Avaliando o desempenho de aglomerados com biblioteca MPI utilizando benchmark MPBench

Nicolas Kassalias, Edson T. Midorikawa

Departamento de Engenharia de Computação e Sistemas Digitais, Universidade de São Paulo

Av. Professor Luciano Gualberto, travessa 3, 158, São Paulo, SP

{edson@pcs.usp.br}

Resumo

Este trabalho compara o desempenho de aglomerados com 8 e 16 nós com o pacote de benchmarks MPBench utiliza bibliotecas de passagem de mensagem MPI.

O trabalho apresenta uma visão a respeito de arquiteturas paralelas, aglomerados, programação com troca de mensagem de mensagens e ao final expõe-se os testes de avaliação e seus resultados.

Os resultados dos testes com 2 processos apresentaram comportamentos similares em ambos os aglomerados. Os testes com mais de dois processos mostram uma relação entre a quantia de processos com a queda de desempenho.

Abstract

This work compares the performance between two cluters with 8 and 16 nodes with MPBench benchmarks.

This paper gives an overview of architecture parallels, clustes, MPI and PVM.

The tests with 2 process are similar between both clusters, but when there are more process the performance get down.

Keywords: programação paralela, benchmarks, MPI, PVM, cluster.

1. INTRODUÇÃO

A crescente demanda de processamento e manipulação de grandes volumes de dados, tem estimulado novas pesquisas para aumentar o desempenho dos sistemas computacionais. Para isso adotam-se várias medidas alternativas, tais como: incrementar a taxa de velocidade do clock do processador, uso de pileline, alterações no sistema de memória, etc. Isso tem gerado diversos tipos de arquiteturas podendo-se citar as com processadores RISC, vetoriais, superescalares, entre outras.

A programação paralela e distribuída é uma das possíveis alternativas de obtenção de alto desempenho. Esse modo de programar gera a necessidade de novos equipamentos, linguagens, compiladores, bibliotecas, novas métricas de desempenho, etc.

Segundo [TANEN2][TATH], O desenvolvimento de arquiteturas paralelas nasceu com a necessidade do aumento da velocidade de processamento. Apesar de as máquinas RISC prometerem um incremento no desempenho, tais máquinas ainda utilizam a concepção de Von Neumann de um computador, que consiste em ter um barramento de dados ligando uma unidade de processamento (CPU) a uma unidade de memória e a dispositivos de entrada e saída.

A estrutura tradicional de um computador com um processador que envia requisições seqüencias através de um barramento para a memória que responde a uma requisição por vez é conhecida por "gargalo de Von Newmann". Existem pesquisas para eliminar esse "gargalo". Uma dessas tentativas é a construção de máquinas paralelas que apresentam mais de um processador.

Pode-se dividir o projeto de computadores paralelos em três categorias baseando-se no fluxo de dados: computadores com fluxo único de instruções e dados, denominados SISD (*Single Instruction, Single Data*); fluxo único de instruções e múltiplo fluxo de dados, máquinas classificadas como SIMD (*Single Instruction Multiple Data*); computadores com fluxo múltiplo de instruções e de dados MIND (*Multiple Instruction Multiple Data*).

O processamento paralelo gera cuidados como a necessidade de controle de sincronização, mecanismos de comunicação entre tarefas, gerência de recursos compartilhados, análises de

dependência de dados. Os programas podem ser paralelizados automaticamente ou explicitamente.

No caso de paralelização explícita, utilizam-se comandos específicos de programação paralela, diretiva de compilação ou bibliotecas.

A utilização de vários processadores e equipamentos para aumentar o desempenho de um sistema computacional gera problemas de concorrência destes processadores aos recursos do hardware e, novas técnicas de mantê-los o maior tempo possível ocupado tem sido objeto de pesquisa.

O desempenho de uma máquina não cresce proporcional ao número de processadores disponíveis. Devido aos problemas de disputa pelo uso do barramento pelos processadores, pode ocorrer até uma queda de desempenho.

A comunicação entre os processos tem sido implementada em maior escala de duas formas: Por memória compartilhada ou troca de mensagens.

Na comunicação por memória compartilhada, cada processador tem acesso a uma memória global, sendo possível, portanto, um processador ler os dados armazenados por outros. Esta forma é mais simples de programar. Máquinas com este tipo de comunicação devem prover controle de acesso à memória bastante complexa para evitar que dois processadores acessem a mesma área de memória simultaneamente. Um dos problemas deste tipo de controle de acesso para implementar a exclusão mútua é a geração de "gargalo" quando existir uma grande quantidade de processadores.

Na comunicação com troca de mensagens, a ideia é permitir que os processos comuniquem-se sem precisar do compartilhamento de uma memória global. Máquinas conectadas em rede poderiam executar a programação paralela através da utilização de bibliotecas de troca de mensagens. Entre as bibliotecas mais conhecidas pode-se citar a PVM (Parallel Virtual Machine) e MPI (Message-Passing Interface).

Segundo [UFRGS2], a ideia de utilizar as bibliotecas de comunicação permite construir uma máquina "virtual" com vários processadores. Com isso pode-se dividir uma grande tarefa para que cada um dos processadores execute paralelamente partes dessa tarefa, contribuindo assim para o resultado final.

A utilização dessas bibliotecas esta sendo largamente utilizada devido a possibilidade de execução paralela com um investimento em hardware relativamente baixo, uma vez que pode-se realizar tarefas paralelizadas com aglomerados (clusters) que são máquinas independentes ligadas em rede.

A pesquisa em programação paralela tem buscado técnicas eficientes para simular memória compartilhada, conhecidas como DSM (Distributed Shared Memory). A idéia típica consiste em ligar-se vários processadores através de uma rede de alta velocidade e simular memória compartilhada através de troca de mensagens. Tipicamente, cada processador determina uma área da sua memória que será compartilhada com os demais processadores do sistema. O gerenciador de memória, então, ficará encarregado de fazer a devida coerência desta parte da memória.

