MARCELO LOYOLA STIVAL

AVALIAÇÃO DE DESEMPENHO EM AGLOMERADOS DE PCS INTERLIGADOS POR ETHERNET

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Roberto André Hexsel

CURITIBA

2006

MARCELO LOYOLA STIVAL

AVALIAÇÃO DE DESEMPENHO EM AGLOMERADOS DE PCS INTERLIGADOS POR ETHERNET

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Roberto André Hexsel

CURITIBA

2006

MARCELO LOYOLA STIVAL

AVALIAÇÃO DE DESEMPENHO EM AGLOMERADOS DE PCS INTERLIGADOS POR ETHERNET

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comissão formada pelos professores:

Orientador: Prof. Roberto André Hexsel
Departamento de Informática, UFPR

Prof. Dr. Aldri Luiz dos Santos
Departamento de Computação, UFC

Prof. Dr. Luiz Carlos Pessoa Albini
Departamento de Informática, UFPR

Curitiba, 15 de stembro de 2006

AGRADECIMENTOS

Meus sinceros agradecimentos ao orientador, professor Dr. Roberto A Hexsel, que prestou indispensável auxílio durante o desenvolvimento deste trabalho. Agradeço também ao professor Carlos Carvalho, responsável pelo ambiente computacional (Nautilus), por ter disponibilizado os recursos e apoiado na configuração do sistema. Estendo meus agradecimentos aos colegas Leslie Harlley Watter e Sérgio Luiz Marques Filho por terem cedido parte de seus trabalhos e apoiado o desenvolvimento deste estudo, e também a todos que contribuíram de alguma forma.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho apresenta uma avaliação de desempenho de um aglomerado de PCs interligados por rede Ethernet, que empregam o padrão *MPI* para troca de mensagens. São discutidos diversos aspectos que podem guiar o projeto de aplicações paralelas, assim como comparados diferentes parâmetros – outros além da largura de banda e da latência, normalmente usados na avaliação de sistemas de comunicação. A análise apresentada neste trabalho tem como objetivo medir o desempenho que é exposto para as aplicações pela biblioteca de alto nível, e que resulta de complexas interações de diversos componentes do sistema. O modelo *LogP* é usado como base para a análise desenvolvida neste trabalho, e algumas extensões do modelo propostas anteriormente também são discutidas.

São definidos os testes usados para medição dos parâmetros que caracterizam o sistema, destacando-se a necessidade de considerar a semântica das primitivas MPI para obter resultados mais precisos. Técnicas de programação também são avaliadas, em particular técnicas de sobreposição de comunicação com computação – que têm papel fundamental em ambientes em que os custos de comunicação são elevados.

Algumas das ferramentas usadas para medir o desempenho do sistema de comunicação foram estendidas com o desenvolvimento de novos programas de teste, para que seja possível avaliar a sobreposição de comunicação com computação, e a sobrecarga de processamento envolvido com comunicação.

Os testes mostram que o desempenho exposto pelo MPI para as aplicações fica muito abaixo da capacidade nominal da rede – a largura de banda é da ordem de 350 Mbps em uma rede de 1 Gbps. São avaliadas duas implementações do padrão MPI, *MPICH2* e *OPENMPI*, bem como o protocolo *LLC* como camada de transporte da biblioteca *OPENMPI*, em substituição ao TCP/IP com o objetivo de diminuir a sobrecarga de processamento. Para mensagens pequenas, o desempenho da biblioteca *OPENMPI* com protocolo TCP é pior que o desempenho da biblioteca *MPICH2*. A diferença é da ordem de 17% para mensagens entre 1 e 32 bytes e diminui com o tamanho da mensagem até

que tornam-se similares para mensagens de 8kB. Para mensagens maiores que 8kB, o desempenho da biblioteca *OPENMPI* é melhor, e a diferença chega a 7% com mensagens de 256kB. O protocolo *LLC* apresentou bom ganho de desempenho em relação ao TCP/IP, que para mensagens pequenas é da ordem de 7% e chega a 12% com mensagens de 16kB.

Dois *kernels* científicos, *FFT* e *Radix*, também são usados na análise de desempenho com o objetivo de avaliar a eficiência de técnicas de sobreposição da comunicação com computação. Foram comparadas versões *multi-threaded* e versões baseadas em primitivas não-bloqueantes. O *kernel* Radix tem padrão de comunicação irregular, e obteve ganho de desempenho da ordem de 50% com o uso de múltiplos *threads*. O *FFT* mostrou-se mais eficiente com o uso de primitivas não-bloqueantes na maioria dos casos, mas os resultados dependem do tamanho da entrada e do número de processos.

ABSTRACT

This work presents a low-level performance evaluation of an Ethernet cluster of workstations that uses MPI as a message passing environment. The evaluation focuses on the communication system, and considers a set of parameters wider than the usual bandwidth and latency. The main objective of this work is to measure the performance exposed to the applications by the high-level software library, which depends on the complex interactions of the components of the system. The LogP model and some extensions proposed elsewhere are used as a basis for the analysis presented here.

The tests used to measure the parameters that characterize the system are described, and the importance of taking into account the behavior of MPI primitives is emphasized. Overlap of communication with computation is fundamental to improve the performance of parallel applications targeted to clusters of workstations because of the communication being so expensive in these systems. This work evaluates techniques that provide some level of overlap of communication with computation, and describes the tests used to measure the opportunity of overlap exposed by the system. A set of tools were used to evaluate the system with focus on communication costs, and new tests were implemented to assess communication overlap and software overhead.

The results show that the performance exposed by the high-level library is well below network capacity. The bandwidth attainable by the applications on top of MPI is around 350Mbps on a 1Gbps network. Two implementations of the MPI standard are evaluated (*MPICH2* and *OPENMPI*), as well as the LLC and TCP protocols as the transport layer of *OPENMPI*.

When using short messages, the *MPICH2* library performs better than *OPENMPI* with the TCP protocol. The difference is about 17% for messages of length between 1 and 32 bytes and it becomes smaller when message size increases. The performance is virtually the same for messages of 8kB. For messages longer than 8kB, *OPENMPI* performs better, with a gain of around 7% for messages of 256kB. The LLC protocol shows a large gain in

performance when compared to TCP as the transport layer of *OPENMPI*. The difference ranges from 7% for short messages to 12% for messages of 16kB.

Two scientific kernels, *FFT* and *Radix Sort*, were used to compare a multi-threaded approach to the use of non-blocking operations to assess the overlap of communication with computation. The Radix Sort kernel has an irregular communication pattern, and shows an improvement close to 50% with a multi-threaded approach. The FFT kernel performs better with the non-blocking operations on most cases, but its performance depends on the problem size and the number of process.

CAPÍTULO 1

INTRODUÇÃO

Aglomerados de PCs são formados por computadores pessoais interligados por uma rede local, e têm sido amplamente adotados como ambientes de computação paralela com memória distribuída. Por empregar componentes de uso geral produzidos em larga escala, estes ambientes oferecem uma solução de baixo custo e de bom desempenho, além de apresentarem facilidade de expansão e atualização do *hardware*. Em particular, o padrão Ethernet é bastante difundido e apresenta uma boa relação custo desempenho.

Ao comparar o desempenho das CPUs com a capacidade das redes locais nos últimos anos observa-se que o desempenho destes componentes cresce a taxas similares, porém de maneiras muito diferentes. Novas gerações de processadores são disponibilizadas no mercado freqüentemente, enquanto novas tecnologias de rede surgem em intervalos maiores de tempo.

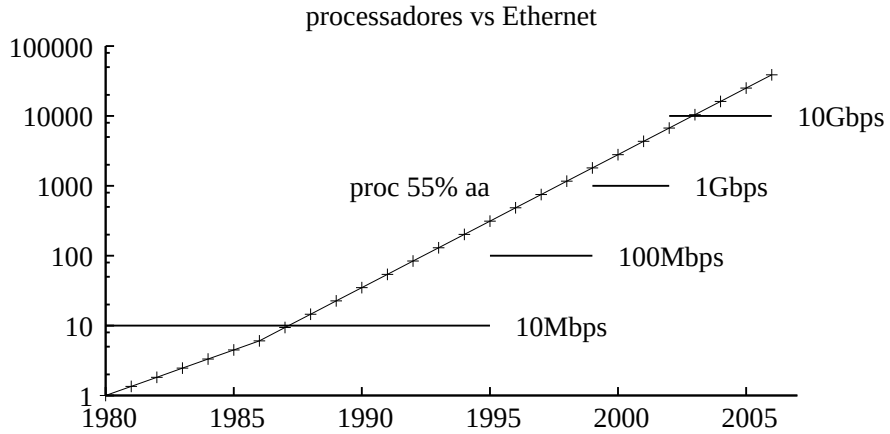


Figura 1.1: Evolução da Ethernet comparada com a evolução dos processadores.

Geralmente, quando uma nova tecnologia de rede é introduzida no mercado, os processadores disponíveis no momento não são capazes de ocupar toda a capacidade da rede. Antes do surgimento da próxima geração de rede, o poder computacional aumenta gradativamente, até que os processadores tenham poder computacional para ocupar toda a capacidade da rede. Este cenário muda quando uma nova tecnologia de rede

é disponibilizada, e a tecnologia dos processadores inicialmente não garante a utilização total da capacidade da rede. O desempenho das CPUs cresce a uma taxa entre 35 e 55% ao ano, enquanto que uma nova geração da Ethernet é disponibilizada em período maior de tempo, tipicamente de quatro anos, e têm sua capacidade multiplicada por 10 a cada nova geração [23].

O gráfico da figura 1.1 mostra a relação entre o crescimento do poder computacional das CPUs e a capacidade de redes Ethernet ao longo do tempo. O gráfico está numa escala hipotética (1 MIPS sendo proporcional a 1 Mbps) que permite comparar visualmente as taxas de melhoria nas duas tecnologias. Com base no gráfico, no período de 1995 a 1999 a rede Ethernet disponível tinha banda de 100 Mbps. Neste mesmo período o desempenho das CPUs cresceu 55% ao ano. Até que em 1999 uma nova geração da Ethernet foi disponibilizada com banda de 1 Gbps. A partir de 2000, a taxa de crescimento no desempenho das CPUs caiu de 55% a.a. para a faixa de 30 a 35% a.a. [23].

A transição entre entre duas gerações de Ethernet causa uma mudança brusca na configuração de hardware dos aglomerados de PCs, e conseqüentemente na eficiência com que as aplicações coordenam o uso dos recursos. Uma aplicação projetada para um determinado patamar de capacidade pode não executar com eficiência quando uma nova geração de rede é implantada.

Algumas técnicas de programação empregadas em aplicações paralelas podem ser mais interessantes dependendo dos patamares relativos de evolução da tecnologia disponível no momento. Em aglomerados de PCs, o custo de comunicação entre as estações é relativamente elevado quando comparado à capacidade de processamento e ao tempo de acesso à memória local – e tem forte influência no desempenho global das aplicações. Uma das técnicas empregadas para amortizar os custos de comunicação consiste em sobrepor comunicação com tarefas de computação. Esta e outras técnicas dependem da relação entre a capacidade de processamento e a capacidade da rede – e portanto dos patamares de evolução da tecnologia.

Do ponto de vista do projeto de aplicações, interessa o desempenho que é exposto para as aplicações, que depende da combinação dos diversos componentes do sistema

de comunicação, tais como biblioteca de alto nível e protocolos de comunicação – e não apenas da capacidade nominal da rede. O grande número de componentes que influenciam no desempenho do sistema de comunicação torna a análise ainda mais complexa. O uso de modelos auxilia neste processo, pois permite caracterizar um sistema com base em uns poucos parâmetros, abstraindo parte da complexidade do sistema. O modelo *LogP* representa de forma realista os custos envolvidos na comunicação, o que torna o modelo especialmente útil em aglomerados de PCs, por que estes apresentam custos de comunicação elevados.

1.1 Objetivos e Contribuições

A avaliação de desempenho em sistemas de computação paralela deve considerar aspectos relevantes para o desempenho da aplicação. Os custos de comunicação não podem ser avaliados apenas com base em parâmetros como largura de banda e latência, normalmente usados na avaliação de redes de interconexão.

Um modelo do sistema com enfoque na comunicação auxilia no projeto de aplicações paralelas, bem como permite avaliar o desempenho com base em um conjunto de características relevantes para o desempenho das aplicações.

A plataforma alvo adotada neste trabalho consiste em aglomerados de PCs ligados por rede Ethernet, que executam aplicativos baseados no padrão *Message Passing Interface (MPI)*. Este trabalho tem como objetivo contribuir para o projeto de aplicações paralelas, bem como para a análise de desempenho de sistemas de comunicação.

Este trabalho contribui com uma metodologia para a análise de desempenho em aglomerados de PCs, composta por um conjunto de parâmetros que representam o sistema com enfoque na comunicação. O conjunto de parâmetros deve ser suficientemente representativo para avaliação de diferentes sistemas de comunicação, e para auxiliar o projeto de aplicações no que tange a integração da comunicação em aplicações paralelas.

Este trabalho emprega algumas ferramentas de domínio público e também contribui com a construção de novos testes que são usados para medir os parâmetros definidos pela metodologia diretamente no sistema.

Outra contribuição deste trabalho consiste em uma avaliação de técnicas de programação que promovem sobreposição da comunicação com computação.

A avaliação de desempenho de um aglomerado de PCs mantido pela Universidade Federal do Paraná também faz parte das contribuições deste trabalho.

1.2 Organização do Trabalho

O texto está organizado da seguinte maneira. O Capítulo 2 descreve o padrão *MPI* e modelos de computação paralela empregados no projeto de algoritmos.

No Capítulo 3 são discutidos diversos parâmetros que devem ser considerados no projeto das aplicações e na comparação de diferentes sistemas de comunicação. São discutidas algumas variantes do modelo LogP e suas limitações, bem como a extensão do modelo adotada neste trabalho. Além da definição dos parâmetros, são descritos os testes usados para medir tais parâmetros diretamente no sistema através de *microbenchmarks*.

A avaliação de desempenho desenvolvida neste trabalho é apresentada no Capítulo 4. São descritos os três programas (*microbenchmarks*) usados na avaliação do sistema, bem como os novos testes implementados neste trabalho para compor a análise. Foram usados os programas *NetPIPE*, *LogP Benchmark* e *Perftest*. Além destes, foram implementados testes para medir a sobrecarga de processamento envolvida na comunicação e a oportunidade de sobreposição de comunicação com computação exposta por diferentes bibliotecas *MPI*. Estes testes foram implementados com base na estrutura do Perftest, e estão descritos juntamente com aquela ferramenta.

Foram avaliados dois *kernels* científicos, *FFT* e *Radix Sort*, que são apresentados na Seção 4.3. Neste trabalho foram geradas novas implementações destes *kernels* para comparação de diferentes técnicas de programação. Em particular, os *kernels* são usados para avaliar a técnica de sobreposição de comunicação com computação através de abordagens *multi-threaded* e com base em primitivas não-bloqueantes. No Capítulo 4 são definidas as métricas, descritos os mecanismos de avaliação, e então são apresentados os resultados experimentais. Esta organização foi escolhida por causa da grande variedade de métricas exploradas. O Capítulo 5 apresenta as conclusões.

CAPÍTULO 2

COMPUTAÇÃO PARALELA COM TROCA DE MENSAGENS

2.1 MPI

O paradigma de comunicação baseado em troca de mensagens é amplamente utilizado em sistemas com memória distribuída. O padrão mais popular é o *Message Passing Interface* (MPI), que conta com diversas implementações destinadas a uma grande variedade de arquiteturas. O padrão MPI é definido pelo MPI Forum [1], um grupo que tem participação de diversas organizações, da academia e da indústria.

A primeira versão do padrão MPI, *MPI-1*, define funções de comunicação ponto-a-ponto, operações coletivas e definições de tipos de dados [14]. O padrão *MPI-2* estende as funcionalidades do *MPI-1* abrangendo criação e gerenciamento de processos, acesso remoto à memória (*one-sided communication*), e operações de entrada e saída paralelas [22].

As primitivas de comunicação ponto-a-ponto possuem versões bloqueantes e não-bloqueantes, além de modos de comunicação síncronos e assíncronos. As operações coletivas definidas pelo padrão são: *Broadcast, Scatter, Gatter, Reduce e All-to-all*, que implementam as operações de difusão (um-para-todos), espalhamento, coleta, redução e todos-para-todos, respectivamente.

Os processos são organizados em grupos ordenados, em que cada processo é identificado por um número seqüencial (*rank*). Os processos participam de um ou mais *contextos de comunicação*. Um contexto de comunicação agrupa processos que podem se comunicar através de mensagens, mas processos que não pertencem a um mesmo contexto não podem trocar mensagens entre si.

[1] http://www.mpi-forum.org/

2.1.1 Comunicação ponto-a-ponto

Esta Seção descreve algumas primitivas de comunicação ponto-a-ponto do padrão MPI. O padrão MPI define a semântica das primitivas de comunicação, mas não define como devem ser implementadas. A Seção 2.1.3 descreve os protocolos usualmente empregados pelas implementações.

Primitivas de Envio

O padrão MPI define primitivas de envio de mensagens bloqueantes e não-bloqueantes. As primitivas de envio bloqueantes retornam apenas quando a mensagem for passada adiante, e o *buffer* que mantém os dados a transmitir puder ser reusado pela aplicação. O retorno de uma primitiva bloqueante não está vinculado necessariamente ao recebimento da mensagem no processo de destino. Se a mensagem for copiada para um *buffer* do sistema localmente, a primitiva retorna para o fluxo normal do programa ao final da cópia e não ao final da transmissão.

Primitivas de envio não-bloqueantes iniciam a transferência de uma mensagem e retornam imediatamente, antes de completar a operação. O *buffer* não pode ser reusado até que a operação seja completada. O programador deve garantir que a operação completou, conforme descrito adiante.

Primitivas de Recebimento

As primitivas de recebimento de mensagens também possuem versões bloqueantes e não-bloqueantes. A versão bloqueante de recebimento retorna apenas quando a mensagem correspondente for recebida e copiada para o *buffer* do usuário. O programa fica bloqueado até que a operação complete.

A primitiva de recebimento não-bloqueante solicita a recepção da mensagem e retorna para o programa sem completar a operação. O programa então pode verificar se a operação foi completada através de primitivas definidas pelo padrão, discutidas a seguir.

Teste de término

As primitivas MPI_Wait e MPI_Test são usadas em conjunto com operações não-bloqueantes, tanto de envio quanto de recebimento.

A primitiva MPI_Wait é usada para aguardar o término de uma operação iniciada de forma não-bloqueante. O programa fica bloqueado até que a operação termine, e então retorna com o status da operação.

