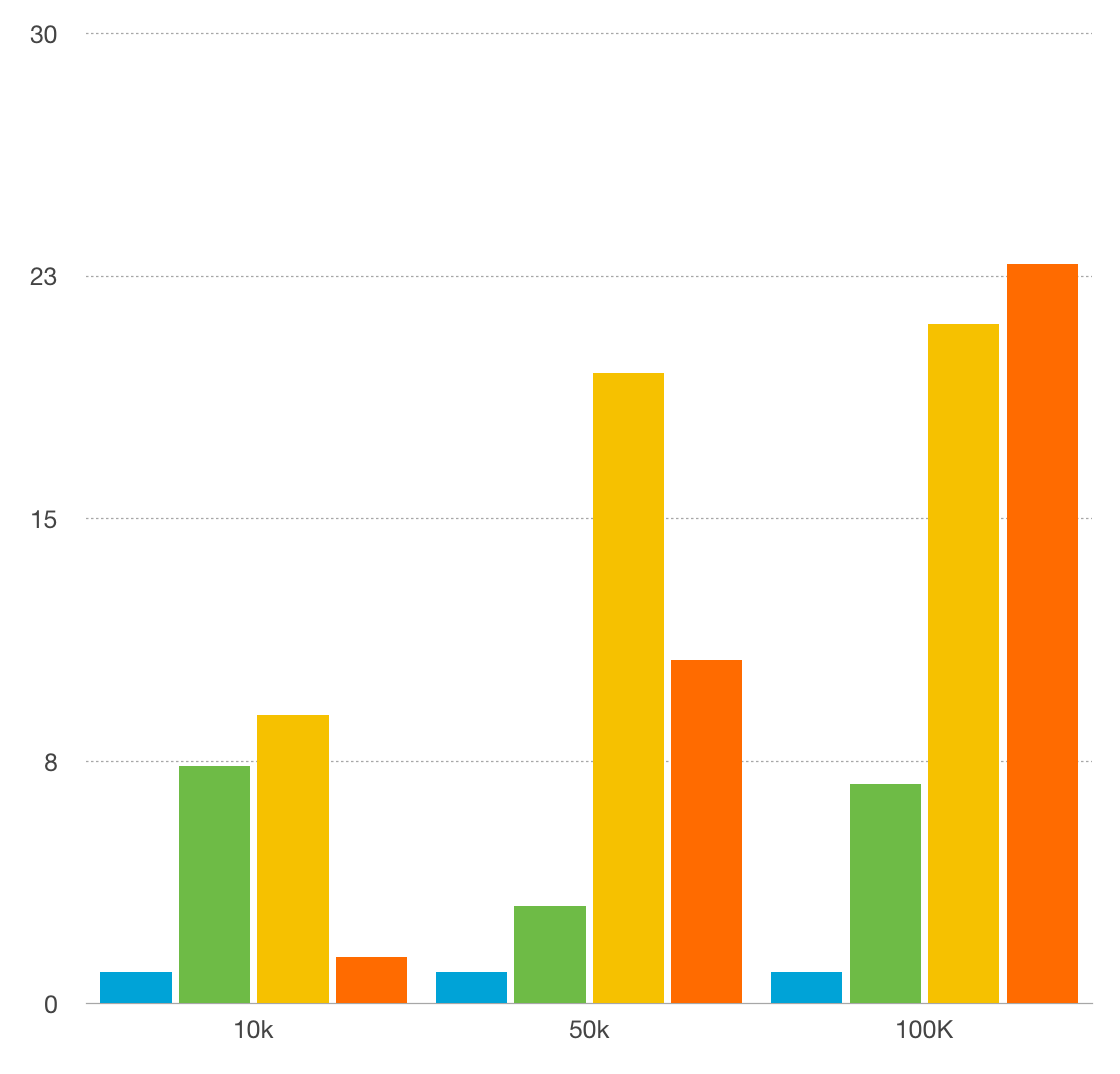
1. Gráfico dos valores calculados de speed up. *(azul - np 1; verde - np 4; amarelo - np 8; laranja- np 16)*

Tabela III

| Eficiência | | | |
| --- | --- | --- | --- |
| N° de processos | N° de elementos do vetor | | |
| 10K | 50K | 100K |
| 1 | 1 | 1 | 1 |
| 4 | 7.35 | 3.035 | 6.8 |
| 8 | 8.92 | 19.49 | 21.013 |
| 16 | 1.45 | 10.62 | 22.88 |

1. Tabela dos valores calculados de Eficiência.

Figura 3



1. Gráfico dos valores calculados de Eficiência. *(azul - np 1; verde - np 4; amarelo - np 8; laranja- np 16)*

# Conclusões

Após uma breve análise dos resultados, alguns pontos relevantes aparecem. Os principais são:

* Em geral, o tempo de execução é bastante reduzido ao aumentar-se o número de processos. Contudo, aumentar excessivamente esta quantidade pode fazer com que a aplicação gaste mais tempo realizando a comunicação entre os processos do que propriamente realizando a ordenação. Isso fica evidente ao utilizar “np" valendo 16 com um vetor de 10 mil elementos, onde o tempo de execução, quando comparado com np 8 e mesmo número de elementos, sofre um acréscimo.
* Os valores de speed up, que idealmente são iguais a “np”, acabam extrapolando o valor ideal. Isso ocorre principalmente pela natureza quadrática da versão sequencial do algoritmo. Essa característica faz com que o tempo de computação do mesmo algoritmo, por parte dos escravos, sofra uma melhoria excepcional ao utilizar um parte menor do vetor sendo ordenada por cada escravo.
* Devido ao excelente speed up encontrado, a eficiência encontrada vai muito além de 1(valor ideal). Porém, isso não significa que a utilização dos processadores seja além do esperado, mas sim que o problema tinha uma natureza, em termos de performance, bem ruim.

##### References

1. LAD ( Laboratório de Alto Desempenho) – Descrição Cluster Atlântica http://www3.pucrs.br/portal/page/portal/ideia/Capa/LAD/LADInfraestrutura/LADInfraestruturaHardware