Após ser verificada as alternativas de aumento de desempenho em sistemas computacionais, passa-se a apresentação da avaliação de desempenho de aglomerados que foram de objeto de estudo deste trabalho.

2.AVALIAÇÃO DE DESEMPENHOS E AGLOMERADOS (CLUSTERS)

Como já mencionado este trabalho realizou seus testes comparativos entre dois aglomerados com 8 e 16 nós.

[UFRGS1] Aglomerado é um conjunto de servidores ou computadores comuns (PCs) ligados em rede e que podem trabalhar de forma coordenada dividindo um grande trabalho em pequenas tarefas distribuídas entre os vários nós.

Existem hoje diversas estruturas de aglomerados, sendo uma das mais famosas a Beowulf devido ao seu baixo custo e flexibilidade.

O projeto Beowulf foi iniciado na CESDIS (Center of Excellence in Space Data & Information Sciences) no verão de 1994 com a construção de um aglomerado com 16 nós, desenvolvido para a Earth and space sciences project (ESS) no Goddard Space Flight Center (GSFC).

Um sistema Beowulf é composto basicamente por:

- Nós: Normalmente constituídos de servidores ou computadores simples.
- Nó central: Um nó que funciona como a interface com o mundo exterior. Normalmente é o equipamento com melhores recursos pois será o responsável por dividir o trabalho entre os outros nós.
- Sistema operacional: Geralmente usa-se o LINUX que é o principal software executado nos nós.
- Infra-estrutura de rede: É por onde os nós trocam suas informações. É constituído pelo cabeamento, HUBs s e SWITCHs . Quanto melhor e mais rápida for a infra-estrutura, mais rápido será o AGOLMERADO.
- Software e bibliotecas de funções especiais: permitem que um programa escrito para um sistema Beowulf tenha o seu processamento dividido entre os nós. Permitem também a utilização das tecnologias Network Virtual Memory / Distributed Shared Memory que dão a ilusão de que a memória total do sistema é a soma das memórias de todos os nós que o compõem.

Para realizar a comparação entre o desempenho dos aglomerados adotaram-se benchmarks que utilizam bibliotecas MPI.

Benchmarks são programas utilizados para medir o desempenho e características de sistemas. Segundo [BALS97], benchmarking significa medir a velocidade com a qual um sistema computacional irá executar uma tarefa de modo a permitir a comparação entre diferentes combinações de *hardware e software*. Pode-se utilizá-lo para auxiliar processos decisórios através da obtenção de dados referentes à relação desempenho / custo, velocidade de execução, desempenho de funções, estabilidade de sistema, etc. Para utilizar o benchmark deve-se identificar com clareza qual seu objetivo, quais os processos estão sendo medidos e envolvidos e quais os recursos serão necessários.

Existem diversos pacotes de benchmarks para medir o desempenho de processamento paralelo com troca de mensagem utilizando bibliotecas MPI ou PVM.

Pode-se mencionar os seguintes:

- SkaMPI MPI Benchmark - Special Karlsruher MPI-Benchmark [LIIN]; Genesis Parallel Benchmarks (PVM) [HPCC]; ScaBench - Scali's [SCAL]; NAS [NAS]; OpenMP C - [PDPL]; MPBench – [ICL]

Neste trabalho foram adotados benchmarks do pacote MPBench que será apresentado adiante.

3. BENCHMARK MPBENCH

O benchmark adotado foi o MPBench que serve para avaliar o desempenho de aglomerados com bibliotecas MPI. Esse pacote é bastante flexível e altamente portável.

O MPBench consiste de oito tipos de testes com diferentes chamadas de biblioteca MPI. Os testes estão apresentados na figura 3.1 abaixo:

Benchmark	***Unidades***	***Número de Processos***
Bandwidth	Kilobytes/seg	2
Roundtrip	Transações/seg	2
Latency	Microsegundos	2
Broadcast	Kilobytes/seg	2, 4, 6, 8 ou 16
Reduce	Kilobytes/seg	2, 4, 6, 8 ou 16
AllReduce	Kilobytes/seg	2, 4, 6, 8 ou 16
Bidirectional Bandwidth	Kilobytes/seg	2
All-to-All	Kilobytes/seg	2, 4, 6, 8 ou 16

Fig. 3.1.- Benchmarks do pacote MPBench

A figura 3.1 apresenta na primeira coluna o tipo de benchmark, na segunda as unidades obtidas pelos testes, já a terceira contém o número de processos criados.

Todos os testes medem o tempo da seguinte maneira:

1. Prepara a configuração do teste.
2. Inicia a medição do tempo.
3. Executa laços de operações para cada tamanho de mensagem trocada em potência de dois bytes e conta o número de iterações.
4. Verifica se as operações finalizaram
5. Finaliza a tomada de tempo.

Por *default*, o MPBench mede mensagens a partir de 2^2 bytes até 2^{16} bytes, em potência de dois para 100 iterações. A memória *cache* é limpa antes de cada laço e depois de cada novo tamanho de mensagem.

Para os benchmarks, existem dois tipos de processos principais, o *mestre* que é único e os *escravos* que podem existir em quantidades variadas. Os testes chamados de *ponto-a-ponto* apresentam apenas dois processos, um *mestre* e um *escravo*. Os demais testes rodam com quantidades variadas de processos *escravos.* A figura 3.1 informa as quantias de processos adotados.

O algoritmo geral do benchmark é o apresentado a seguir.

```
envia o primeiro grupo de mensagens
se a mensagem é maior que N
  espera a resposta do endereço destino Y
senão
  se existe mais para ser enviado
    envia as demais mensagens
fim
```

MPI evita efetuar cópias de dados desnecessárias que normalmente gera uma sobrecarga (overhead). O processo receptor armazena uma quantidade determinada de dados, antes de enviar uma mensagem de retorno. O processo emissor deverá aguardar a respectiva mensagem de recebimento do processo receptor, só após isso ocorrer é que o processo emissor poderá enviar nova mensagem.