A primitiva MPI_Test verifica se a operação iniciada completou e retorna imediatamente, sem esperar o término da operação. Um parâmetro de saída é usado para informar ao programa se a operação já foi encerrada ou está em andamento. Esta operação pode ser usada repetidamente, intercalada ou não com outra tarefa, até que a operação seja completada.

2.1.2 Modos de comunicação

O padrão define quatro modos de comunicação que podem ser combinados com operações de envio bloqueantes e não-bloqueantes: padrão, síncrono, *buffered* e *ready*.

O *modo padrão* de comunicação deixa para a implementação do MPI decidir se a mensagem será armazenada em um *buffer* intermediário ou será transmitida diretamente para o destinatário. Se a mensagem for armazenada localmente em um *buffer* intermediário, a primitiva de envio completa sua execução independentemente de haver uma operação de recebimento associada. Porém, se a implementação não usar um *buffer* intermediário, a operação só se completa quando a operação de recebimento associada for iniciada. A implementação geralmente decide entre uma das abordagens em tempo de execução com base no tamanho da mensagem, como descrito na Seção 2.1.3.

O *modo síncrono* determina que a operação só completa quando a mensagem começar a ser recebida no destinatário. Neste caso, o término da operação indica que o *buffer* pode ser reusado, e que a operação de recebimento correspondente já foi iniciada.

O padrão define o modo de comunicação *buffered*, no qual a mensagem deve ser copiada para um *buffer* intermediário sempre que não houver uma operação de recebimento correspondente. Entretanto, o *buffer* deve ser disponibilizado pela aplicação.

O modo de comunicação *ready*, ou pronto para receber, determina que o envio só pode ser iniciado pela aplicação quando o destinatário estiver pronto para receber – a operação de recebimento associada deve ter sido configurada. Este modo de comunicação permite uma implementação mais eficiente, baseada na hipótese de que o receptor já está preparado para receber a mensagem. Se a mensagem for enviada sem que o receptor esteja preparado para recebê-la imediatamente, ocorre um erro.

2.1.3 Protocolos

As implementações do padrão MPI usualmente empregam protocolos distintos para troca de mensagens longas e curtas. Mensagens curtas empregam o *protocolo Eager*, no qual a mensagem é enviada de forma assíncrona. O protocolo Eager supõe que o receptor pode receber a mensagem, mesmo que não haja uma operação de recebimento correspondente. As mensagens recebidas são armazenadas em buffers intermediários, e em seguida são copiadas para o *buffer* da aplicação fornecido através da operação de recebimento correspondente.

Para mensagens longas é usado o *protocolo rendezvous*, no qual há um passo adicional de sincronismo, de forma que a mensagem só é enviada mediante aceitação pelo receptor. Desta forma, o receptor não precisa necessariamente armazenar a mensagem em um *buffer* intermediário, e a implementação pode copiar a mensagem diretamente para o *buffer* do usuário.

As mensagens MPI são compostas de um envelope que contém informações de controle, e pelos dados da mensagem. Algumas implementações empregam o *protocolo Short* que envia os dados junto com o envelope da mensagem. Como no protocolo Eager, a mensagem é enviada supondo que o receptor pode recebê-la imediatamente.

O limite a partir do qual uma mensagem é considerada curta ou longa varia em cada implementação, e em algumas implementações pode ser ajustado pelo usuário. A implementação OPENMPI tem como padrão um limite máximo de 64KB para uso do protocolo Eager, enquanto que a implementação MPICH2 usa como padrão um limite de 16KB.

Suporte a múltiplos threads

O padrão MPI prevê suporte a múltiplos *threads*, que deve ser requisitado pela aplicação na inicialização atavés do método MPI_Init_thread(). São suportados 4 níveis descritos a seguir.

MPI_THREAD_SINGLE Apenas um *thread* de execução.

MPI_THREAD_FUNNELED A aplicação pode ser multi-threaded, mas apenas o *thread* principal faz chamadas à funções MPI.

MPI_THREAD_SERIALIZED A aplicação pode ser multi-threaded e múltiplos *threads* podem chamar funções do MPI, mas não de forma concorrente. É responsabilidade da aplicação serializar as chamadas.

MPI_THREAD_MULTIPLE Múltiplos *threads* podem chamar funções MPI sem restrições.

2.2 Modelos de Computação Paralela

Esta Seção descreve brevemente alguns modelos de computação paralela, e em seguida descreve o modelo LogP, que é empregado neste trabalho.

O uso de modelos de computação paralela permite abstrair detalhes de diferentes arquiteturas através de parâmetros que descrevem características relevantes do sistema. Os parâmetros que caracterizam uma configuração de hardware servem de base para o projeto de aplicações, além de prover estimativas de desempenho em arquiteturas diferentes.

Em [25] e [24] os autores propõem um modelo baseado em operações abstratas que representam a execução de um programa. As mesmas operações abstratas são usadas para caracterizar um máquina. Combinando a representação do programa em função das operações abstratas, com as medidas de desempenho das operações em uma determinada máquina, pode-se inferir o tempo de execução do programa.

O modelo Parallel Random Access Machine (PRAM) [16] é um modelo teórico usado no projeto de algoritmos paralelos. O PRAM representa uma máquina abstrata constituída de um número arbitrário de processadores e memória compartilhada. Todo processador tem acesso a qualquer valor na memória ao custo de uma unidade de tempo. O modelo abstrai aspectos como comunicação e sincronização, mas explora situações de concorrência nos algoritmos.

O modelo LogP [9][8] caracteriza um sistema com base em quatro parâmetros: latência da comunicação (L), *overhead* da comunicação (o), largura de banda (g), e o número de unidades de processamento (P). O LogP representa de forma realista os custos envolvidos na comunicação entre as unidades de processamento.

Em aglomerados de estações de trabalho com memória distribuída o custo da comunicação é relativamente alto, quando comparado a acessos à memória local. Nestes ambientes a representação dos custos envolvidos na comunicação é fundamental para projetar algoritmos eficientes, além de ser adequado para avaliação do desempenho do subsistema de comunicação.

O modelo LogP é adotado neste trabalho como uma ferramenta para caracterizar o sistema de comunicação e auxiliar no projeto de algoritmos. O modelo LogP, bem como algumas variantes do modelo, são descritos a seguir.

LogP

O modelo LogP [9][8] caracteriza um sistema com base em quatro parâmetros: Latência da comunicação (L),*overhead* da comunicação (o), largura de banda (g), e o número de unidades de processamento (P). A latência, overhead, e *gap* são expressos em função do ciclo do processador. O modelo considera mensagens pequenas, em que o *overhead* de comunicação incide sobre cada byte da mensagem.

Em algumas arquiteturas existe suporte de hardware para transferência de mensagens longas. Por exemplo, dispositivos de DMA integrados à interface de rede realizam a transferência da mensagem sem envolver a CPU. A CPU principal é envolvida apenas na preparação do envio, de forma que o custo não cresce em função do tamanho da mensagem.

Em [9] sugere-se que o uso de um processador dedicado à comunicação, no melhor caso, dobra a capacidade do nó de processamento, e pode ser modelado considerando dois processadores em cada nó.

O modelo LogGP proposto em [2] é uma extensão do LogP, em que o parâmetro adicional G captura a largura de banda para mensagens longas. O parâmetro G representa o *gap* por byte para mensagens longas.

Os parâmetros do modelo LogP podem ser medidos diretamente do sistema através de *microbenchmarks*. Em [7] está descrita uma metodologia para levantar os parâmetros de desempenho do subsistema de comunicação segundo o modelo conceitual LogP. Para representar mensagens longas os autores propõem que os parâmetros sejam expressos em função do tamanho da mensagem, e não apenas considerando um custo fixo de inicialização, que é independente do tamanho.

Bibliotecas de comunicação baseadas em troca de mensagens usam protocolos distintos dependendo do tamanho da mensagem, como ocorre com o padrão MPI. Para troca de mensagens longas frequentemente opta-se por um protocolo síncrono, que introduz um novo componente ao custo de transferência da mensagem. O LogGPS [18] é uma extensão do LogGP que captura o custo de sincronismo presente nas trocas de mensagens longas. O parâmetro G captura o custo por byte, e o parâmetro S, introduzido pelo modelo LogGPS, representa o custo do sincronismo.

2.3 Threads

Um processo é uma abstração de um programa em execução composto por um fluxo de instruções, espaço de endereçamento, pilha de execução, contador de programa e pelo estado corrente da execução.

Cada processo pode conter um ou mais *threads de execução*. Cada *thread* possui um fluxo de instruções, contador de programa e pilha de execução independentes, porém compartilham o mesmo espaço de endereçamento. O *escalonador* aloca os *threads* para execução no processador de acordo com o estado do sistema.

Existem duas estratégias para implementação de threads: *threads em nível de sistema* e

threads em nível de usuário. As *threads* em nível do sistema são gerenciadas pelo Sistema Operacional. As operações relativas ao gerenciamento das *threads* são implementadas através de chamadas do sistema operacional (*system calls*), o que envolve troca de contexto entre o espaço de usuário e o espaço de sistema. As *threads* em nível de usuário são gerenciadas por uma biblioteca, sem envolver o sistema operacional. Esta abordagem evita trocas de contexto entre o espaço de usuário e de sistema, e permite usar políticas de escalonamento diferenciadas de acordo com a necessidade da aplicação.

Neste trabalho são avaliadas técnicas de sobreposição de comunicação através do uso de múltiplos *threads*. na Seção 4.3 são apresentados dois programas (*kernels*) com implementações *multi-threaded*.

CAPÍTULO 3

PROJETO DE APLICAÇÕES PARALELAS

Neste Capítulo são discutidos alguns aspectos relacionados ao projeto das aplicações paralelas, bem como à avaliação de desempenho de sistemas de comunicação. São descritos parâmetros que podem guiar o projeto de aplicações, bem como testes para medir tais parâmetros diretamente no sistema.

O projeto de aplicações paralelas pode considerar algumas características da plataforma alvo com o objetivo de usar adequadamente os recursos e obter melhor desempenho. No âmbito de aglomerados de PCs ligados por uma rede local, o custo da comunicação é relativamente alto quando comparado à capacidade de processamento da *CPU* e ao tempo de acesso à memória local. Portanto, a integração da comunicação em aplicações paralelas têm um papel fundamental no desempenho e eficiência, em particular nas aplicações que fazem uso intensivo de comunicação.

Uma compreensão adequada dos custos envolvidos na comunicação é essencial para o projeto de aplicações paralelas eficientes. O desempenho do sistema de comunicação depende de uma combinação de diversos componentes, como biblioteca de alto nível, protocolos e dispositivo de rede. Se analisado com uma granularidade mais fina, diversos componentes podem ser adicionados à lista, tornando a análise ainda mais complexa.

O uso de modelos auxilia neste processo, pois permite caracterizar um ambiente com base em uns poucos parâmetros, abstraindo parte da complexidade do sistema. O modelo LogP representa de forma realista os custos envolvidos na comunicação, o que torna o modelo especialmente útil para o projeto de aplicações destinadas a aglomerados de PCs, em que a integração da comunicação tem grande influência no desempenho global da aplicação.

O LogP e algumas extensões do modelo são usados neste trabalho em conjunto com outros parâmetros para compor a avaliação do sistema de comunicação e guiar o projeto

de aplicações. A Seção 3.1 descreve o modelo baseado no LogP e algumas variantes. Em seguida os parâmetros são discutidos separadamente, bem como os testes necessários para obter as medidas e aspectos relacionados ao padrão MPI.

A abstração oferecida por uma biblioteca de alto nível tem desempenho e propriedades que emergem da combinação dos diversos componentes do sistema, e que devem ser consideradas no projeto. Em geral, as bibliotecas oferecem primitivas com implementações diferentes para realizar a mesma operação. Por exemplo, o padrão MPI provê primitivas de envio com diferentes modos de comunicação além de versões bloqueantes e não-bloqueantes. Medir o desempenho e conhecer a semântica destas primitivas é essencial para integração da comunicação de forma eficiente.

3.1 Modelos

O modelo LogP como proposto originalmente define parâmetros que representam os custos de comunicação para mensagens pequenas e de tamanho fixo [10]. Em geral, os sistemas possuem suporte especial para transmissão de grandes quantidades de dados *(Bulk Transfer)*.

A extensão do modelo proposta em [2] incorpora um novo parâmetro para capturar o custo por byte para transmissão de mensagens grandes, prevendo suporte especial do sistema de comunicação. O parâmetro G representa o custo adicional para cada byte de uma mensagem grande. No LogGP a distinção entre mensagem pequena e grande não é relacionada com a distinção feita pelo MPI entre mensagens curtas e longas. O parâmetro G representa o custo por byte, de forma que é capaz de modelar um sistema no qual o parâmetro cresce linearmente com o tamanho da mensagem, para mensagens maiores que um determinado valor.

No padrão MPI, mensagens longas e curtas são transmitidas usando protocolos distintos que podem apresentar medidas diferentes do parâmetro G. O crescimento pode ser linear nos intervalos em que as mensagens são consideradas longas ou curtas, mas a taxa de crescimento do parâmetro não é a mesma para os dois intervalos.

Em [6] os autores usam parâmetros adicionais para representar o *gap* de acordo com o

tamanho da mensagem. Os autores identificaram três intervalos para os quais o valor do parâmetro G é diferente: mensagens com até 1400B, entre 1400B e 64kB e acima de 64kB. A transição em 64kB corresponde ao limite que diferencia mensagens curtas e longas, e que são transmitidas através de protocolos diferentes. O intervalo com até 1400B corresponde a mensagens que são transmitidas em um único pacote Ethernet [1]. Os autores utilizaram um parâmetro para representar o *gap* (G) em cada intervalo identificado na configuração avaliada. O *overhead* de envio também é diferenciado para mensagens com até 1400B e para mensagens maiores que 1400B.

No modelo LogGP, assim como em [6], características do sistema de comunicação motivaram a inclusão de novos parâmetros. A inclusão destes parâmetros está relacionada com detalhes do sistema de comunicação, tais como o protocolo usado por bibliotecas de alto nível ou suporte a transferência de mensagens grandes. Adicionar novos parâmetros para cobrir estas situações torna o modelo mais complexo, contrariando o objetivo inicial da modelagem.

Como descrito em [7] e [10] os parâmetros podem ser expressos em função do tamanho da mensagem para contemplar mensagens longas. O modelo LogP Parametrizado [19] [20] estende o modelo representando os parâmetros *overhead* e *gap* em função do tamanho da mensagem. O *overhead* de envio e de recepção são representados por parâmetros distintos, ambos dependentes do tamanho da mensagem.

A latência, conforme definida pelo LogP, corresponde ao tempo despendido na rede com o transporte da mensagem. O tempo despendido no envio de uma mensagem pode ser modelado como a soma do *overhead* de envio, da latência e do *overhead* de recepção. O modelo originalmente supõe que os componentes do tempo de transmissão sejam serializados – e não admite sobreposição.

Em [4] os autores argumentam que a premissa de que os componentes envolvidos na transmissão da mensagem sejam serializados pode não ser verdadeira para algumas configurações. Os autores reportam resultados em que a soma dos componentes excede

[1] O tamanho padrão para o pacote Ethernet é de 1500B. Mensagens do MPI que não ultrapassam os 1500B após somados os bytes de controle, como cabeçalhos TCP/IP e envelope do MPI, podem ser transmitidos em um único pacote.

o tempo total da transmissão, efeito justificado pela sobreposição parcial do *overhead* de envio e do *overhead* de recepção. Os autores propõem o uso da latência definida como o tempo despendido na transmissão da mensagem de um processo a outro, denotada como latência fim-a-fim (ou *End-to-End Latency – EEL*).

O modelo adotado neste trabalho usa a abordagem do LogP parametrizado de representar os parâmetros em função do tamanho da mensagem. Desta forma, o modelo mantém a generalidade e abstração das complexidades inerentes à configuração. Também é adotada a latência fim-a-fim como parâmetro do modelo, que representa a latência exposta para a aplicação, denotada por EEL. Os *overheads* de envio e de recepção também são representados por parâmetros distintos e em função do tamanho da mensagem. Os parâmetros do LogP como empregados neste trabalho estão listados a seguir.

$EEL(m)$ Latência fim-a-fim que corresponde ao custo da transmissão de uma mensagem de tamanho m de um processo a outro.

$Os(m)$ Carga na *CPU* imposta pela comunicação durante o envio de uma mensagem de tamanho m.

$Or(m)$ Carga na *CPU* imposta pela comunicação durante a recepção de uma mensagem de tamanho m.

$g(m)$ Intervalo de tempo mínimo entre mensagens consecutivas de tamanho m. A recíproca corresponde à largura de banda para mensagens de tamanho m.

L Latência de transporte de uma mensagem pequena.

P Número de processos.

Por motivos de simplicidade, o tamanho da mensagem pode ser omitido da modelagem, mas quando é usado o modelo estendido fica implícito que o parâmetro do modelo deve ser medido em função do tamanho da mensagem ($ELL(m)$ é equivalente a ELL). A latência de transporte L pode ser útil para modelar determinadas operações, mas seu uso é restrito a cenários em que as mensagens são pequenas.

3.2 Largura de banda

A largura de banda é um parâmetro importante para aplicações que transmitem grandes quantidades de dados. Esta medida relaciona a quantidade de dados que podem ser transmitidos por unidade de tempo. Do ponto de vista da aplicação é importante considerar a largura de banda exposta pelo sistema de comunicação como um todo, e não apenas a capacidade nominal da rede de interconexão. Por exemplo, nos resultados descritos na Seção 4, observa-se que a largura de banda exposta pelo MPI fica em torno de 350Mbps em uma rede com capacidade nominal de 1Gbps.

Mesmo para um parâmetro relativamente simples como a largura de banda, ou vazão, podem ser usadas diferentes abordagens para sua medição. É preciso diferenciar a largura de banda atingida no envio de uma única mensagem, da largura de banda sustentada com o envio consecutivo de várias mensagens.

Para medir a largura de banda em uma única mensagem, pode-se usar um teste do tipo ping-pong com um determinado tamanho de mensagem. A metade do tempo do ping-pong corresponde ao custo do envio de uma mensagem, e o inverso do tempo determina a largura de banda para o tamanho de mensagem testado.

A vazão sustentada com o envio consecutivo de mensagens pode ser medida por um teste no qual um processo apenas envia muitas mensagens para um destinatário que apenas as recebe. No modelo LogGP, a largura de banda sustentada com mensagens consecutivas é a recíproca do parâmetro $g(m)$, onde m é o tamanho da mensagem.

Para medir corretamente a largura de banda, o teste deve considerar o tempo despendido desde o início das operações de envio, até que a última mensagem seja recebida pelo destinatário. Entretanto, no padrão MPI o envio de uma mensagem pode completar localmente antes que ela seja recebida pelo destinatário. Para garantir que todas as mensagens foram recebidas, pode-se adicionar uma mensagem de controle retornada pelo destinatário após receber a última mensagem do fluxo, tal como descrito em [4]. O tempo consumido com a mensagem de controle é amortizado pelo grande número de mensagens enviadas consecutivamente.