A seguir será apresentado cada um dos oito benchmarks referenciados na figura 3.1

3.1 Bandwidth

Este teste mede a largura de banda. Apresenta um laço duplo aninhado. O laço mais externo varia o tamanho da mensagem e o interno realiza a contagem de operações de envio.

Depois de completadas as iterações o processo *escravo* envia um sinal de retorno ao processo emissor, esse sinal é um dado de tamanho de 4 bytes que é retornado ao processo *mestre*. Essa é a forma de informar ao emissor quando os processos *escravos* finalizaram o recebimento dos dados e estão prontos para receber mais mensagens.

Essa troca de informações faz-se necessária porque a emissão pode ser completada antes do recebimento dos dados por parte do processo *escravo*. É lógico que essas mensagens de retorno geram sobrecarga, mas para uma grande quantidade de iterações esse efeito é minimizado.

O pseudocódigo do processo *mestre* é apresentado a seguir:

```
faça variando o tamanho das mensagens
  tempo inicial
  faça variando o contador de iterações
    envie( mensagem)
  recebe(retorno)
  tempo final
```

A seguir o pseudocódigo dos processos *escravos* :

```
faça variando o tamanho das mensagens
    faça variando o contador de iterações
        recebe(mensagem)
    envie(retorno)
```

3.2 Bidirectional Bandwidth

Este benchmark mede a largura de banda bidirecional. Apresenta um duplo aninhamento de laços. Como o anterior, o laço externo varia o tamanho das mensagens e o interno conta as operações de envios de mensagens. O processo mestre e o escravo, diferente dos demais testes executam um recebimento não blocado, seguido de um envio também não blocado e finalmente seguido de uma espera para cada uma das iterações.

O pseudocódigo do processo mestre é apresentado a seguir:

```
faça variando o tamanho das mensagens
    tempo inicial
    faça variando o contador de iterações
        imediato(nonblocking) recebe(mensagem)
        imediato(nonblocking) envie(mensagem)
        espere até mensagens terem sido recebidas
    tempo final
```

O pseudocódigo dos processo escravo é apresentado a seguir:

```
faça variando o tamanho das mensagens
  faça variando o contador de iterações
    imediato(nonblocking) recebe(mensagem)
    imediato(nonblocking) envie(mensagem)
    espere até mensagens terem sido recebidas
```

3.3 Roundtrip

As medidas obtidas no benchmark Roundtrip são obtidas de forma similar ao benchmarks Bandwidth, exceto que os processos escravos, depois de receberem as mensagens, ecoam essas mensagens de volta para o processo mestre. Esse benchmark é também conhecido como *ping-pong*. As unidades de medidas são dadas em transações por segundo.

A seguir o pseudocódigo do processo *mestre*:

```
faça variando o tamanho das mensagens
  tempo inicial
  faça variando o contador de iterações
    envie(mensagem)
    receba(mensagem)
  tempo final
```

O pseudocódigo dos processos *escravos* vem a seguir:

```
faça variando o tamanho das mensagens
  faça variando o contador de iterações
    recebe(mensagem)
    envie(mensagem)
```

3.4 Latency

Este *benchmark* mede o tempo para que uma aplicação envie uma mensagem e continue a sua execução. Os resultados deste teste variam muito em função de como a camada de transmissão de dados é implementada. Este benchmark é semelhante ao Bandwidth exceto que não recebe

mensagem de retorno. Este teste é diferente dos demais, pois ele apenas registrar o tempo para o envio de dados.

A seguir tem-se o pseudocódigo do processo *mestre*:

```
faça variando o tamanho das mensagens
  tempo inicial
  faça variando o contador de iterações
    envie(mensagem)
   tempo final
```

Abaixo o pseudocódigo dos processos *escravos* :

```
faça variando o tamanho das mensagens
  faça variando o contador de iterações
    recebe(mensagem)
```

3.5 Broadcast e Reduce

Estas duas aplicações são muito comuns. As operações entre os benchmarks Broadcast e Reduce parecem como imagens refletidas em um espelho. A diferença básica entre ambos, é que o *Reduce* inverte a direção da comunicação em relação ao Broadcast. Ambos os testes retornam os valores medidos em kilobytes por segundo. O broadcast é um modo de passagem de mensagem que envia a partir de um processo, mensagens para todos. Já o reduce é justamente ao contrário do broadcast, isto é, vários processos enviam suas mensagem para apenas um.

A seguir está o pseudocódigo de ambos os processos o *mestre* e os *escravos*:

```
faça variando o tamanho das mensagens
  tempo inicial
  faça variando o contador de iterações
    reduce ou broadcast(mensagem)
    envie(mensagem)
  tempo final
```

3.6 AllReduce

AllReduce é uma derivação do benchmark *All-to-all.* aonde cada processo envia dados para os demais processos. Esse benchmark poderia ser implementado com a operação de reduce, seguida de um broadcast. O teste mede a largura de banda de comunicação entre múltiplos processos.

O pseudocódigo do processo *mestre* vem a seguir:

```
faça variando o tamanho das mensagens
  tempo inicial
  faça variando o contador de iterações
    allreduce(mensagem)
tempo final
```

A seguir o pseudocódigo do processo *escravo* :

```
faça variando o tamanho das mensagens
  faça variando o contador de iterações
    allreduce(mensagem)
```

3.7 All-to-All

All-to-all mede a velocidade de comunicação ao estilo *round-robin* entre múltiplos processos. O laço externo varia o tamanho da mensagem e o interno conta as operações de envio de mensagem. Cada processo envia uma mensagem que vale ao tamanho da mensagem total dividida pelo número de total de processos.

O pseudocódigo deste teste é o seguinte:

```
faça variando o tamanho das mensagens
  tempo inicial
  faça variando o contador de iterações
    all-to-all(mensagem)
  tempo final
```

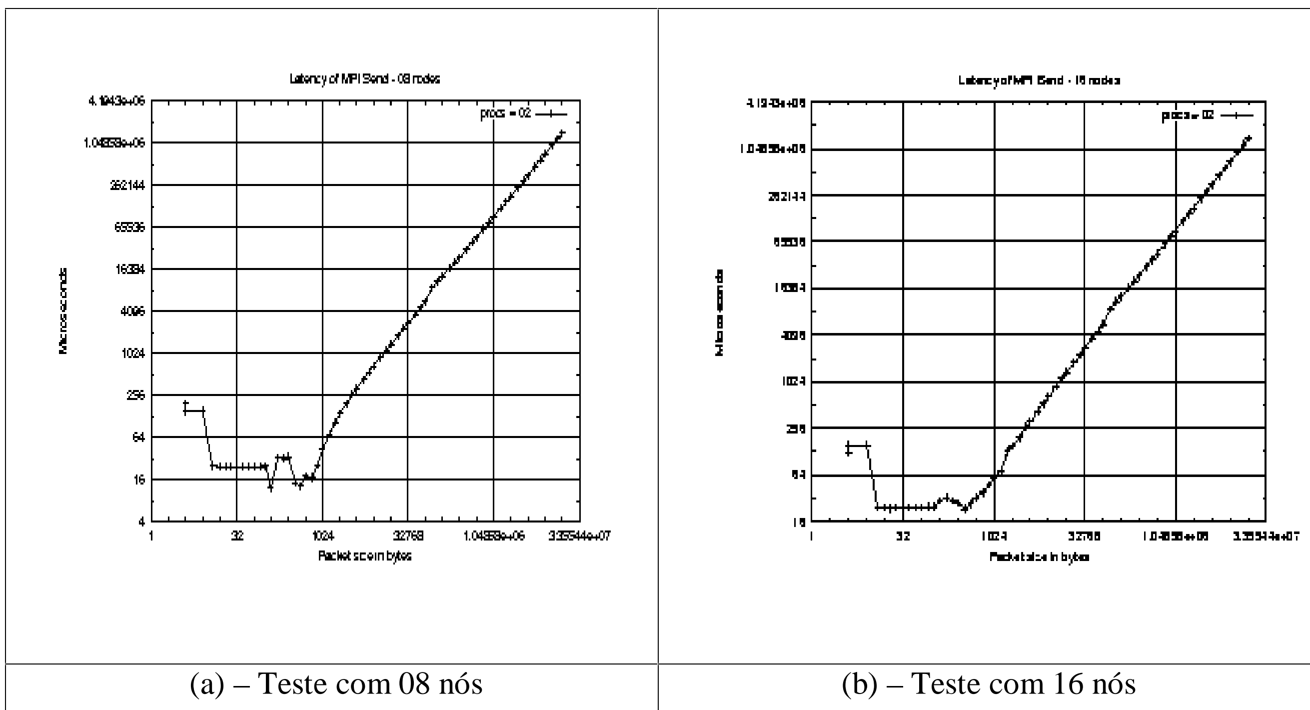
Feita a apresentação dos benchmarks, mostram-se os resultados e sua análise.

4. RESULTADOS OBTIDOS

A seguir serão apresentados os gráficos contendo os resultados dos benchmarks. Cada subseção apresenta gráficos que contém as curvas obtidas a partir dos testes realizados nos aglomerados com 8 e 16 nós.

Para as subseções de 4.1 a 4.4, o número de processos criados nos testes é de dois, já para as os demais testes, o número de processos irá variar conforme o teste realizado.

4.1. Benchmark Latency



(a) – Teste com 08 nós | (b) – Teste com 16 nós

Figura 4.1: Gráfico dos resultados obtidos com o benchmarks Latency.

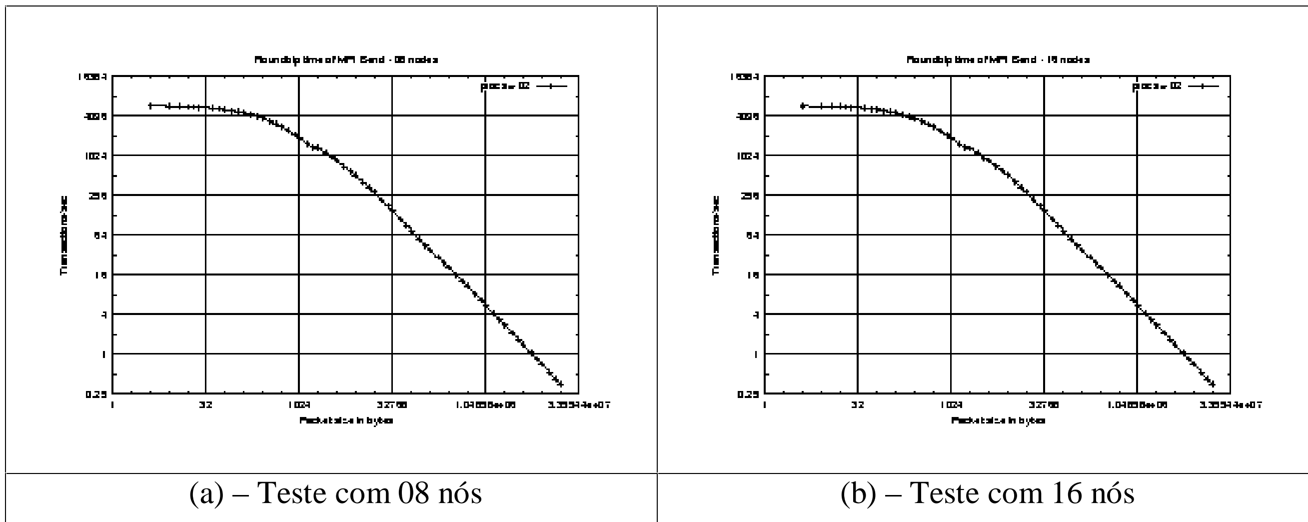
A curva apresenta no eixo das ordenadas o tempo para que a aplicação envie uma mensagem, e no eixo das abscissas tem-se o tamanho das mensagens enviadas. O gráfico apresenta uma curva obtida com o aglomerado de 8 nós e uma outra curva obtida com o aglomerado de 16 nós, os testes foram realizados com a quantia de dois processos cada.