A ferramenta NetPIPE [26] mede a largura de banda através de testes de ping-pong

e através de testes com mensagens consecutivas em uma única direção. No NetPIPE os testes denominados unidirecional e bidirecional são baseados no padrão ping-pong, e o teste com mensagens consecutivas é denominado *stream*. O NetPIPE mede o tempo do teste no processo que recebe as mensagens para garantir que a medida contemple o recebimento da todas as mensagens.

Em [4] os autores descrevem um teste de largura de banda que emprega filas de mensagens pendentes, ou em andamento, baseado na hipótese de que o subsistema de comunicação pode tratar de forma mais eficiente mensagens agrupadas. O teste solicita o envio de Q mensagens através de primitivas não-bloqueantes, onde Q corresponde ao tamanho da fila. Quando metade das solicitações de envio forem completadas, é solicitado o envio de mais $Q/2$ mensagens, e assim sucessivamente. O teste com uma fila de tamanho 1 corresponde ao teste de largura de banda sustentada descrito anteriormente.

Como parte deste trabalho, foi implementado um novo teste baseado na estrutura do *Perftest* [2], para medir a largura de banda sustentada em mensagens consecutivas empregando o mecanismo de filas, tal com descrito na Seção 4.2.3.

3.3 Latência

A latência fim-a-fim, conforme empregada neste trabalho, corresponde ao tempo consumido na transmissão de uma mensagem entre dois processos. A latência pode ser medida através de um teste de ping-pong com mensagens de tamanho m, em que a metade da duração do ping-pong corresponde à latência fim-a-fim.

Para mensagens suficientemente grandes, o *overhead* de envio e o de recepção podem ser sobrepostos, fazendo com que a latência fim-a-fim seja menor que a soma dos *overheads* de envio e recepção.

O inverso da latência fim-a-fim corresponde à largura de banda obtida com a ferramenta NetPIPE para o teste unidirecional, que por sua vez é baseado em testes de ping-pong. O teste de latência fim-a-fim é implementado pela ferramenta *Perftest* com

[2]Manual de uso da ferramenta e link para download podem ser encontrados em `http://www-unix.mcs.anl.gov/mpi/mpptest/`

o nome de *roundtrip*, e seus resultados são mostrados na Seção 4.2.3.

3.4 Padrões de Comunicação

Alguns padrões de comunicação são recorrentes em aplicações paralelas, e podem ser implementados com diferentes abordagens. Em alguns casos podem ser implementados em uma única primitiva das bibliotecas de alto nível. Por exemplo, uma operação de difusão *(broadcast)*, na qual um processo envia uma mensagem para todos os outros, é implementada por uma única primitiva do padrão *MPI* (MPI_Bcast), mas também pode ser implementada pela aplicação com base em diferentes algoritmos.

O desempenho de operações com um determinado padrão pode ser inferido com base nos parâmetros do modelo LogP, mas também pode ser medido diretamente no sistema comparando diferentes abordagens de implementação. Neste trabalho foi desenvolvido um teste com o padrão de comunicação todos-para-todos, no qual cada processo troca mensagens com todos os outros do grupo de comunicação. O teste foi implementado com base na estrutura do *Perftest*, e os resultados são apresentados na Seção 4.2.3.

3.5 Sobrecarga de processamento

A sobrecarga, ou *overhead*, corresponde à carga adicional de trabalho da *CPU* envolvida com a comunicação, e é representada no modelo como Os e Or para envio e recebimento respectivamente. Em um sistema de comunicação organizado em camadas, cada uma delas contribui com um nível de abstração mais elevado, mas acrescenta alguma sobrecarga de processamento. Para o projeto de aplicações paralelas interessa o *overhead* exposto pela biblioteca de alto nível, mas pode-se usar o *overhead* como medida de comparação entre duas configurações, em particular para comparação de diferentes pilhas de protocolos.

A sobrecarga também depende do método empregado no envio da mensagem, e portanto não apresenta necessariamente um comportamento linear com o aumento do tamanho da mensagem.

Um método comumente empregado para medir a sobrecarga em operações de envio

e recebimento é tomar o tempo consumido na execução da primitiva de comunicação. Esta estratégia é implementada pelo LogP Benchmark descrito em [20]. A medida da sobrecarga de recebimento exige um controle para garantir que a primitiva de recebimento seja iniciada quando a mensagem já estiver disponível, para que a medida não inclua o tempo decorrido na rede antes do envolvimento da CPU.

Novamente, é importante considerar aspectos do sistema de comunicação, no caso do padrão *MPI*, pois a semântica da primitiva usada no teste pode influenciar no resultado obtido. No modo de comunicação padrão, a primitiva de envio pode copiar a mensagem para um *buffer* intermediário, e o fluxo retorna para o programa mesmo antes da mensagem ter sido efetivamente transmitida. O sistema ainda usa tempo de processamento para efetivamente transmitir a mensagem, que pode implicar em carga de trabalho para a CPU. No caso da sobrecarga de recebimento, antes do início da primitiva, o sistema é envolvido na cópia da mensagem do adaptador de interface para um *buffer* intermediário, que pode implicar em carga de trabalho da CPU, e que não faz parte da medida. A sobrecarga que incide sobre a *CPU* na cópia entre o *buffer* intermediário e o dispositivo de rede depende da configuração, mas espera-se que seja relativamente pequena devido ao uso de mecanismos como DMA.

No modo de comunicação síncrono, a primitiva retorna quando o processo de destino inicia a recepção da mensagem, o que envolve uma mensagem de aceitação enviada pelo destinatário. Neste caso, o tempo decorrido na execução da primitiva pode incluir intervalos em que a *CPU* não é utilizada, e este tempo ocioso da *CPU* não deve fazer parte da medida. Assim, a medida é afetada pela latência na comunicação durante a negociação entre o processo de origem e de destino, e portanto pode representar um erro significativo. Portanto, o tempo despendido para completar uma operação de envio não pode ser interpretado como carga adicional de trabalho para a *CPU* – são medidas distintas.

Um teste para medir a sobrecarga em operações de envio e recepção é descrito em [4], tendo por base o teste de largura de banda com mensagens consecutivas. O teste proposto consiste em inserir gradativamente computação entre a primitiva não-bloqueante

Teste de Sobrecarga

`for (int i = 0; i < REPETICOES; i++){`
`  MPI_Isend();`
`  OverlapComputation(len);`
`  MPI_Wait();`
`}`

Tabela 3.1: Teste de sobrecarga com primitivas não-bloqueantes.

(de iniciação) e a primitiva associada (de conclusão) para completar a operação, como descrito na Tabela 3.1. Quando não há computação inserida entre as primitivas de comunicação a medida corresponde ao *gap*. A mesma estrutura pode ser usada para medir a sobrecarga de recebimento.

Se o *gap* não é dominado pelo *overhead*, há um intervalo de tempo entre as chamadas consecutivas da primitiva de comunicação em que a *CPU* fica ociosa, e portanto há uma oportunidade para realizar trabalho útil. Ao inserir quantidades suficientemente pequenas de computação, o tempo do teste não se altera. Quando a quantidade de computação excede a oportunidade de sobreposição, o tempo do teste cresce com o aumento da quantidade de computação. A carga de trabalho da *CPU* envolvida na comunicação corresponde à diferença entre o *gap* e a quantidade de computação escondida – que não altera o tempo total do teste.

Esta estratégia pode ser usada tanto para medir a sobrecarga de envio quanto a sobrecarga de recebimento. O teste expõe tanto a sobrecarga quanto a oportunidade de sobreposição do *gap* na transmissão de mensagens consecutivas, como tratado na Seção 3.6. Novos testes baseados nesta abordagem foram implementados com base na ferramenta *Perftest*, e estão descritos na Seção 4.2.3.

3.6 Sobreposição de Comunicação com Computação

A sobreposição de comunicação com computação útil é uma técnica conhecida para esconder a latência em aplicações paralelas, especialmente em sistemas que apresentam custo elevado de comunicação entre as unidades de processamento. Apesar de ser amplamente reconhecida, nem sempre há um conhecimento preciso de como explorar esta técnica ou mesmo de como medir a oportunidade de sobreposição exposta pelos sistemas

de comunicação.

O objetivo da sobreposição da comunicação com a computação é aproveitar tempo ocioso da *CPU* para realizar computação útil, e assim esconder parte do custo de comunicação [13]. Neste trabalho, foram identificadas três situações, ou padrões de comunicação, em que pode haver oportunidade de sobreposição, que foram classificadas como: ping-pong, local e *gap*.

3.6.1 Ping-pong

Processo A	Processo B

`MPI_Send();`
`OverlapComputation(len);`
`MPI_Recv();`
	`MPI_Recv();`
	`MPI_Send();`

Tabela 3.2: Teste de sobreposição em operação de ping-pong entre dois processos A e B. A medida é realizada no processo A.

Em um acesso à memória de uma unidade remota, a *CPU* é envolvida inicialmente na transmissão de uma mensagem, mas fica ociosa até receber a mensagem de resposta. O tempo ocioso pode ser usado para realizar computação útil que não depende do resultado da comunicação em andamento. Uma leitura à memória remota pode ser modelada por um teste de ping-pong.

Em termos dos parâmetros do modelo, o tempo total da leitura corresponde ao dobro da latência fim-a-fim ($2 * EEL$), mais o tempo consumido no processo remoto para obter o valor requisitado – que é omitido da modelagem.

Considerando inicialmente um cenário em que as mensagens são pequenas e a latência do transporte (L) é conhecida, pode-se modelar o teste de ping-pong por $Os + L + Or + Os + L + Or$. No processo de origem, a *CPU* é envolvida no envio da mensagem (Os) e depois na recepção da resposta (Or), e fica ociosa durante o restante da comunicação ($L + Or + Os + L$), que neste cenário representa a oportunidade de sobreposição.

Para abranger mensagens grandes e a possibilidade de sobreposição dos parâmetros, o modelo estendido emprega a latência fim-a-fim (EEL) ao invés da latência do transporte da mensagem (L), além de empregar parâmetros em função do tamanho da mensagem.

O teste pode ser modelado como $(EEL + EEL)$, e a oportunidade de sobreposição corresponde ao tempo de ida e volta $(2 * EEL)$ menos o tempo de *CPU* despendido localmente com o envio e recepção da mensagem $(Os + Or)$.

No caso da sobreposição com padrão ping-pong, a oportunidade de sobreposição pode ser estimada com base no modelo, mas também pode ser medida diretamente no sistema. Para medir a sobreposição, o teste implementa um algoritmo de ping-pong, e gradativamente insere computação entre o envio da mensagem e a recepção da resposta, como descrito na Tabela 3.2. Desta forma, enquanto a computação estiver escondida, ou sobreposta com a comunicação, o tempo total do teste não se altera significativamente. Quando a quantidade de computação é maior que a oportunidade de sobreposição, o tempo do teste é afetado pela computação inserida. No cenário descrito para mensagens pequenas, o teste de sobreposição pode ser modelado por $Os + max((L + Or + Os + L), C) + Or$, onde C representa a quantidade de computação inserida na execução do teste. Com base no modelo estendido, o tempo do teste pode ser representado como $max(C, ((2 * EEL) - (Os + Or))$, em que os parâmetros são medidos em função do tamanho da mensagem.

A ferramenta *Perftest* implementa um teste de sobreposição baseado no padrão de comunicação ping-pong, descrito na Seção 4.2.3.

3.6.2 Local

A *sobreposição local* consiste na oportunidade de sobreposição entre uma primitiva de comunicação não-bloqueante e o término da operação, para mensagens não consecutivas. Neste caso, a oportunidade de sobreposição local não está associada ao *gap* em mensagens consecutivas.

A oportunidade de sobreposição nesta situação depende do protocolo empregado pelo MPI na transmissão da mensagem. No modo de comunicação síncrono, há uma negociação entre as unidades envolvidas na comunicação, o que torna o desempenho desta primitiva sensível à latência da rede. Durante a negociação, pode haver intervalos em que a *CPU* da unidade de origem fica ociosa esperando pela aceitação do processo de destino. Da mesma

forma, a unidade de destino pode ficar ociosa entre o envio da aceitação e a chegada da mensagem.

No modo de comunicação padrão, se a mensagem for copiada para um *buffer* intermediário, é provável que a *CPU* esteja engajada na cópia dos dados e haja pouca ou nenhuma oportunidade de sobreposição – isto entretanto depende da implementação. Note que para mensagens longas o modo padrão de comunicação no MPI pode empregar um protocolo síncrono levando à situação descrita anteriormente.

O tempo despendido para completar o envio de uma mensagem engloba o parâmetro Os, mas também envolve custos com o protocolo de transporte, que quando envolve mensagens de aceitação é influenciado pela latência da rede. Para modelar a oportunidade de sobreposição local, seria necessário considerar características do protocolo de transporte, tornando a análise complexa e dependente da configuração. Esta medida pode ser obtida diretamente no sistema através de testes, ou *microbenchmarks*.

No teste de sobreposição local um processo envia mensagens através de uma primitiva de envio não-bloqueante seguida de uma primitiva associada para completar a operação, e gradativamente insere-se computação entre a primitiva de envio e a primitiva associada para completar a operação. Uma vez que a oportunidade de sobreposição é preenchida, o teste é afetado pelo acréscimo de computação.

Para que possa haver sobreposição local, é preciso que a comunicação progrida após a solicitação de envio não-bloqueante, em paralelo com a execução da aplicação. Entretanto, em algumas configurações a comunicação não progride até que a primitiva para completar a operação seja executada. Deste modo, a computação inserida não fica sobreposta com a comunicação, mesmo havendo tempo de *CPU* ocioso durante o processamento do protocolo de transporte.

Se houver tempo ocioso de CPU, mas a comunicação não progredir, o tempo do teste cresce ao se inserir computação indicando que não há oportunidade de sobreposição. Logo, o teste mede a oportunidade de sobreposição exposta pelo sistema de comunicação, e não necessariamente o tempo ocioso da *CPU*. O novo teste de sobreposição local foi implementado com base na estrutura do *Perftest*, e está descrito na Seção 4.2.3.

3.6.3 gap

Quando um processo envia mensagens consecutivas, ele fica sujeito à capacidade de injetar mensagens na rede, que é representada como *gap* (g) no modelo. Quando o parâmetro g não é dominado pelo *overhead* (Os), há oportunidade de sobreposição que pode ser representada por $g - Os$.

Para medir a oportunidade de sobreposição do *gap*, um processo envia mensagens consecutivamente para um destinatário que apenas as recebe. Gradativamente insere-se computação entre as primitivas de envio, até que a oportunidade de sobreposição seja preenchida por computação e acréscimos subsequentes na quantidade de computação aumentem o tempo do teste. Este teste possui duas implementações, uma baseada em primitivas bloqueantes e outra em primitivas não-bloqueantes. O algoritmo baseado em primitiva não-bloqueante é o mesmo usado no teste de sobrecarga descrito na Tabela 3.1. Os novos testes de sobreposição do *gap* foram implementados com base na estrutura do *Perftest* e estão descritos na Seção 4.2.3.

3.7 Multi-threading

Em aplicações paralelas o uso de Threads pode ser usado para implementar operações de comunicação de forma assíncrona, e assim tornar concorrentes as fases de comunicação e computação. Em [13] esta técnica é avaliada comparando-se implementações que empregam threads e implementações que têm as fases de comunicação e computação seqüenciais. Três *kernels científicos* foram usados na avaliação, dos quais dois apresentaram ganho de desempenho da ordem de 20% na implementação baseada em threads. Um dos kernels apresentou desempenho pior que o da implementação original. O autor propõe estender o trabalho avaliando outras implementações de *threads* para verificar o impacto no desempenho.

Diferentes abordagens para integrar comunicação e computação são discutidas em [27]. Os autores avaliam o uso de *threads* de usuário, *threads* do kernel, e processos separados (*deamoms*) para esconder a latência da comunicação. Sua conclusão é de que através do

uso de uma implementação adaptada de *threads* de usuário é possível sobrepor até 80% do custo de comunicação em aglomerados ligados por rede Ethernet de 10Mbps.

O uso de *threads* é comparado com o uso de primitivas de comunicação assíncronas em [12]. Os resultados não apontam vantagem significativa no uso de *threads* para promover sobreposição de comunicação. Segundo os autores, o desempenho das aplicações que empregam *threads* para realizar comunicação pode ser alcançado com primitivas de comunicação assíncronas disponibilizadas pela biblioteca de comunicação. Entretanto, o artigo avalia apenas o desempenho global da aplicação, sem considerar outros critérios como aqueles discutidos neste trabalho.

Uma alternativa considerada em [9] é atribuir mais de uma tarefa ao mesmo processador, de forma que enquanto uma tarefa está bloqueada aguardando comunicação, outra prossegue com computação útil. Pode-se presumir que a quantidade ideal de tarefas depende dos parâmetros da máquina. O número de processos poderia ser parametrizado, e assim a aplicação poderia ser submetida a diferentes configurações com pouco esforço. No entanto, esta técnica depende da existência de paralelismo suficiente na aplicação. Nem sempre é possível particionar o problema para atribuir o número ideal de tarefas a cada estação do aglomerado. Além disso, esta abordagem introduz custos associados as trocas de contextos que dificultam a modelagem. O trabalho aqui descrito enfoca a integração da comunicação em uma única tarefa.

3.8 Trabalhos Relacionados

Este trabalho usa como base o modelo LogP [10] e outras extenções propostas em trabalhos anteriores, tais como LogGP [2] e LogP Parametrizado [19] [20].

Grande parte dos sistemas possuem suporte especial para transmissão de mensagens grandes *(Bulk Transfer)*, de forma que a medida do *gap* depende do mecanismo usado na comunicação. O LogGP introduz o parâmetro G para representar o *gap* na transmissão de mensagens grandes.

As bibliotecas do padrão MPI usam protocolos distintos de acordo com o tamanho da mensagem, o que pode produzir medidas distintas de alguns parâmetros. Em [6] os

autores identificaram intervalos nos tamanhos das mensagens que apresentam medidas diferentes dos parâmetros *gap* e *overhead*, e representam tais diferenças com a inclusão de novos parâmetros.

A inclusão de novos parâmetros torna o modelo complexo e restrito a uma determinada configuração. O modelo adotado neste trabalho segue a abordagem do LogP Parametrizado, em que os parâmetros são expressos em função do tamanho da mensagem.

A ferramenta *LogP Benchmark* [20] permite medir os parâmetros do modelo LogP de forma rápida, sem saturar a rede com mensagens longas. Entretanto, os testes usados para medição dos *overheads* de envio e recepção usam uma abordagem pouco precisa. Neste trabalho foram construídos novos testes para medir os parâmetros do LogP de forma mais realista.