Este benchmark é de um tipo singular no pacote MPBench, visto que ele mede o tempo da emissão da mensagem e além de ser de um tipo de aplicação com biblioteca MPI não bloqueante.

Verifica-se que os gráficos são irregulares em suas fases iniciais. As curvas passam a ter um comportamento regular para mensagens com pacotes superiores a 1024 bytes.

O tempo de latência para mensagens com tamanho inferiores a 1024 KB é um pouco maior para o gráfico da figura 4.1 (a) em relação a figura 4.1 (b), indicando menor demora na troca de mensagens para o aglomerado com 16 nós. Já para mensagens superiores a 1024 KB as curvas são praticamente iguais, indicando comportamento semelhante em termos de tempo de latência. A quantidade de nós não interferiu nos resultados. Os tempos medidos crescem a medida em que o tamanho da mensagem enviada venha a ser maior.

4.2. Roundtrip



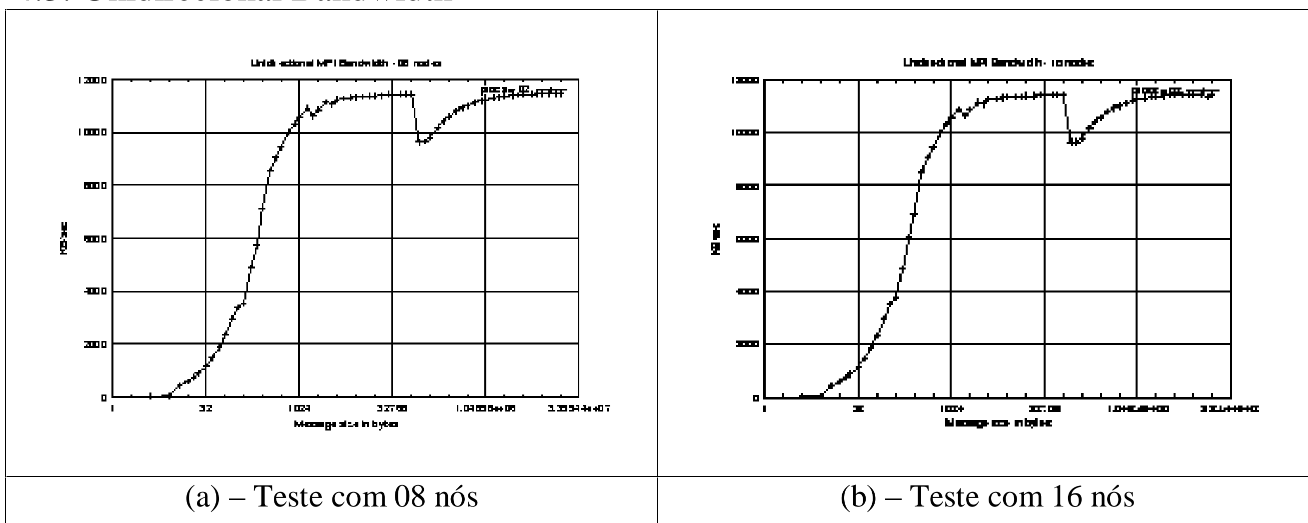
(a) – Teste com 08 nós	(b) – Teste com 16 nós

Figura 4.2: Gráfico dos resultados obtidos com o benchmarks Roundtrip.

A curva apresenta no eixo das ordenadas o número de transações por segundo, sendo essa medida muito comum para aplicações que utilizam bancos de dados e servidores. O eixo das abscissas contém o tamanho das mensagens enviadas. Assim como no *Latency*, foram utilizados os aglomerados com 8 e 16 nós, e as curvas foram geradas com as aplicações contendo dois processos cada.

Os resultados foram similares nos dois aglomerados. A curva é decrescente, onde a taxa de transações por tempo diminui a medida em que a mensagem aumenta de tamanho. Isso se deve a quanto maior for o tamanho da mensagem, maior será a demora do processo escravo a retornar um "eco" para o processo mestre. Como mencionado este teste por apresentar essa característica é também conhecido como *ping-pong*.

4.3. Unidirecional Bandwidth



(a) – Teste com 08 nós	(b) – Teste com 16 nós

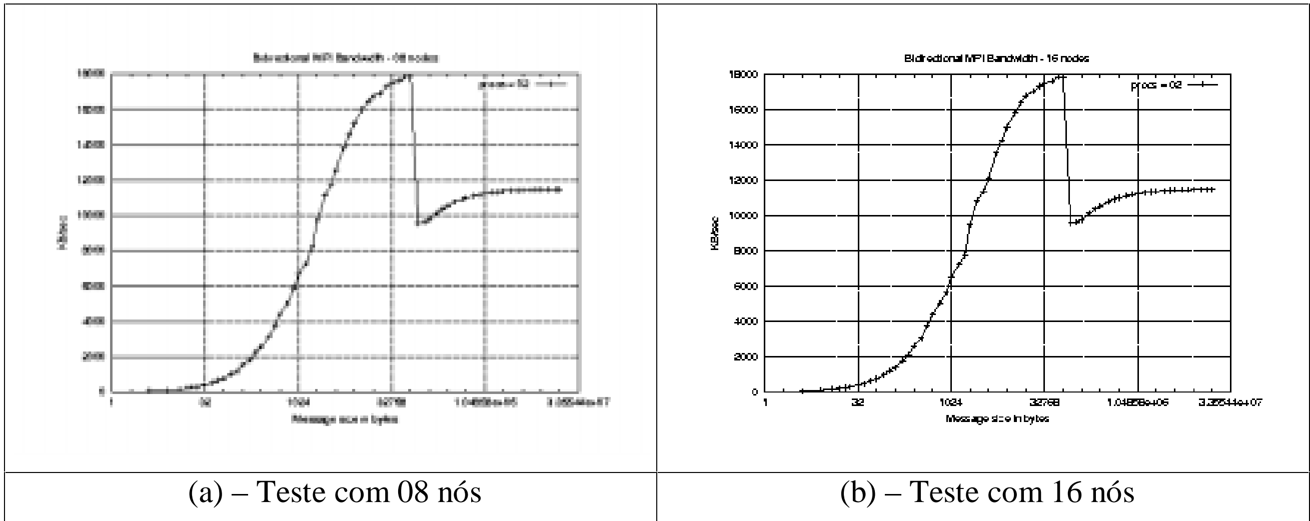
Figura 4.3: Gráfico obtido com o benchmarks unidirecional Bandwidth.

Este teste mede a largura de banda em KB/s, medidos no eixo das ordenadas e o tamanho da mensagem trocada é mostrado no eixo das abscissas. Como as curvas praticamente se sobrepõem, os resultados foram similares para os dois aglomerados.

Percebe-se que a velocidade de transmissão aumenta com o tamanho da mensagem transmitida. Isso possivelmente se deve ao melhor aproveitamento de envio de mensagem por transação, isto é, se a mensagem for muito pequena para cada envio, haverá um desperdício da

capacidade de envio de dados. Porém verifica-se que a largura de banda sofre um abrupto decréscimo para mensagens de tamanho próximo aos 128KB. Esse fato também pode estar relacionado com o tamanho da memória *cache* do processador, onde o tamanho da *cache* é de 128KB. Em seguida a curva torna a crescer com o tamanho da mensagem.

4.4. Bidirecional Bandwidth

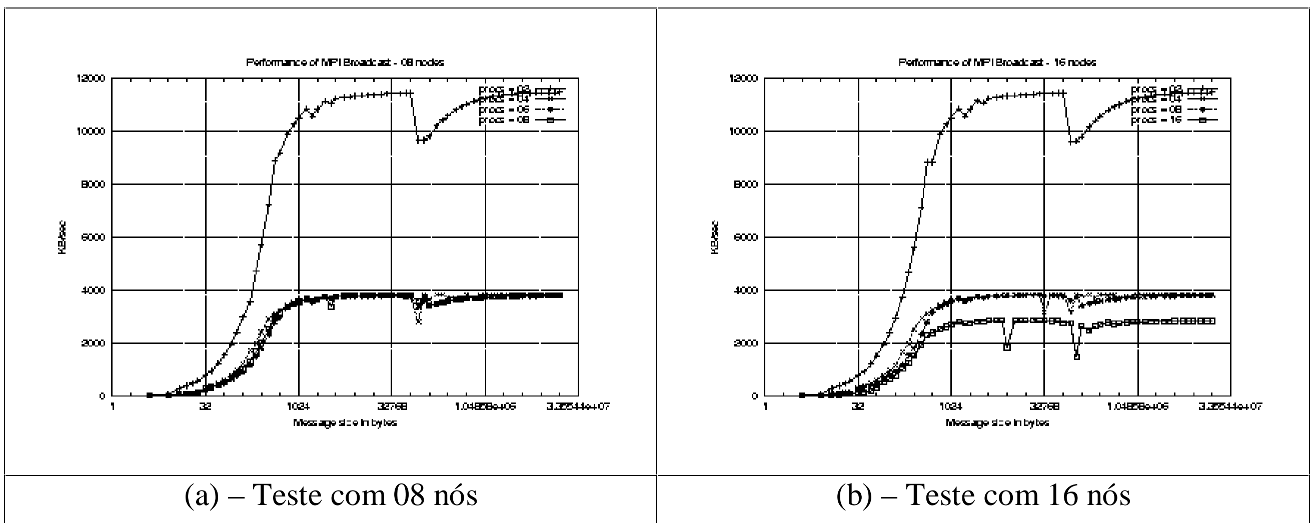


(a) – Teste com 08 nós	(b) – Teste com 16 nós

Figura 4.4: Bidirecional Bandwidth.

Como já mencionado este benchmark mede a largura de banda bidirecional. O processo mestre e o escravo executam uma troca de mensagem não blocada, portanto esta aplicação faz uso de biblioteca MPI não bloqueante. O gráfico apresenta as mesmas características do teste *Unidirecional Bandwidth*, onde se tem um comportamento das curvas praticamente idênticas. Logo se conclui que o número de nós não interferiu nos testes.

4.5. Broadcast



(a) – Teste com 08 nós	(b) – Teste com 16 nós

Figura 4.5: Broadcast

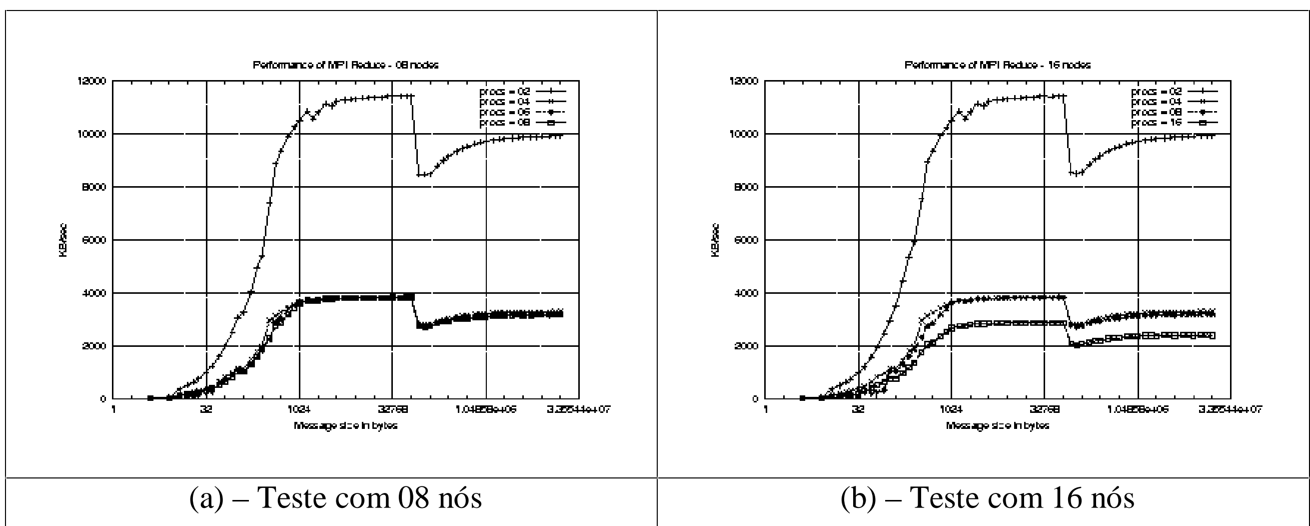
A curva referente ao *benchmark Broadcast* apresenta no eixo das ordenadas, a largura de banda dada em KB por segundos e no eixo das abscissas tem-se o tamanho das mensagens enviadas. O gráfico referente a figura 4.5 (a), com 8 nós, apresenta curvas com 2, 4, 6 e 8 processos, já o gráfico referenciado pela figura 4.5 (b), com 16 nós, contém curvas com 2, 4, 8 e 16 processos.