O *NetPIPE* [26] é uma ferramenta independente de protocolo usada para avaliar o desempenho de um sistema de comunicação com base na largura de banda. No NetPIPE o teste da largura de banda consiste no inverso do tempo consumido na transmissão do conteúdo de uma única mensagem. Nos testes desenvolvidos neste trabalho foram criados testes de largura de banda sustentada com mensagens consecutivas, além de testes que empregam filas de mensagens pendentes configuradas através de primitivas não-bloqueantes.

Em [4] são avaliadas tecnologias recentes de redes de alto desempenho que são empregadas em aglomerados de PCs. É descrito um conjunto de *microbenchmarks* usado para avaliar o desempenho com base em um conjunto de parâmetros, tais como sobrecarga de processamento, latência e largura de banda.

O uso de *threads* distintos para promover sobreposição da comunicação com computação foi explorado em [13]. Em [12] os autores comparam o uso de threads com o uso de primitivas não bloqueantes para promover sobreposição da comunicação. Entretanto, este trabalho foi baseado em tecnologias disponíveis no ano de 1995 com redes de 10Mbps. No trabalho aqui apresentado estas técnicas são comparadas.

CAPÍTULO 4

AVALIAÇÃO DE DESEMPENHO

A primeira parte da avaliação de desempenho descrita no que se segue visa a análise de desempenho e caracterização do sistema de comunicação através de testes, ou *microbenchmarks*, executados diretamente no sistema avaliado. Na segunda parte da avaliação são empregados trechos de aplicações científicas, que também oferecem uma avaliação do desempenho do sistema, além de comparar diferentes abordagens de implementação. Estas abordagens foram avaliadas de forma isolada na primeira parte, através de *microbenchmarks*.

Com o objetivo de facilitar a leitura, o texto está organizado de forma que a descrição dos experimentos, testes e kernels científicos, é seguida da apresentação de seus resultados.

4.1 Ambiente Computacional

Esta seção descreve o ambiente computacional usado para os experimentos relatados no que se segue. O ambiente referenciado como *Nautilus* é um aglomerado de PCs mantido pela Universidade Federal do Paraná [1].

O Nautilus conta com três conjuntos de máquinas, identificados como A, G e M, das quais apenas o conjunto A foi empregado na avaliação. Cada grupo possui estações com configuração idênticas. O conjunto A é composto de 10 estações com um processador cada, ligadas por uma rede Ethernet de 1Gbps. O conjunto G conta com 16 estações bi-processadas, ligadas por uma rede Ethernet de 1Gbps. O conjunto de estações identificado como M possui 16 estações com um processador, ligadas por uma rede de 100Mbps. As estações do conjunto M estão ligadas por uma rede Myrinet, mas que opera em modo de emulação da Ethernet. Os conjuntos G e M não são usados na avaliação de desempenho apresentada neste trabalho. A tabela 4.1 lista a configuração das estações

[1] www.ufpr.br

de cada conjunto.

	A	G	M
CPU			
Num Procs	1	2	1
CPU Marca	AuthenticAMD	AuthenticAMD	AuthenticAMD
Modelo	Athlon 64 3500+	Athlon	Athlon XP 1700+
CPU Clock	2GHz	2GHz	1.5GHz
RAM	2GB	2GB	1GB
Adaptador de interface de rede			
Marca	Marvell	Intel Corp.	Myricom Inc.
Modelo	Yukon Gigabit Ethernet	82544EI Gigabit Ethernet	Myrinet 2000
Velocidade	1Gbps	1Gbps	2Gbps [2]

Tabela 4.1: Configuração dos conjuntos de estações do Nautilus.

Em todas as estações do Nautilus, o dispositivo de rede está ligado ao barramento PCI de 32-bits a 66Mhz. Os processadores de 64-bits do conjunto A operam no modo 32-bits.

Os experimentos realizados no conjunto A do Nautilus contaram com máquinas totalmente dedicadas, sem compartilhamento com outros usuários. No conjunto A também foi usado o Kernel com suporte ao protocolo *LLC* descrito na seção 4.1.

Além do Nautilus, o ambiente computacional do C3SL [3] também foi usado. Foram usadas duas estações bi-processadas ligadas por rede Gigabit Ethernet. Cada estação possui dois processadores AMD Opteron de 1.5GHz e 4GB de memória RAM. As estações contam com dispositivos de rede NetXtreme BCM5704 ligados ao barramento PCI de 64-bits. Estas estações foram usadas para demonstrar alguns dos testes por apresentar características diferentes das observadas no Nautilus.

MPI

Neste trabalho foram usadas as bibliotecas *OPENMPI* [4] e MPICH2 [5], que são distribuídas gratuitamente.

Foi usada a versão 1.0.3 da biblioteca MPICH2, com o protocolo TCP. A biblioteca *OPENMPI* pode ser usada com diferentes componentes de transporte de dados, chamados

[3] www.c3sl.ufpr.br
[4] http://openmpi.org/
[5] http://www-unix.mcs.anl.gov/mpi/mpich2/

de BTLs (*byte transfer layer*). Na avaliação foram usados dois BTLs, um que emprega o protocolo TCP e é distribuído com o *OPENMPI*, e outro BTL baseado no protocolo *LLC* descrito adiante.

Também foi avaliado um recurso da biblioteca *OPENMPI* que emprega um thread progresso, usado internamente pela biblioteca para permitir que a comunicação iniciada por uma primitiva não-bloqueante progrida mesmo após o fluxo de execução retornar para o programa. Este recurso é habilitado em tempo de compilação da biblioteca através da opção *–progress-thread*.

Durante os testes desenvolvidos neste trabalho foi identificado um problema com a biblioteca OPENMPI. Quando muitas operações de envio são iniciadas consecutivamente, a biblioteca apresenta um erro. Este problema impediu alguns testes baseados em mensagens consecutivas, tais como o teste de sobrecarga de processamento e largura de banda sustentada. O erro foi reportado aos mantenedores da biblioteca [6].

LLC

Em aglomerados de PCs ligados por rede Ethernet normalmente é usado o protocolo *TCP/IP* para comunicação entre as estações. Em um ambiente de rede local, alguns serviços oferecidos pelo protocolo *TCP/IP* não são necessários, tais como roteamento e endereçamento entre redes IP. Uma versão modificada do *kernel* do Linux permite o uso de uma interface de soquete com o protocolo de enlace da Ethernet [11] – padrão IEEE 802.2 (*Logic Link Control - LLC*) [17].

Em [29], o autor propõe substituir o protocolo *TCP/IP* pelo protocolo de enlace da Ethernet em bibliotecas de comunicação de alto nível, com o objetivo de reduzir a carga de processamento introduzida pelo TCP/IP. O autor desenvolveu um componente de transporte para a biblioteca *OPENMPI* que usa o protocolo *LLC* como camada de transporte. A versão do *OPENMPI* com o componente de transporte (BTL) baseado no protocolo *LLC* foi disponibilizada para a avaliação de desempenho desenvolvida neste trabalho.

[6] https://svn.open-mpi.org/trac/ompi/ticket/232

4.2 Microbenchmarks

4.2.1 Netpipe

Esta Seção descreve o *NetPIPE*, uma das ferramentas usadas para avaliação de desempenho do sistema de comunicação. Os testes implementados pelo *NetPIPE* estão brevemente descritos, e em seguida são discutidos alguns aspectos relacionados ao uso do *NetPIPE* com o padrão MPI. Os resultados obtidos com o *NetPIPE* no Nautilus são apresentados na seqüência, comparando-se o desempenho das bibliotecas *OPENMPI* e *MPICH2*, além de comparar o desempenho do protocolo *LLC* como camada de transporte para o *OPENMPI*.

Descrição

O *NetPIPE* [26] [7], ou *Network Protocol Independent Performance Evaluator*, é uma ferramenta desenvolvida para avaliação de desempenho de um sistema de comunicação, em particular a largura de banda. A ferramenta é independente de protocolo, e tem sido usada na avaliação e comparação de implementações MPI e redes de interconexão [21] [28]. Pode-se medir o desempenho de sistemas de comunicação de alto nível, tais como Message Passing Interface (MPI) e Parallel Virtual Machine (PVM), e também das camadas de comunicação inferiores tais como *TCP* e GM [8].

A ferramenta implementa testes de comunicação entre dois processos para medir a largura de banda do sistema. Os testes são repetidos com tamanhos de mensagem incrementados gradativamente, de forma que o resultado é obtido em função do tamanho de mensagem usado no teste.

Através de parâmetros, pode-se controlar a execução do programa e selecionar um dos testes implementados pela ferramenta: unidirecional, bidirecional ou *stream*. A versão *unidirecional* implementa um teste de ping-pong, em que as mensagens são enviadas alternadamente pelos processos. Apesar de haver troca de mensagens nos dois sentidos,

[7] `http://www.scl.ameslab.gov/netpipe/`
[8] Protocolo usado em redes Myrinet

os processos não enviam mensagens simultaneamente, e o desempenho reportado pela ferramenta corresponde à capacidade de transmissão em uma única direção. No teste *bidirecional* dois processos enviam mensagens simultaneamente. Este teste captura a capacidade do sistema de comunicação quando há transmissão nas duas direções ao mesmo tempo. A ferramenta também implementa um teste de *stream*, no qual apenas um processo envia mensagens, e o outro apenas recebe. Neste teste, as mensagens são enviadas consecutivamente ao processo de destino, sem esperar por uma resposta a cada iteração.

Uso do *NetPIPE* com MPI

O *NetPIPE* possui uma implementação baseada no padrão MPI com os testes de largura de banda descritos anteriormente.

Nos testes do tipo ping-pong, o envio das mensagens é intercalado com uma operação de recebimento, de forma que as operações de envio não são executadas consecutivamente. Desta forma, o teste baseado no padrão ping-pong não é influenciado pela capacidade de injetar mensagens na rede, representada pelo parâmetro *gap* do modelo LogP.

O teste de ping-pong mede a largura de banda atingida na transmissão do conteúdo de cada mensagem, quando esta não pertence a um conjunto de operações de envio consecutivas. A ferramenta considera o tempo consumido no envio da mensagem como metade da duração do teste de ping-pong. O teste de *stream* envia as mensagens consecutivamente para o processo de destino. Portanto, a medida é sensível à capacidade de injetar mensagens na rede exposta pelo MPI, e não apenas à vazão atingida na transmissão de uma mensagem. O teste de *stream* captura a largura de banda que pode ser sustentada pelo envio de mensagens consecutivas de um determinado tamanho.

Uma possível falha da biblioteca *OPENMPI* faz com que o teste do tipo *stream* não funcione corretamente. A falha está relacionada ao envio de mensagens consecutivas, que em grande número causam um erro em tempo de execução. Por este motivo os resultados do teste de *stream* estão restritos ao uso da biblioteca *MPICH2*.

Resultados

Esta Seção apresenta os resultados obtidos com o *NetPIPE* no conjunto A de máquinas do Nautilus. Os resultados comparam o desempenho dos protocolos *TCP* e *LLC* como camadas de transporte da biblioteca *OPENMPI*, e comparam o desempenho entre as bibliotecas *MPICH2* e *OPENMPI*.

O desempenho obtido com o conjunto A de máquinas do Nautilus apresenta uma queda acentuada na vazão para alguns tamanhos de mensagem, como mostra o gráfico da Figura 4.1. O gráfico representa a largura de banda, no eixo Y, em função do tamanho das mensagens que estão representados em escala logarítimica no eixo X. Os gráficos mostram os resultados obtidos com a biblioteca *OPENMPI* para os protocolos *TCP* e LLC, combinados com os modos de comunicação padrão e síncrono do MPI.

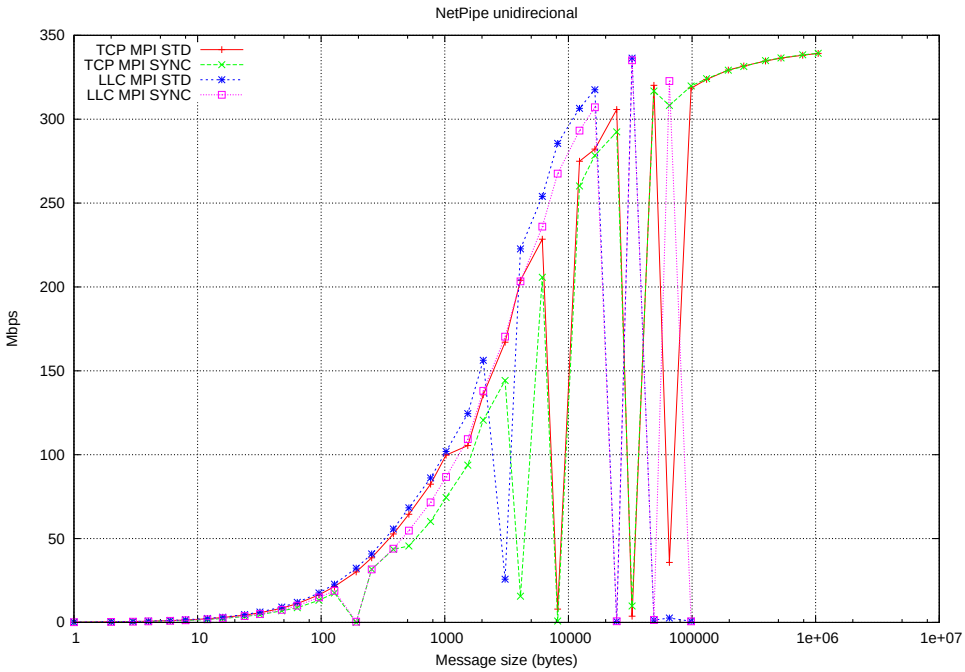


Figura 4.1: Resultados do *NetPIPE* para mensagens variando de 1 byte a 1MB – eixo X em escala logarítimica.

Para alguns tamanhos de mensagem ocorrem quedas acentuadas na largura de banda, destacando-se da projeção esperada do gráfico. Estas anomalias são observadas apenas no conjunto A de máquinas do Nautilus, e estão relacionadas ao uso do dispostivo Marvel Yukon com o *driver* sk98lin para Linux [9]. O driver sk98lin foi substituído nos Kernels mais

[9]Como já foi reportado anteriormente em outro estudo disponível em `http://www.digit-life.com/`

recentes, a partir da versão 2.6.16, pelo driver sky2. Entretanto, o ambiente disponível para a avaliação usa a versão 2.6.14 do Kernel do Linux com o driver sk98lin.

	OPENMPI (LLC)	*OPENMPI* (TCP)	*MPICH2*
16kB	317.55	282.16	266.14
32kB	336.39	–	–
1MB	–	339.26	315.66

Tabela 4.2: Largura de banda obtida com o *NetPIPE* para mensagens de 16kB, 32kB e 1MB com as bibliotecas OPENMPI-LLC, OPENMPI-TCP e *MPICH2*. Os resultados anômalos foram omitidos.

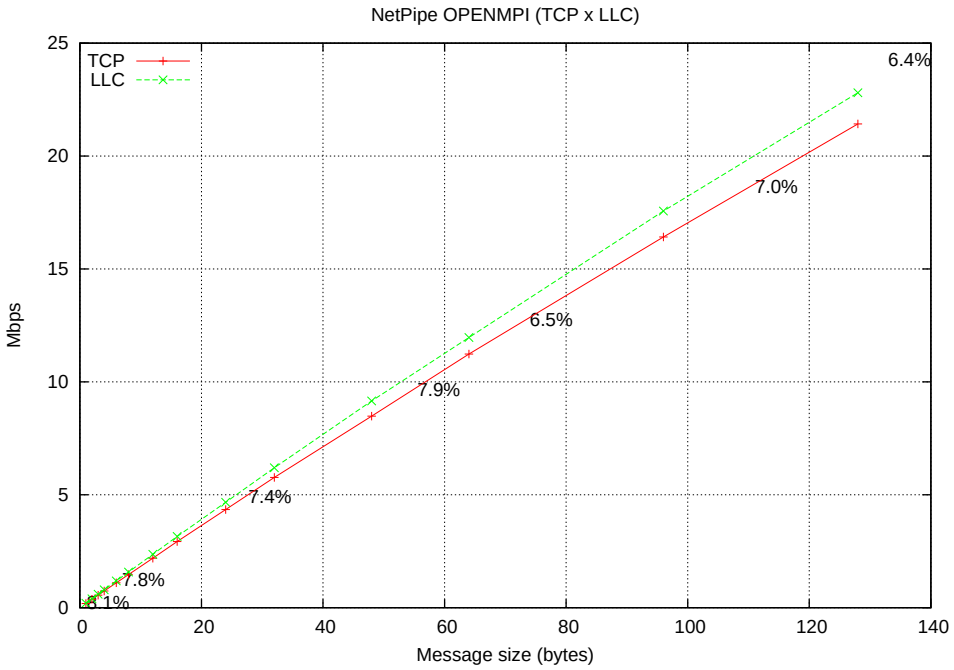


Figura 4.2: Comparação do desempenho do protolo *LLC* em relação ao *TCP* para mensagens com até 128 bytes.

A execução do *NetPIPE* com a biblioteca *OPENMPI* com protocolo *TCP* reporta uma vazão de 339.26 Mbps em mensagens de 1MB. O *OPENMPI* com protocolo *LLC* apresentou vazão de 336.39 Mbps para mensagens de 32kB. Porém, a medida com melhor desempenho do *LLC* ocorre para mensagens de 32kB, e apesar de apresentarem uma tendência crescente com o aumento do tamanho da mensagem, os resultados para mensagens acima de 32kB são erráticos.