Esta aplicação difunde mensagens, partindo do processo *mestre* para os *escravos*. Este tipo de operação ocorre freqüentemente em diversas aplicações. Os resultados tiveram por um lado características similares às encontradas para as aplicações *Bandwidth*, porém com a diferença de que a curva gerada com dois processos obteve um desempenho muito superior às demais curvas.

O melhor desempenho deu-se para os testes com apenas 2 processos, sendo esse desempenho similar aos dois aglomerados.Para as curvas com mais que 2 processos, verificou-se que para o gráfico referente ao aglomerado com 8 nós as curvas se sobrepuseram, indicando comportamento similar. Quanto ao gráfico obtido ao aglomerado com 16 nós, nota-se que para as curvas com 4 e 8 processos, há uma sobreposição de curvas entre si, além disso, essas curvas também são similares às obtidas com o aglomerado contendo 8 nós.

A diferença reside no comportamento para a curva com 16 processos obtida no aglomerado com 16 nós. Essa curva é inferior às demais, indicando um desempenho também inferior no que se refere à largura de banda. O mau desempenho pode ser explicado pela grande concorrência entre os muitos processos criados.

4.6. Reduce

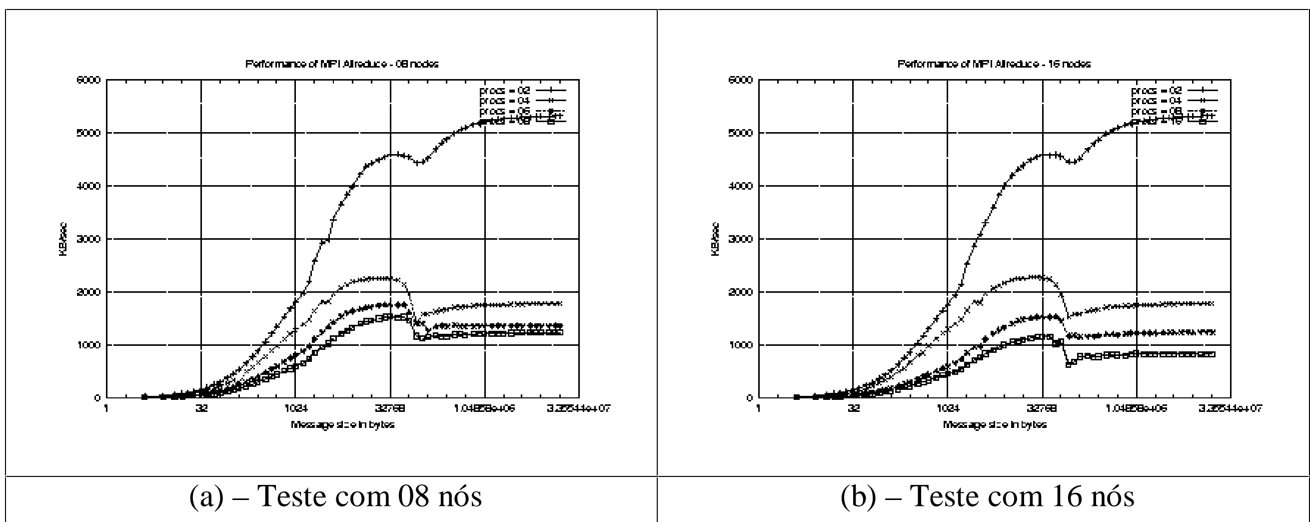


(a) – Teste com 08 nós | (b) – Teste com 16 nós

Figura 4.6: Reduce

Como já mencionado este *benchmark* é similar ao *Broadcast*, com a diferença de que o *Reduce* inverte a direção da comunicação em relação ao *Broadcast*, portanto como era de se esperar seus gráficos parecem com os gerados pelo *Broadcast.*

4.7. Allreduce



(a) – Teste com 08 nós | (b) – Teste com 16 nós

Figura 4.7: Allreduce

O *benchmark Allreduce* também apresenta no eixo das ordenadas, a largura de banda dada em KB por segundos e no eixo das abscissas, o tamanho das mensagens enviadas.

AllReduce apresenta a característica de que cada processo envia dados para os demais processos. Esse *benchmark* é uma combinação do código do *Reduce* e do *Broadcast*..

Para o gráfico gerado pelo aglomerado com 8 nós, figura 4.7 (a), existem 4 curvas geradas com 2, 4, 6 e 8 processos. Para o gráfico gerado pelo aglomerado com 16 nós, figura 4.7 (b), exitem 4 curvas com 2, 4, 8 e 16 processos.

As curvas correspondendo a 2 processos apresentam comportamento similar e foram as que indicam melhor largura de banda para os dois aglomerados.

A curva com 16 processos apresentou o pior desempenho, isso também pode decorrente pela concorrência entre os muitos processos existentes.

As demais curvas são muito parecidas entre os dois aglomerados, inclusive no que se refere a queda de desempenho na medida em que se aumenta o número de processos, como mencionado a concorrência é um fator significativo na questão da diminuição do desempenho.