A Tabela 4.2 apresenta os resultados para mensagens de 1MB, 32kB e 16kB. Para mensagens de 1MB observa-se o melhor desempenho obtido para a biblioteca *OPENMPI*

`articles2/gig-eth-64bit/gig-eth-64bit-apr2004-p1-3.html`

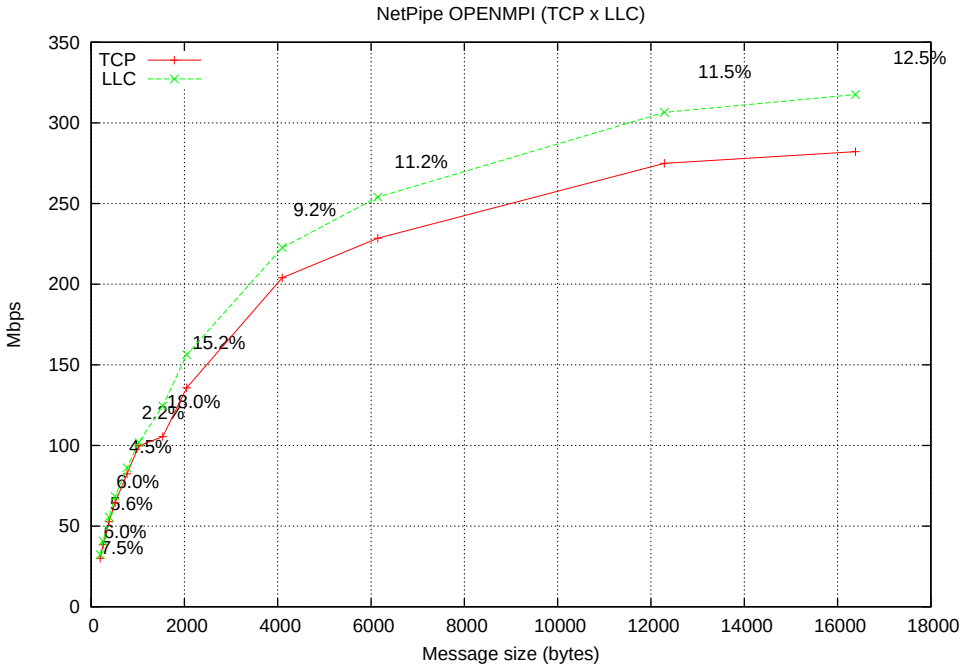


Figura 4.3: Comparação do desempenho do protolo *LLC* em relação ao *TCP* para mensagens entre 128 bytes e 16kB.

com o protocolo TCP. Dos resultados válidos obtidos com o protocolo LLC, a medida com melhor desempenho é para mensagens de 32kB. O maior tamanho de mensagem em que a medida é coerente – não representa uma anomalia – para os dois protocolos é 16kB. Para mensagens de 16kB o *OPENMPI* com protocolo *TCP* atinge 282.16 Mbps e com o protocolo *LLC* atinge 317.55 Mbps, que representa um ganho de 12% do *LLC* em relação ao TCP.

As Figuras 4.2 e 4.3 mostram os resultados do *NetPIPE* apenas para tamanhos de mensagem que não apresentam resultados erráticos. O gráfico da Figura 4.2 mostra os resultados para mensagens com até 128 bytes, e o gráfico da Figura 4.3 para mensagens entre 128 bytes e 16kB.

Estes resultados mostram que o protocolo *LLC* tem desempenho superior ao *TCP* para mensagens longas, porém as anomalias presentes na configuração testada dificultam uma comparação precisa para mensagens maiores que de 32kB.

Foram comparados os desempenhos da biblioteca *MPICH2* e *OPENMPI* com protocolo TCP. O gráfico da Figura 4.4 mostra que para mensagens com até 512B o *MPICH2* apresenta desempenho superior. Para mensagens maiores que 512B a biblioteca

OPENMPI apresenta melhor desempenho, como mostra a Figura 4.5.

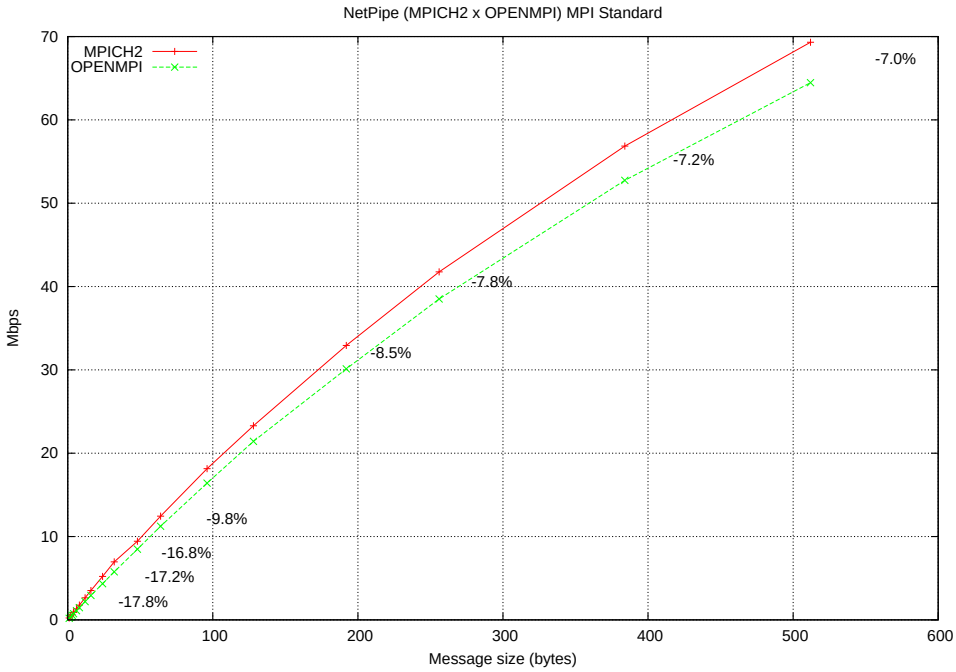


Figura 4.4: Comparação do desempenho da biblioteca *MPICH2* em relação a biblioteca *OPENMPI* para mensagens com até 512kB.

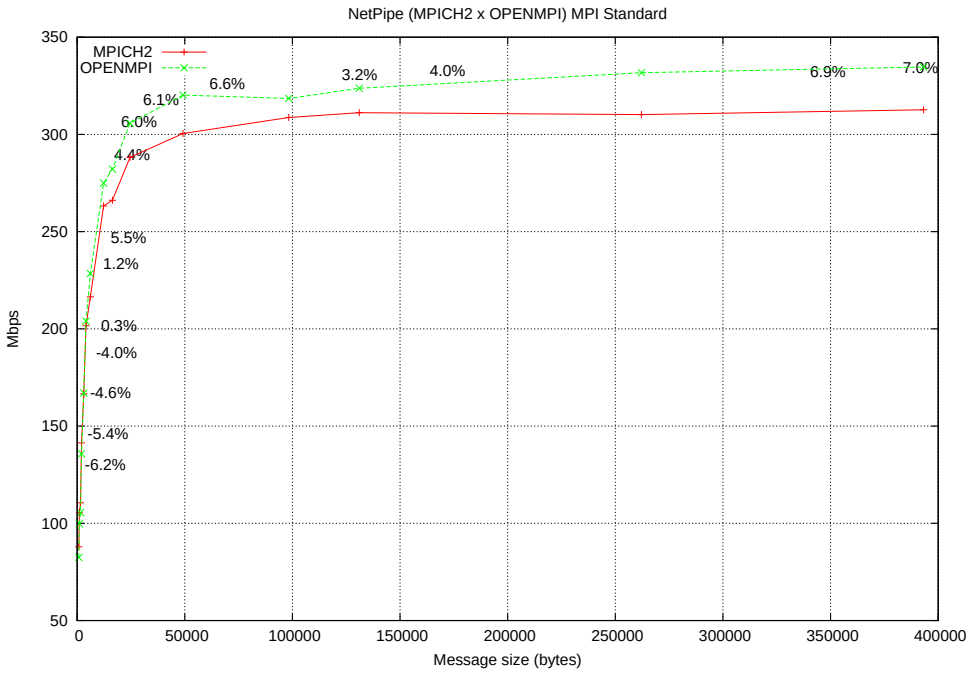


Figura 4.5: Comparação do desempenho da biblioteca MPICH2 em relação a biblioteca OPENMPI para mensagens entre 512 bytes e 256kB.

O gráfico da Figura 4.6 apresenta os resultados do teste do tipo *stream* com modos de comunicação padrão e síncrono, além de apresentar o resultado do teste unidirecional

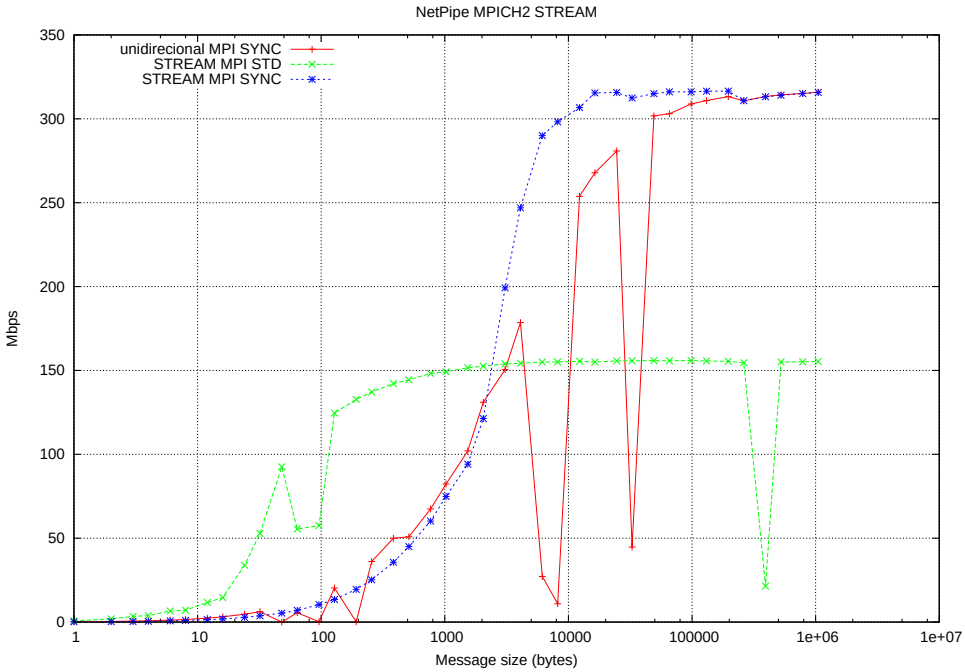


Figura 4.6: Resultado do *NetPIPE* do teste do tipo *stream* – eixo X em escala logarítimica. O eixo Y representa a largura de banda em Mbps em função do tamanho da mensagem.

síncrono para comparação.

Para mensagens de até 2kB o desempenho do teste de *stream* com modo de comunicação padrão é superior ao teste unidirecional. Para mensagens maiores que 2kB os resultados estabilizam em torno de 155Mbps.

O teste de *stream* com modo de comunicação síncrono apresenta desempenho superior na faixa de 2kB até 256kB. Para mensagens maiores que 256kB o resultado fica próximo ao obtido no teste unidirecional.

O modo de comunicação síncrono apresenta resultados inferiores aos obtidos com o modo de comunicação padrão para mensagens curtas, tanto para o *OPENMPI* quanto para o *MPICH2*. Conforme descrito na Seção 2.1 as implementações usam protocolos distintos para mensagens curtas e longas. O limite padrão para a biblioteca *OPENMPI* é de 64kB, e para a implementação *MPICH2* de 16kB. Para mensagens maiores que o limite, espera-se que o desempenho dos dois modos de comunicação sejam iguais, como pode ser verificado no gráfico da Figura 4.7 para o *OPENMPI* e no gráfico da Figura 4.8 para o *MPICH2*.

Na comparação entre o modo síncrono e padrão, nota-se que há um pequeno intervalo

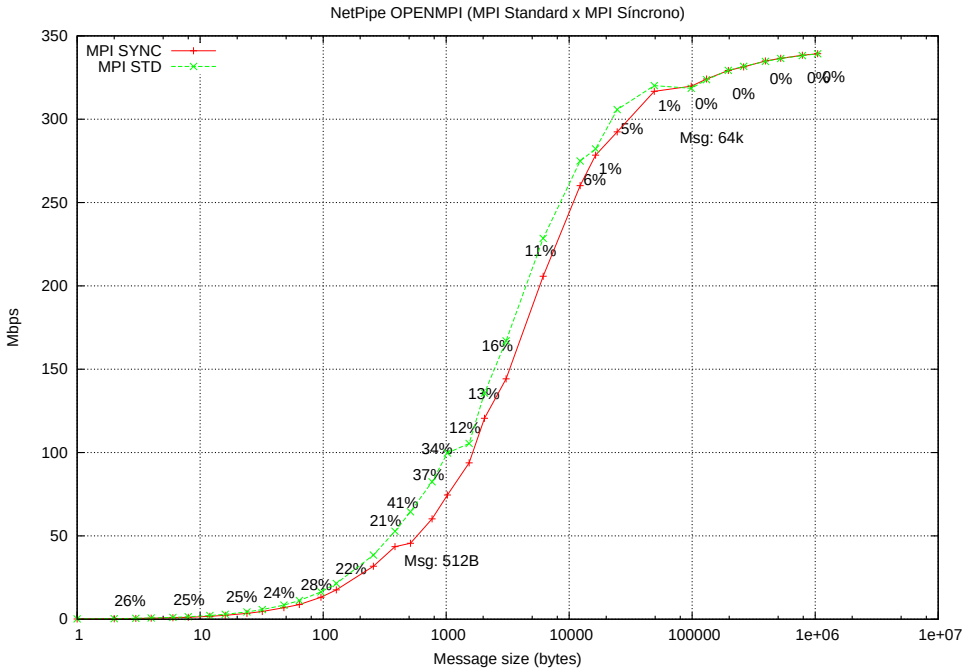


Figura 4.7: Comparação do desempenho dos modos de comunicação síncrono e padrão da biblioteca *OPENMPI*.

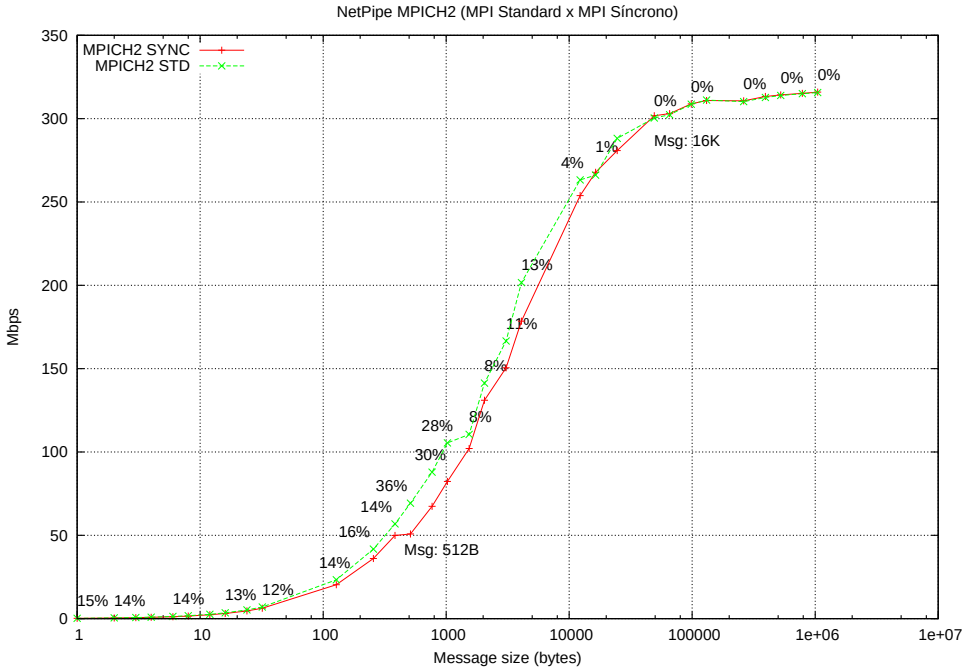


Figura 4.8: Comparação do desempenho dos modos de comunicação síncrono e padrão da biblioteca *MPICH2*.

em que o desempenho do modo síncrono afasta-se do desempenho do modo padrão. Ao passar de 384 bytes para 512 bytes o desempenho fica muito próximo, tanto para o *OPENMPI* (4.7) quanto para o *MPICH2* (4.8).

Os gráficos das Figuras 4.9 e 4.10 mostram o desempenho da biblioteca *OPENMPI*

com suporte a um *Thread* de progresso. Os resultados foram obtidos com a versão 1.0 do *OPENMPI* para os testes sem o *Thread* de progresso, e com a versão 1.1.1 do *OPENMPI* com o *thread* de progresso.

O objetivo desta comparação é verificar qual a influência do *thread* de progresso no desempenho do *OPENMPI*. Como foram usadas versões diferentes do *OPENMPI*, os resultados estão sujeitos a eventuais modificações entre a versão 1.0 e a versão 1.1.1, além da adição do recurso (*thread* de progresso).

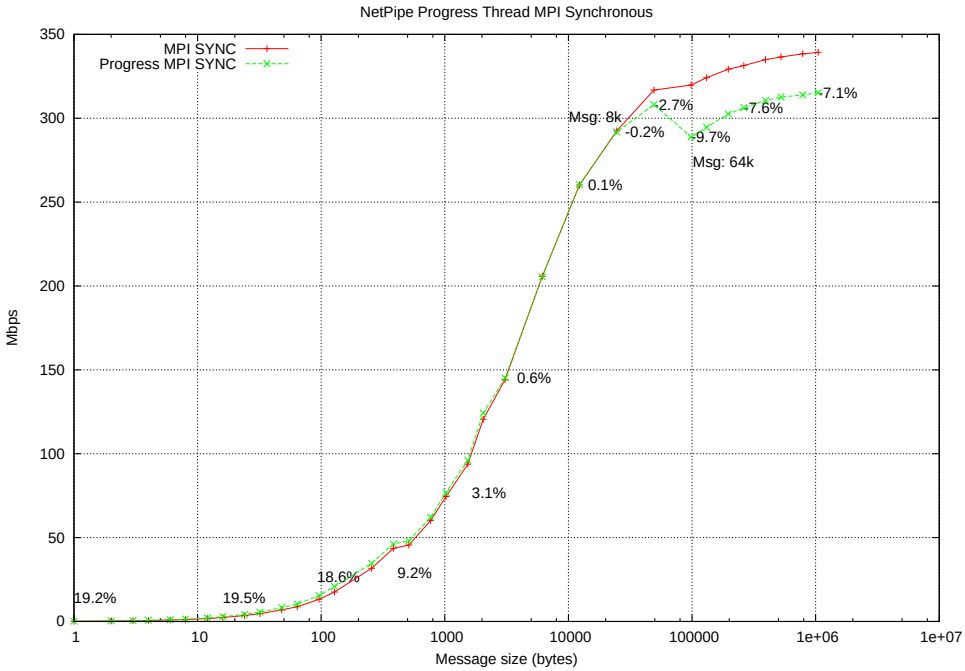


Figura 4.9: Comparação do desempenho do *OPENMPI* com *thread* de progresso – modo de comunicação síncrono

O thread de progresso é usado apenas com o protocolo *rendezvous*, empregado na transmissão de mensagens longas. Nota-se que há uma queda no desempenho com thread de progresso para mensagens maiores que 64kB, que corresponde ao limite entre mensagens curtas e longas no *OPENMPI*. A sobrecarga introduzida pelo uso do *thread* de progresso resulta em uma perda de desempenho da ordem de 7% no teste de largura de banda.

4.2.2 LogP

O *LogP Benchmark* [20] foi desenvolvido com a proposta de oferecer um método rápido para medir os parâmetros do LogP em um ambiente baseado no padrão MPI – evitando

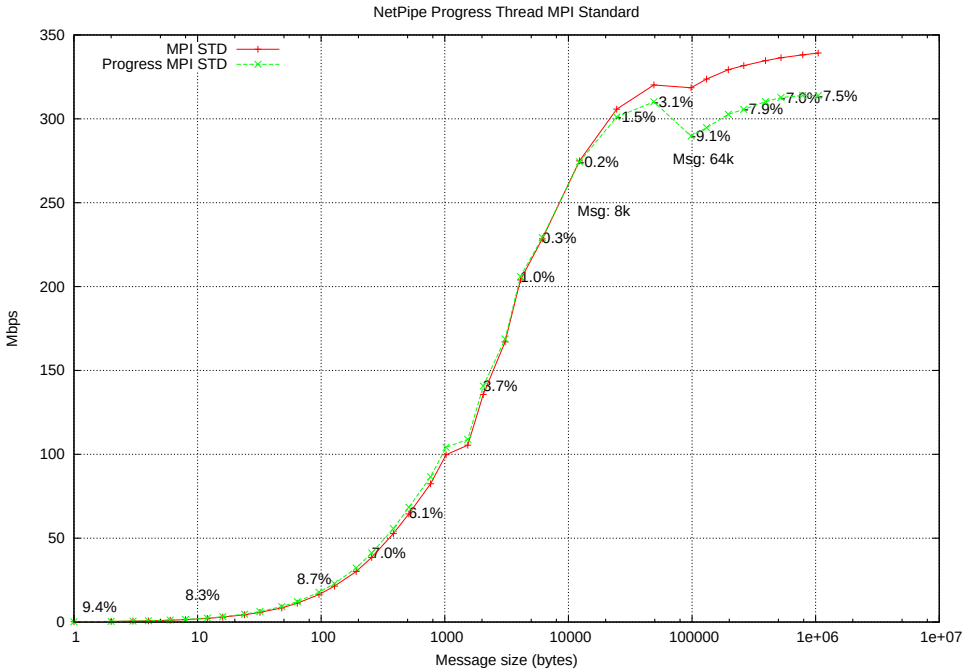


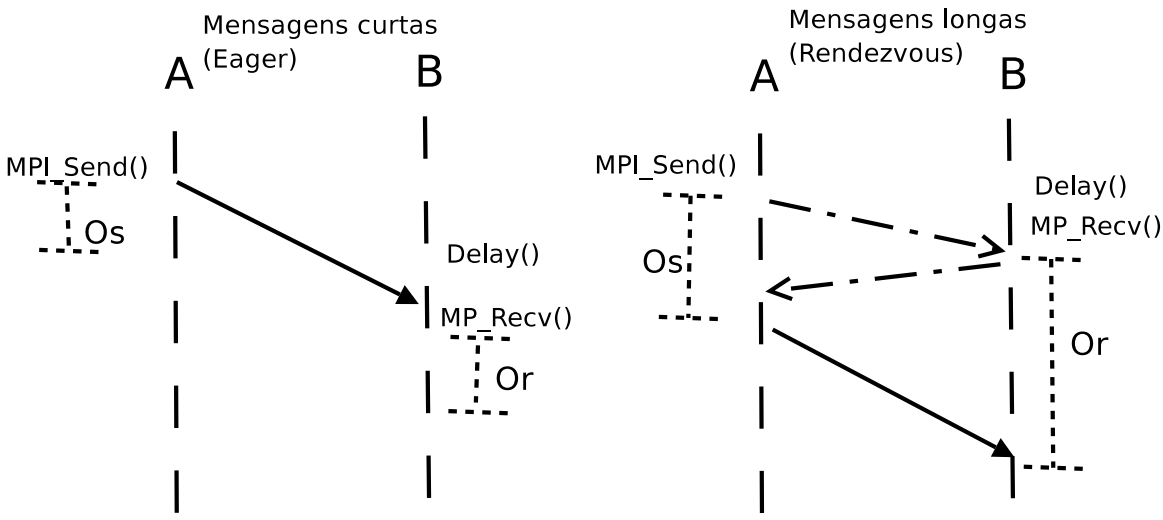
Figura 4.10: Comparação do desempenho do *OPENMPI* com *thread* de progresso – modo de comunicação padrão.

saturar a rede com mensagens longas. A rede é saturada apenas para obter o *gap* em mensagens de tamanho zero. As medidas para os outros tamanhos de mensagem são deduzidas a partir de testes de ping-pong e do *gap* medido para mensagens de tamanho zero.

Nesta Seção são apresentados os testes implementados pelo *LogP Benchmark* para medir os parâmetros do modelo LogP, e algumas limitações da abordagem usada nas medidas. Também são mostrados os resultados medidos no conjunto A de máquinas do Nautilus. O *LogP Benchmark* apresentou problemas na execução com o protocolo LLC, e a causa não foi identificada. Os resultados apresentados foram obtidos apenas com o protocolo TCP/IP.

A principal limitação identificada na implementação do *LogP Benchmark* está associada à medida da sobrecarga de processamento. A ferramenta considera o tempo consumido nas primitivas de comunicação como medida do *overhead*. Entretanto, o tempo que a primitiva consome para completar uma operação não corresponde necessariamente à sobrecarga adicional de processamento, como é mostrado a seguir.

A Figura 4.11(a) mostra o esquema usado pelo *LogP Benchmark* para medir os



(a) Mensagem curta

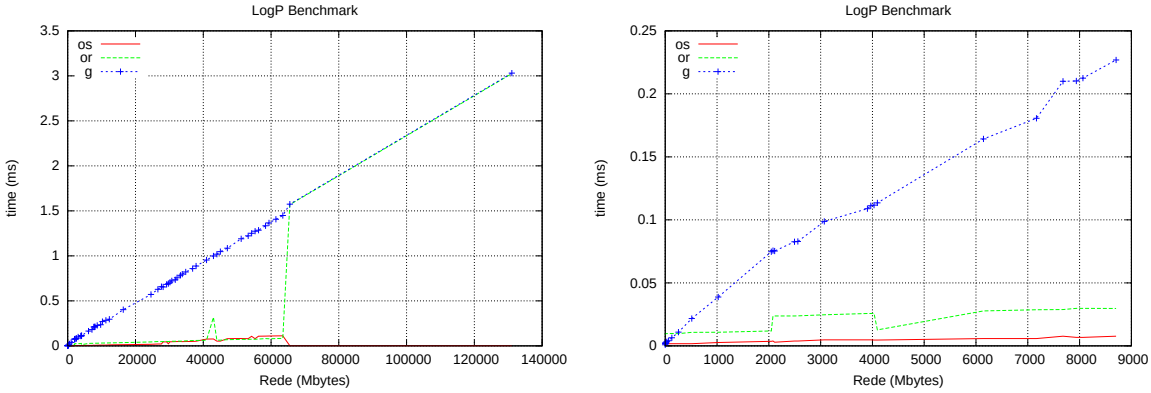
(b) Mensagem longa

Figura 4.11: *Esquema que representa a medida efetuada pelo LogP Benchmark com o uso de mensagens curtas (a) e com mensagens longas (b). As setas com linha cheia representam a troca da mensagem com os dados da aplicação, e com linhas pontilhadas representam trocas de mensagens de controle.*

parâmetros quando são usadas mensagens curtas (protocolo *eager*), e a Figura 4.11(b) o esquema para mensagens longas (protocolo *rendezvous*). Quando uma mensagem curta é transmitida com modo de comunicação padrão, a mensagem é copiada para um *buffer* intermediário do sistema, e a operação completa mesmo que a mensagem não tenha sido efetivamente transmitida. Assim, o sistema ainda tem uma carga de trabalho para efetivamente transmitir a mensagem, que pode implicar em carga de trabalho da CPU – e portanto em mais *overhead* de comunicação.

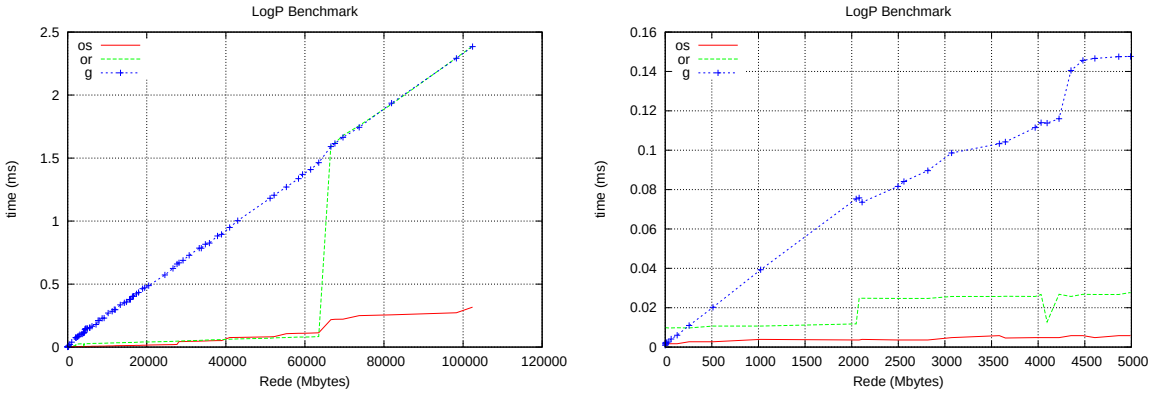
No caso do *overhead* de recepção, o *LogP Benchmark* também mede o tempo consumido para completar a operação. Para não incluir na medida o tempo decorrido antes que a mensagem chegue a destino, é adicionado um tempo de espera antes de iniciar a operação de recepção para garantir que ao iniciar a primitiva a mensagem já esteja disponível. Entretanto, antes de iniciar a primitiva de recepção a mensagem pode ser recebida pelo sistema de comunicação e armazenada em um *buffer* intermediário – dependendo do protocolo usado. Assim, ao iniciar a primitiva de recepção já foi consumido tempo de CPU para tratar interrupções e copiar a mensagem para o *buffer* intermediario – tempo que é ignorado pela medida.

A seguir são apresentados os resultados obtidos com o *LogP Benchmark* no conjunto A de máquinas do Nautilus, com a biblioteca *OPENMPI* e protocolo *TCP*. O modo com que o *LogP Benchmark* mede os parâmetros não são precisos, como descrito anteriormente, e por isso os resultados são diferentes dos obtidos com base em testes de sobreposição apresentados na Seção 4.2.3.6.



(a) todo intervalo (b) mensagens pequenas

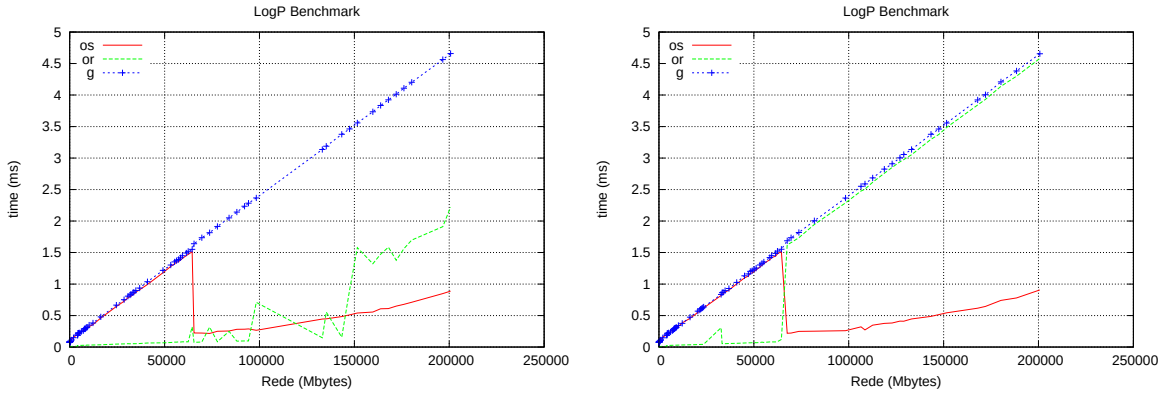
Figura 4.12: *Resultados obtidos com o LogP Benchmark com primitiva não-bloqueante e modo de comunicação padrão.*



(a) todo intervalo (b) mensagens pequenas

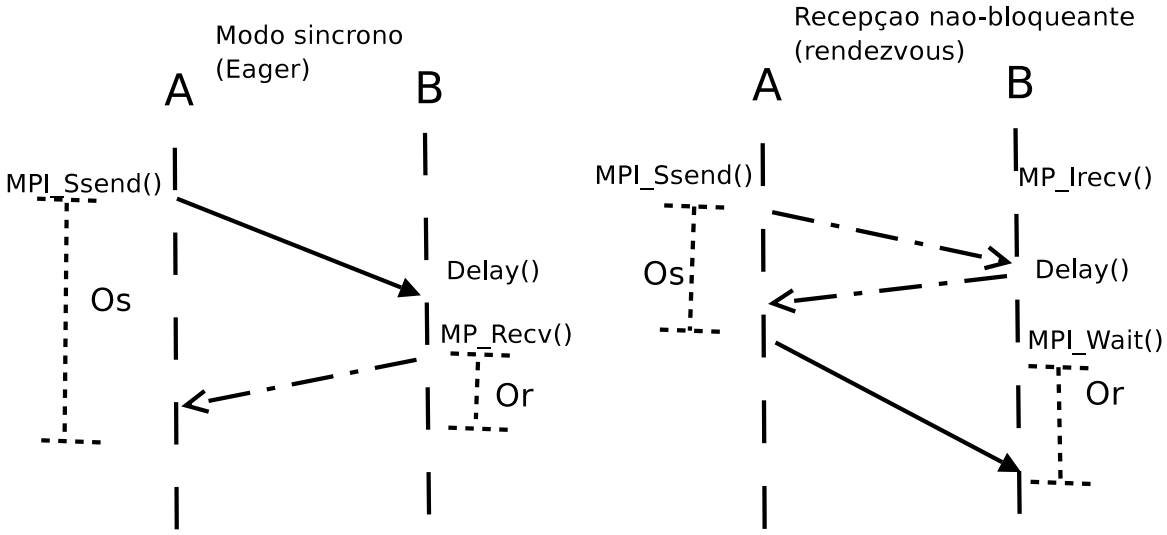
Figura 4.13: *Resultados obtidos com o LogP Benchmark com primitiva bloqueante e modo de comunicação padrão.*

A Figura 4.12 mostra os resultados com primitiva não-bloqueante e modo de comunicação padrão, e a Figura 4.13 os resultados com primitiva bloqueante – também com modo de comunicação padrão. Nos resultados com modo de comunicação padrão, nota-se que há uma alteração abrupta da medida do *overhead* de recepção na transição entre mensagens curtas e longas, que para o OPENMPI ocorre em 64kB. Na transição,



(a) recebimento não-bloqueante (b) recebimento bloqueante

Figura 4.14: *Resultados obtidos com o LogP Benchmark com primitiva de envio bloqueante e modo de comunicação síncrono.*



(a) Síncrono (eager) (b) Recepção não-bloqueante

Figura 4.15: *Esquema que representa a medida efetuada pelo LogP Benchmark com modo de comunicação síncrono e mensagens curtas (a), e a medida efetuada com primitiva de recepção não-bloqueante (b). As setas com linha cheia representam a troca da mensagem com os dados da aplicação, e com linhas pontilhadas representam trocas de mensagens de controle.*

o *overhead* de recepção passa de 0.0855 *ms* para 1.5771 *ms*. Esta mudança observada na transição pode estar relacionada ao modo como o parâmetro é medido, e não a um crescimento da sobrecarga de CPU envolvida na comunicação. Na transição, o MPI passa a usar o protocolo de mensagens longas (*rendezvous*) para transmissão da mensagem. Com o modo de comunicação padrão, e mensagens menores que 64kB, ao iniciar a primitiva de recepção a mensagem já foi armazenada localmente em um *buffer* intermediário, e por este

motivo completa rapidamente. No protocolo *rendezvous* usado para mensagens longas, ao iniciar a primitiva de recepção a mensagem ainda não foi completamente transmitida para a estação de destino. Para completar a operação, é enviada uma mensagem de aceitação e só então a mensagem é considerada efetivamente transmitida. Portanto, o tempo consumido com a primitiva de recebimento não corresponde necessariamente à carga de trabalho da CPU.

A Figura 4.14 mostra os resultados do *LogP Benchmark* com primitiva bloqueante e modo de comunicação síncrono. O gráfico da direita (a) mostra o resultado do teste combinado com primitiva de recebimento não-bloqueante, e o gráfico da direita (b) mostra o resultado do teste com primitiva de recebimento bloqueante. O esquema da Figura 4.15 mostra como o *LogP Benchmark* efetua as medidas com modo de comunicação síncrono.

Quando é usado o modo de comunicação síncrono, o *OPENMPI* utiliza o protocolo *rendezvous* na trasmissão da mensagem, entretanto com uma abordagem diferente no caso de mensagens curtas. Na transmissão de mensagens curtas com modo de comunicação síncrono, a primeira mensagem do *rendezvous* com a solicitação de envio contém também parte dos dados da mensagem – ou toda mensagem se esta for suficientemente pequena. Entretanto, a primitiva só completa quando a aceitação for recebida – mesmo que todos os dados já tenham sido transferidos na primeira mensagem.

A Figura 4.15(a) mostra como é obtido o *overhead* de envio e recepção com modo de comunicação síncrono para mensagens curtas, transmitidas com o protocolo *eager*. O processo B, que recebe a mensagem, aguarda um intervalo de tempo suficiente para que a mensagem esteja disponível para recepção. Quando a primitiva de recebimento é iniciada pelo processo B, a mensagem já está armazenada em um *buffer* intermediário do sistema, e assim o parâmetro *Or* representa apenas o tempo consumido para obter a mensagem armazenada localmente. O *overhead* de envio inclui todo tempo consumido para completar a mensagem, que abrange a transmissão dos dados e a recepção da mensagem de confirmação. Na transmissão de mensagens longas, representada na Figura 4.15(b), o processo A envia uma mensagem requisitando a transmissão da mensagem. Quando a primitiva de recepção é iniciada no processo B, este envia uma mensagem de aceitação,

e então o processo A envia a mensagem com os dados da aplicação. A primitiva de envio completa mediante a aceitação do processo B, mesmo que a mensagem não tenha sido transferida efetivamente. A medida do parâmetro *Os* inclui o tempo consumido na negociação – que não corresponde necessariamente à carga de trabalho da CPU – mas descarta tempo de CPU envolvido na transmissão da mensagem com os dados da aplicação. O parâmetro *Or* inclui o tempo consumido com a negociação e com a recepção da mensagem com os dados da aplicação.

Quando o *LogP Benchmark* faz uso de primitiva de recebimento não-bloqueante, a primitiva que configura a recepção da mensagem no processo B é escalonada no início do teste – conforme representado na Figura 4.15(b). O parâmetro *Or* corresponde ao tempo consumido na primitiva MPI_Wait, usada para completar a recepção. Porém, como a operação é configurada previamente, o processo B envia a mensagem de aceitação e a comunicação pode progredir antes de ser iniciada a primitiva MPI_Wait.