Finalmente, pode-se notar o efeito da memória *cache*, onde quando o tamanho da mensagem chega próxima a sua capacidade, ocorre a conseqüente queda de desempenho, indicado pela inflexão das curvas para mensagens de tamanhos próximos a 64 KB.

4.8. Alltoall

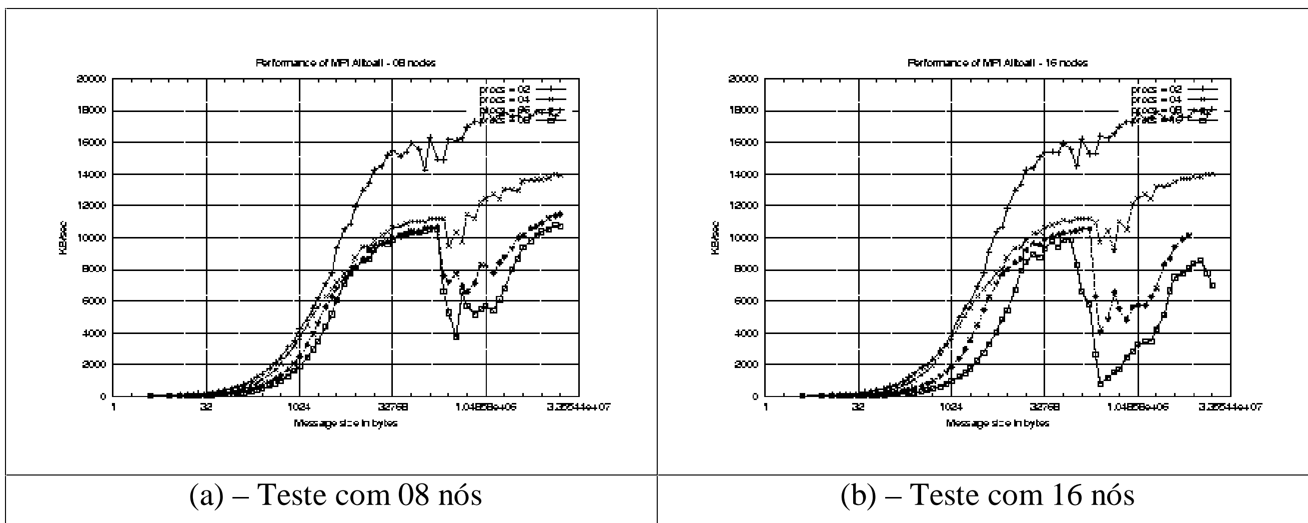


Figura 4.8: Alltoall

Este *benchmark* é semelhante ao *Allreduce*, cada processo envia uma mensagem que é igual ao tamanho total dividida pelo número de total de processos.

Nota-se que quanto maior for o número de processos, pior será o desempenho. Nesse sentido os resultados foram semelhantes nos dois aglomerados.

Os melhores desempenhos foram para os testes com 2 processos.

Com exceção da curva com 2 processos, as demais apresentam comportamento próximo até atingirem os limites do tamanho da memória *cache*. As curvas mostram acentuada diferença de desempenho para mensagens com dimensões superiores a *cache*. Verifica-se que o pior desempenho foi obtido com 8 processos para o aglomerado com 8 nós e 16 processos para o aglomerado com 16 nós.

Finalmente encerra-se a apresentação e discussão dos resultados obtidos nos testes.

7. CONCLUSÃO E DISCUSSÃO

Este trabalho comparou o desempenho de aglomerados com 8 e 16 nós com o pacote de benchmarks MPBench.

Verificou-se que para os testes com apenas 2 processos, o comportamento das curvas foi no geral similares para ambos os aglomerados. Uma explicação para esse fato seria que para um pequeno número de processos criados nestes testes, houve a geração de concorrência pelos recursos relativamente baixa.

Para os testes com mais de dois processos criados, a quantia de processos está relacionada com a queda de desempenho devido ao aumento de concorrência.

O trabalho contribui também com a apresentação de arquiteturas paralelas, aglomerados, e conceitos de programação com troca de mensagem de mensagens com bibliotecas PVM e MPI .

Finalmente destaca-se que o MPI vem tornando-se um padrão de comunicação por trocas de mensagens. Isso ocorre devido a sua portabilidade e eficiência.

8. TRABALHOS FUTUROS

Para trabalhos futuros pretende-se realizar novos testes com outros benchmarks de aglomerados ou outra possibilidade seria repetir os testes realizados com outros aglomerados.

9 – REFERÊNCIAS

[BALSA] BALSA, ANDRE D. *Linux Benchmarking HOWTO*, andrewbalsa@usa.net http://metalab.unc.edu/LDP , August 1997.

[HPCC] http://www.hpcc.ecs.soton.ac.uk/RandD/genesis/genesis.html

[ICL] http://icl.cs.utk.edu/projects/llcbench

[LIIN] http://liinwww.ira.uka.de/~skampi/

[MPBEN] http://icl.cs.utk.edu/projects/llcbench/mpbench.html

[NAS] http://www.nas.nasa.gov/Software/NPB/

[PDPL] http://pdplab.trc.rwcp.or.jp/pdperf/Omni/benchmarks/NPB

[SCAL] http://www.scali.com/performance/index.html

[TANE1] TANENBAUM, A. S., *Distributed Operating System*s , Prentice Hall, 1995.

[TANEN2] TANENBAUM, A S. , Structured Computer Organization, Prentice Hall do Brasil, 1990.

[TATH] http://tathy.comp.ita.cta.br/~sobreira/paralela.htm.

[UFSC1] http://www.inf.ufsc.br/~zancanel/Aurora.htm.

[UFRGS1] http://www.inf.ufrgs.br/procpar/disc/cmp134/trabs/T1/981/BWulf/CMP-134-T1.html

[UFSCS2] http://www.inf.ufrgs.br/gpesquisa/procpar/disc/cmp157/trabalhos/sem98-2/commkernel/