4.2.3 Perftest

O Perftest é uma ferramenta usada para medir o desempenho em ambientes baseados no padrão MPI. O Perftest é distribuído com o MPICH, e também está disponível separadamente para cópia [10]. As medidas de desempenho podem ser reproduzidas em diferentes ambientes e com quaisquer implementações do padrão MPI.

Além de medidas tradicionais de ping-pong, o Perftest inclui outros testes como troca de mensagens e sobreposição de comunicação com computação. Os testes de comunicação ponto-a-ponto podem ser efetuados entre um ou mais pares de processos simultaneamente, o que permite avaliar a escalabilidade e a contenção com o crescimento do número de processos envolvidos.

Para evitar medidas discrepantes, cada teste repete diversas vezes o algoritmo, e o tempo decorrido é dividido pelo número de repetições para obter o tempo médio, que é adotado como resultado de cada execução do teste. Cada execução é repetida um

[10] Manual de uso da ferramenta e link para download podem ser encontrados em: `http://www-unix.mcs.anl.gov/mpi/mpptest/`

determinado número de vezes. Portanto, em uma rodada completa, há dois laços de repetição: um laço interno para obter o tempo médio de execução do algoritmo a cada execução do teste, e um laço externo para repetir o teste e armazenar os resultados. O número de repetições dos testes para obter a média, bem como o número de repetições de cada execução podem ser passados como argumentos para o programa. Quando o tempo de uma execução não é significativamente maior que a resolução do relógio (pelo menos 100 vezes maior), o teste é executado com um número maior de repetições. O menor tempo, o maior tempo, e o tempo médio obtidos nas repetições do teste são gravados em um arquivo de saída.

A versão atual do Perftest é 1.3a. Com base nesta versão foram implementados novos testes para melhor avaliar o sistema de comunicação. Foram criados novos testes de sobreposição de comunicação com computação, sobrecarga de processamento, novas implementações do teste de troca com abordagens distintas, testes de comunicação com o padrão todos-para-todos e testes de largura de banda. A seguir estão descritos alguns dos testes do Perftest, e também os novos testes desenvolvidos com base na estrutura do Perftest. Alguns gráficos apresentados neste Seção foram divididos em grupos de acordo com o tamanho das mensagens. Mensagens pequenas com até 4kB, mensagens médias entre 8kB e 80kB e mensagens grandes entre 96kB e 1MB. A divisão nestes três intervalos é apenas para melhorar a visualização, e não tem relação com a distinção entre mensagens curtas e longas feita pelo MPI.

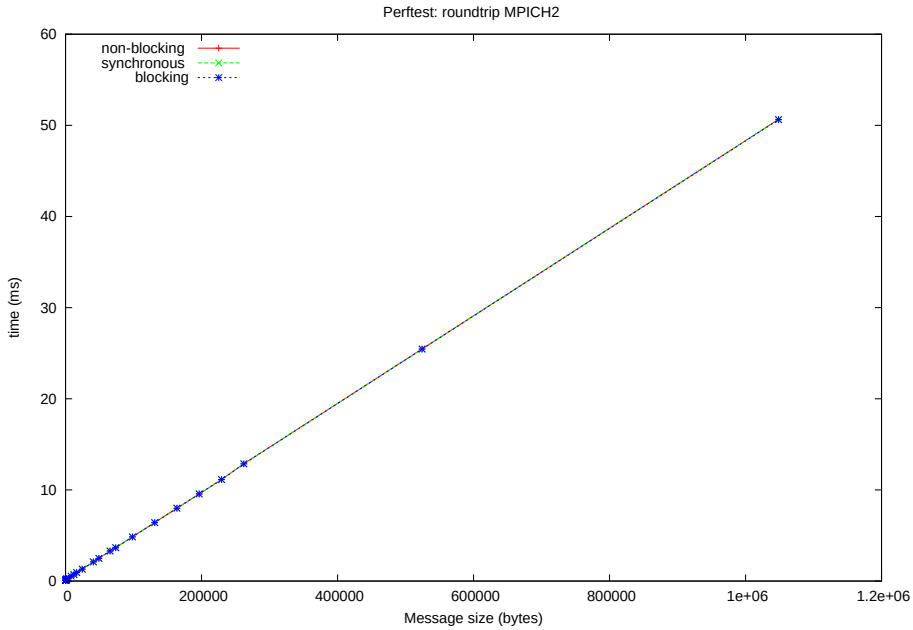
Processo A	Processo B

`start = get_time();`
`if (int i = 0; i < AVG_REPS; i++){`
`  MPI_Send();`
`  MPI_Recv();`
`}`
`finish = get_time();`
`return (finish - start);`
	`start = get_time();`
	`if (int i = 0; i < AVG_REPS; i++){`
	`  MPI_Recv();`
	`  MPI_Send();`
	`}`
	`finish = get_time();`
	`return (finish - start);`

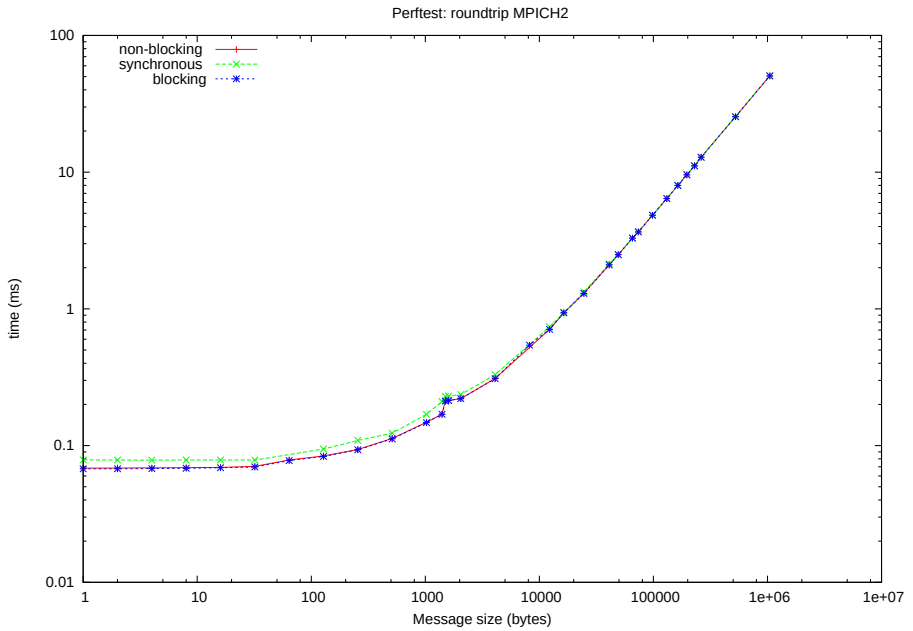
Tabela 4.3: Teste de ping-pong, ou *roundtrip*, entre dois processos A e B.

4.2.3.1 Ping-pong

No Perftest, o teste de ping-pong é chamado de *roundtrip*, e está representado na Tabela 4.3. A metade do tempo do teste de ping-pong corresponde à latência fim-a-fim do modelo (EEL). O inverso da latência fim-a-fim é a largura de banda atingida na transmissão de mensagens não consecutivas, como no teste unidirecional do NetPIPE.



(a) escala normal



(b) escala logarítimica

Figura 4.16: *Teste de ping-pong com a biblioteca MPICH2, à esquerda em escala normal e à direita em escala logarítimica.*

No gráfico da Figura 4.16(b) pode-se observar que a implementação baseada no modo de comunicação síncrono possui desempenho pior que as demais para mensagens pequenas. O desempenho do modo síncrono se equipara ao desempenho da primitiva bloqueante com modo padrão na transição entre mensagens curtas e longas, que para o *MPICH2* ocorre em 16kB.

O tempo do teste de ping-pong para mensagens de 1MB com a biblioteca *MPICH2* é de 50.62 ms. A metade do tempo do ping-pong corresponde à latência fim-a-fim (EEL), que para mensagem de 1MB é de 25.31 ms. A largura de banda pode ser formulada como $(m/EEL) * 8$, em que m é o tamanho da mensagem em *bytes* e o fator 8 é usado para obter o resultado em *bits* por segundo. Logo, para mensagens de tamanho 1MB, pode-se obter a largura de banda por $(1MB/0.02531) * 8 = 316.08Mbps$, que é equivalente ao resultado obtido com o NetPIPE no teste unidirecional.

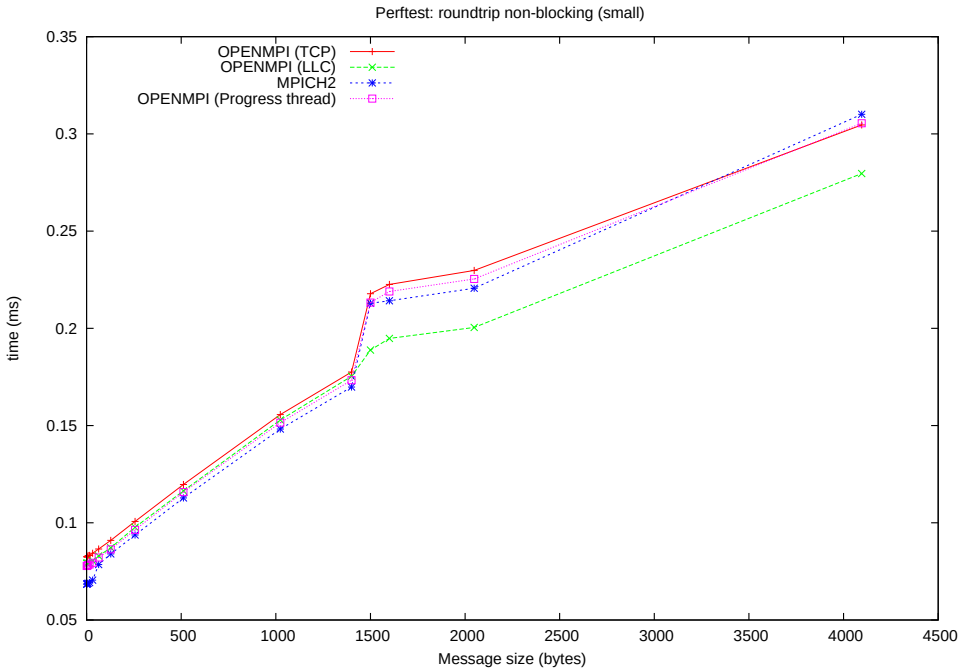


Figura 4.17: Comparação do resultado do teste de ping-pong com mensagens de tamanho pequeno em diferentes configurações (*OPENMPI-TCP*, *OPENMPI-LLC*, *OPENMPI* com *Thread* de progresso e *MPICH2*).

Os gráficos das Figuras 4.17 e 4.18 mostram os resultados do teste de ping-pong não-bloqueante para mensagens pequenas e grandes, respectivamente. Os gráficos comparam os resultados das configurações *OPENMPI* com LLC, *OPENMPI* com TCP,

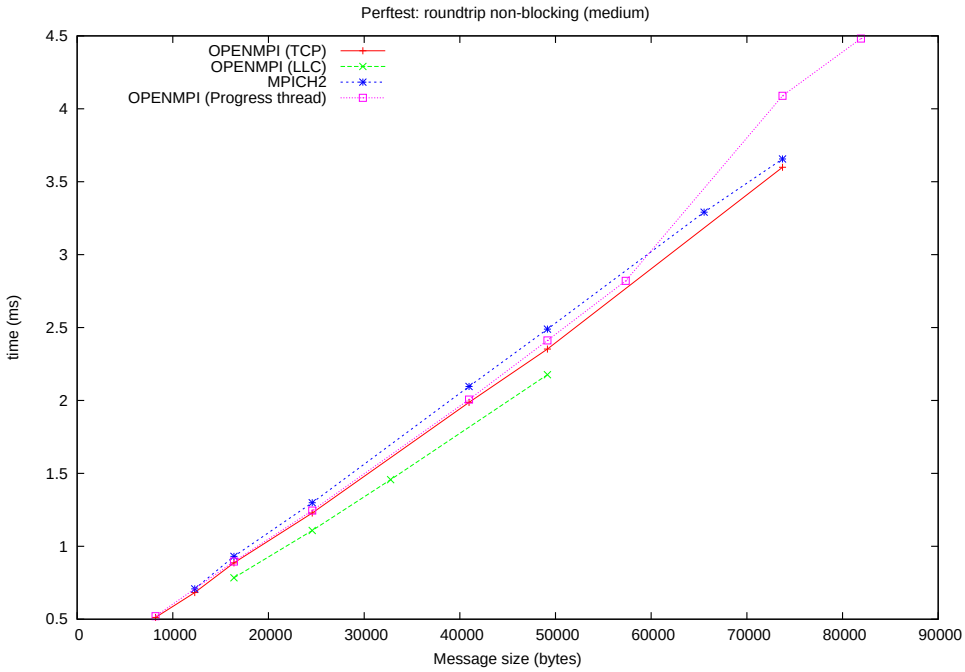


Figura 4.18: Comparação do resultado do teste de ping-pong com mensagens de tamanho médio em diferentes configurações (*OPENMPI-TCP*, *OPENMPI-LLC*, *OPENMPI* com *Thread* de progress e *MPICH2*.

OPENMPI com *Thread* de progresso e *MPICH2*. Para mensagens maiores que 1500B, o *LLC* apresenta ganho de desempenho da ordem de 10%, como está representado na Figura 4.17. A transição entre mensagens curtas e longas representada na Figura 4.18 mostra o efeito do uso de um *thread* de progresso para mensagens maiores que 64kB com primitiva não-bloqueante. O *OPENMPI* com *LLC* apresenta o melhor desempenho, mas tem apenas medidas para mensagens com até 32kB representadas no gráfico – mensagens maiores que 32kB reportaram resultados erráticos e foram eliminados.

O gráfico da Figura 4.19 mostra o resultado do teste de ping-pong executado simultaneamente entre 1, 2 e 4 pares de processos com mensagens grandes. O tempo do teste não se altera com o crescimento do número de processos envolvidos, o que indica que não há contenção na rede para até 4 pares em execução simultânea. Note que as três curvas estão sobrepostas.

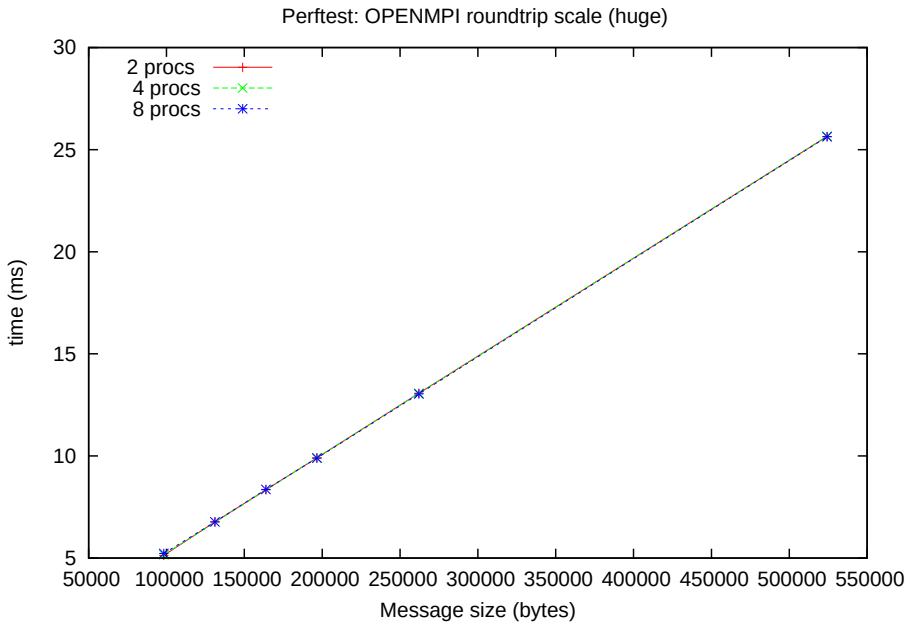
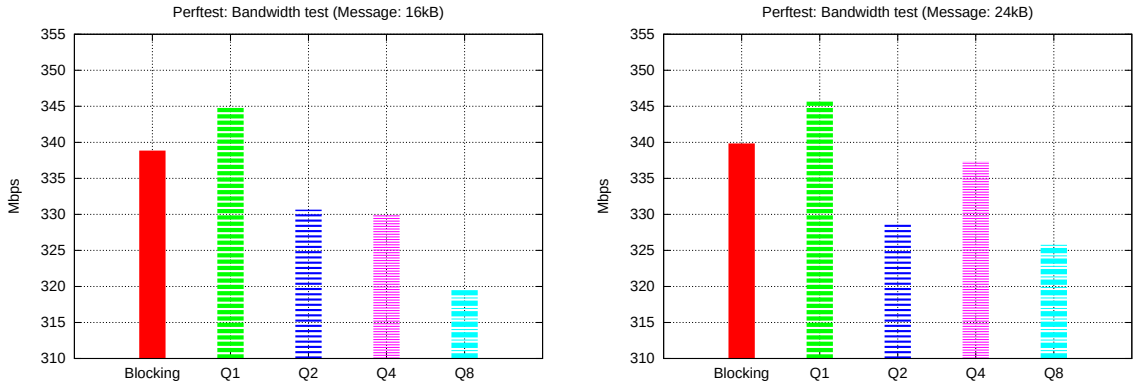


Figura 4.19: Teste de escalabilidade com o teste de ping-pong para 2, 4 e 8 processos simultâneos.



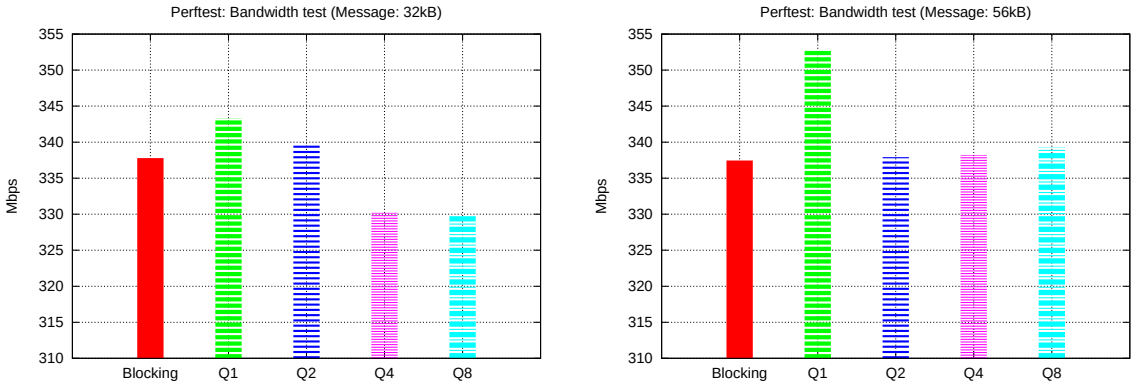
(a) 16kB (b) 24kB

Figura 4.20: *Resultado do teste de largura de banda com mensagens de 16kB (a) e 24kB (b), e com filas de tamanho 1, 2, 4 e 8 mensagens, além do resultado com primitiva bloqueante.*

4.2.3.2 Largura de banda

Foi implementado um novo teste de largura de banda que segue a abordagem descrita em [4], em que são usadas filas para manter uma ou mais mensagens pendentes através de primitivas não-bloqueantes. O teste solicita o envio de Q mensagens, em que Q é o tamanho da fila. Quando $Q/2$ mensagens são completadas, o teste solicita o envio de mais $Q/2$ mensagens, e assim sucessivamente. O algoritmo está descrito na Tabela 4.4.

A seguir são apresentados os resultados para mensagens entre 16kB e 72kB. Outros



(a) 32kB (b) 56kB

Figura 4.21: *Resultados do teste de largura de banda com mensagens de 32kB (a) e 56kB (b), e com filas de tamanho 1, 2, 4 e 8 mensagens, além do resultado com primitiva bloqueante.*

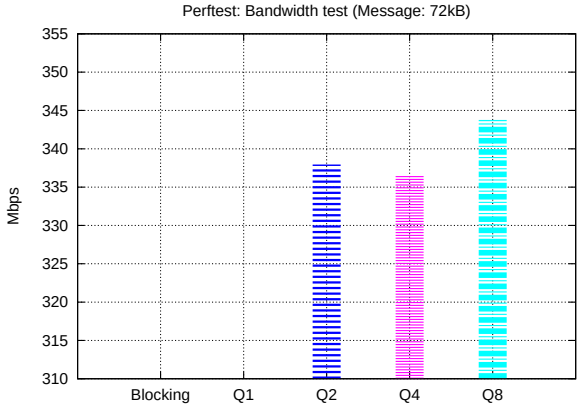


Figura 4.22: *Resultados do teste de largura de banda com mensagens de 72kB, e com filas de tamanho 1, 2, 4 e 8 mensagens, além do resultado com primitiva bloqueante.*

tamanhos de mensagem foram descartados por apresentarem medidas erráticas. Porém, com este conjunto de testes pode-se avaliar o desempenho obtido com o uso de filas e a eficiência do sistema de comunicação em tratar grupos de mensagens pendentes.

As Figuras 4.20(a), 4.20(b), 4.21(a), 4.21(b) e 4.22 apresentam os gráficos com os resultados do teste de largura de banda com mensagens de 16kB, 24kB, 32kB, 56kB e 72kB, respectivamente. Os resultados foram obtidos com a biblioteca *OPENMPI* com o protocolo *TCP* no conjunto A de máquinas do Nautilus. Os gráficos mostram os resultados com filas de tamanho 1, 2, 4 e 8, além do teste com versão bloqueante. O teste com mensagem de tamanho 72kB apresentou resultados erráticos com fila de tamanho um, que foram eliminados do gráfico (4.22). O teste com fila de tamanho 1 obteve a maior

Largura de banda com fila (Q)

```
//solicita q/2 mensagens
for (j = 0; j < queue_len/2; j++){
    MPI_Isend();
}

t0=get_time();
for(i=1;i < iters; i++){
  //inicia q/2 messagens
  for (k = 0; k < queue_len/2; k++){
       MPI_Isend();
  }
  //aguarda pelas primeiras q/2 mensagens
  MPI_Waitall(queue_len/2);
}
MPI_Waitall(queue_len/2);

//garante que todas as msgs foram recebidas
MPI_Recv();
t1=get_time();
```

Tabela 4.4: Algoritmo do teste de largura de banda com fila (Q) de tamanho maior do que 1.

largura de banda observada com este conjunto de mensagens. O desempenho com filas de tamanho 1 supera o desempenho obtido com primitivas bloqueantes. Com mensagens de 16kB e 24kB, os testes bloqueantes apresentaram melhor desempenho do que os testes com filas de tamanho 2, 4 e 8. Para mensagens maiores, com 32kB e 56kB, o desempenho dos testes com fila se aproxima do desempenho do teste bloqueante. O teste com fila de tamanho 8 tem desempenho muito inferior aos demais com mensagens de 16kB, mas apresenta ganho em relação aos demais com o crescimento do tamanho da mensagem, e supera o desempenho com filas de tamanho 2 e 4 com mensagens de 72kB.

A Figura 4.23 apresenta o gráfico com os resultados obtidos em um par de estações do *C3SL* no teste de largura de banda com mensagens de 8 bytes a 1MB. Neste ambiente a estratégia de agrupar as mensagens também não tem efeito positivo no desempenho. Para todos os tamanhos de mensagem o desempenho do teste com fila de tamanho 1 é melhor que os demais, chegando a 949.38 *Mbps* em mensagens com 160MB.

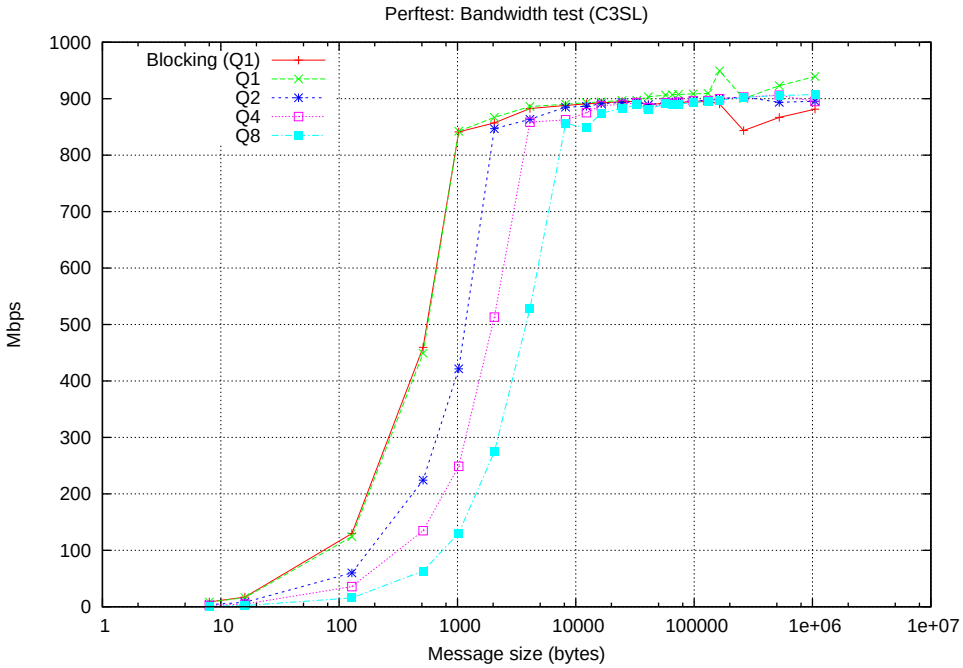


Figura 4.23: Teste de largura de banda em um par de estações do *C3SL* com mensagens entre 8 bytes e 1MB – em escala logarítimica no eixo X.

4.2.3.3 Troca

O Perftest implementa testes de troca de mensagens entre pares de processos. O *teste de troca* consiste no envio de uma mensagem do processo A para o processo B e vice-versa, em que o envio é escalonado antes da recepção nos dois processos envolvidos, de forma que o envio das mensagens ocorre nas duas direções simultaneamente.

Os testes de troca baseados em primitivas bloqueantes e não-bloqueantes de recebimento fazem parte do conjunto de testes da ferramenta. Foram desenvolvidas novas implementações do teste de troca que empregam as primitivas MPI_Sendrecv e MPI_Sendrecv_replace. As diferentes implementações do teste de troca estão descritas a seguir, e o algoritmo de cada teste está representado na Tabela 4.5. Os dois processos envolvidos na troca implementam o mesmo algoritmo.

Troca com recebimento bloqueante Na troca com recebimento bloqueante, os dois processos do par enviam uma mensagem através da primitiva MPI_Send e iniciam a recepção através da primitiva MPI_Recv, como mostra o algoritmo descrito na Tabela 4.5 (acima e à esquerda). Como descrito na Seção 2.1.1, a operação de envio padrão completa

Primitiva bloqueante	Primitiva não-bloqueante

`start = get_time();`
`if (int i = 0; i < AVG_REPS; i++){`
`  MPI_Send();`
`  MPI_Recv();`
`}`
`finish = get_time();`
`return (finish - start);`
	`start = get_time();`
	`if (int i = 0; i < AVG_REPS; i++)`
	`{`
	`  //configura recebimento`
	`  MPI_Irecv(mensagem);`
	`  MPI_Send(mensagem);`
	`  MPI_Wait(mensagem);`
	`}`
	`finish = get_time();`
	`return (finish - start);`

Primitava MPI_Sendrecv	

`start = get_time();`
`if (int i = 0; i < AVG_REPS; i++)`
`{`
`  MPI_Sendrecv(mensagem);//troca`
`}`
`finish = get_time();`
`return (finish - start);`	

Tabela 4.5: Algoritmos do teste de troca com primitiva bloqueante, com primitiva não-bloqueante e modo padrão e com primitiva de envio e recebimento conjugadas.

quando o *buffer* de envio puder ser reusado, e não necessariamente quando o destino receber a mensagem. Entretanto, como ocorre para mensagens longas, se a operação de envio completar apenas após o início da recepção pelo destino, o algoritmo de troca implementado com recebimento bloqueante pode entrar numa situação de *deadlock*.

Troca com recebimento não-bloqueante Na versão não-bloqueante do teste de troca, os processos envolvidos preparam uma operação de recebimento não-bloqueante através da primitiva MPI_Irecv, e então iniciam o envio da mensagem ao destino através de uma operação bloqueante. Nesta implementação, como a operação de recebimento é configurada antes do envio da mensagem não há como ocorrer uma situação de *deadlock*.

O algoritmo baseado em recebimento não-bloqueante possui implementação com os modos de comunicação síncrono e padrão, e está descrito na Tabela 4.5 (acima e à direita).

Troca com primitivas de envio e recebimento conjugadas O padrão *MPI* define funções que conjugam o envio e recebimento de mensagens em uma

única primitiva: MPI_Sendrecv e MPI_Sendrecv_replace. A primitiva MPI_Sendrecv recebe *buffer*s distintos para envio e recebimento, e a primitiva MPI_Sendrecv_replace recebe a mensagem no mesmo *buffer* usado para o envio. Foram implementados dois novos testes, um baseado na primitiva MPI_Sendrecv e outro baseado na primitiva MPI_Sendrecv_replace, ambos seguem o algoritmo descrito na Tabela 4.5 (abaixo e à esquerda).

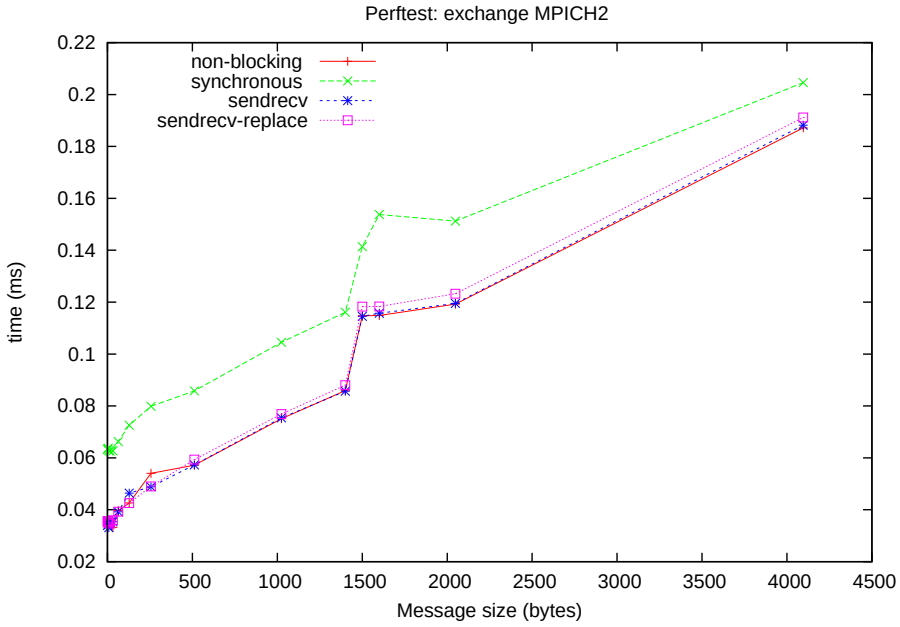
No gráfico da Figura 4.24(a) pode-se observar que a implementação baseada no modo de comunicação síncrono apresenta o pior desempenho para mensagens curtas, com até 16kB, no caso do MPICH2.

As implementações do teste de troca baseadas nas primitivas de envio e recebimento conjugados apresentam desempenho pior do que a implementação com primitivas de envio e recebimento separadas e com modo de comunicação padrão. Das implementações com envio e recebimento conjugados, a primitiva MPI_Sendrecv_replace apresenta o pior desempenho, cuja diferença em relação à implementação não-bloqueante varia de 1.6% para mensagens de 1B até 5% para mensagens com 1MB. Os demais casos apresentam desempenho similar.

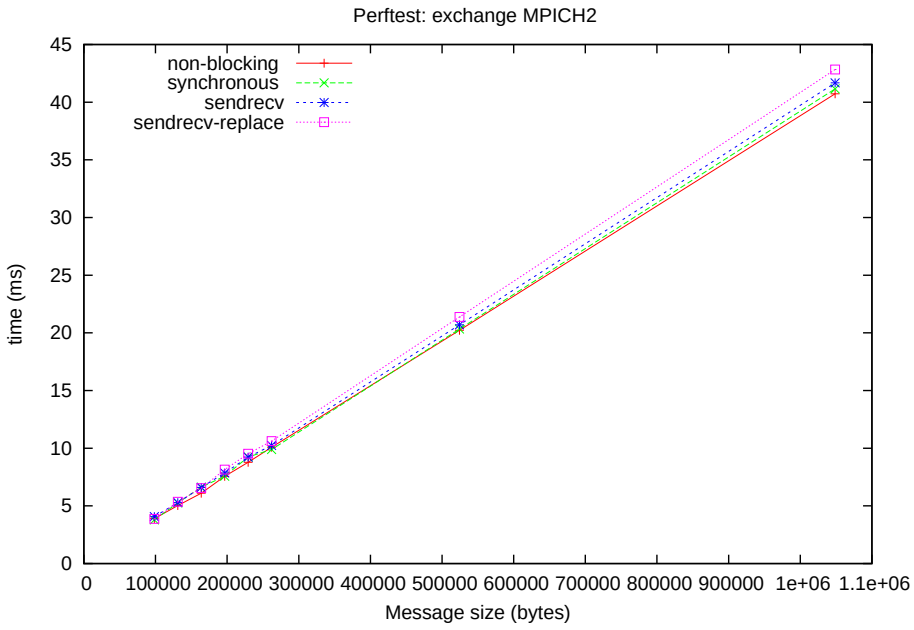
4.2.3.4 Todos-para-Todos

Neste trabalho, foram desenvolvidos testes com padrão de comunicação *todos-para-todos*, no qual cada processo troca dados com todos os outros do mesmo grupo de comunicação. Os testes com este padrão de comunicação foram implementados seguindo abordagens diferentes com o objetivo de identificar a melhor implementação em cada situação. Uma das implementações do teste utiliza uma operação coletiva do padrão *MPI* que realiza a comunicação do tipo todos-para-todos através de uma única primitiva: MPI_Alltoall. As demais implementações do teste todos-para-todos são baseadas em primitivas de comunicação ponto-a-ponto, em que o algoritmo realiza uma operação de troca com cada participante do grupo de comunicação.

A seqüência das trocas é ordenada de tal forma que, a cada iteração, quando o processo A elege o processo D como destino, o processo D elege A como origem. A ordem garante



(a) mensagens pequenas



(b) mensagens grandes

Figura 4.24: *Resultado do teste de troca com a biblioteca MPICH2, para mensagens pequenas (a) e para mensagens grandes (b).*

que as trocas são realizadas em pares, e que a cada iteração um processo participa apenas de uma troca, evitando dependências que possam causar *deadlock*.

Primitiva de comunicação coletiva do padrão MPI O padrão *MPI* define uma primitiva que realiza a comunicação no padrão todos-para-todos

Envio bloqueante	Envio não-bloqueante

`//conf recepção para cada parceiro`
`if (int p = 0; p <n_parceiros ; i++){`
`origem = parceiros[p];`
`MPI_Irecv(origem, mensagem);`
`}`
`//envia para cada parceiro`
`if (int p = 0; p <n_parceiros ; i++){`
`destino = parceiros[p];`
`MPI_Send(destino, mensagem);`
`}`
`//aguarda completar`
`MPI_Waitall(recebimentos);`
	`//conf. recebimento para cada parceiro`
	`//conf. recebimento para cada parceiro`
	`if (int p = 0; p <n_parceiros ; i++){`
	`origem = parceiros[p];`
	`MPI_Irecv(origem, mensagem);`
	`}`
	`//envia para cada parceiro`
	`if (int p = 0; p <n_parceiros ; i++){`
	`destino = parceiros[p];`
	`//envio não-bloqueante`
	`MPI_Isend(destino, mensagem);`
	`}`
	`//aguarda envios pendentes`
	`MPI_Waitall(envios);`
	`//aguarda recebimentos pendentes`
	`MPI_Waitall(recebimentos);`

Tabela 4.6: Teste do padrão de comunicação todos-para-todos com envio bloqueante (à esquerda) e com envio não bloqueante (à direita). Todos os processos envolvidos implementam o mesmo algoritmo.

(MPI_Alltoall). Um teste baseado nesta primitiva foi implementado com base na estrutura do Perftest. O teste consiste no uso direto da primitiva MPI_Alltoall(), que implementa a troca com todos os participantes do teste.

Primitiva de envio e recebimento conjugados Foram implementados dois testes com esta abordagem, um baseado na primitiva MPI_Sendrecv e outro baseado na primitiva MPI_Sendrecv_replace.

Primitiva de envio bloqueante No teste baseado em primitivas bloqueantes, as operações de recebimento são configuradas previamente, e as mensagens são enviadas aos destinatários através da primitiva MPI_Send. O algoritmo só aguarda o recebimento de cada mensagem dos outros parceiros após terminar as operações de envio, como mostra o algoritmo descrito na Tabela 4.6 (à esquerda).

Primitiva de envio não-bloqueantes No teste baseado em primitivas não-bloqueantes, as operações de recebimento são configuradas previamente, e o envio das mensagens é solicitado através de primitivas não-bloqueantes de envio. O algoritmo

configura os recebimentos e solicita o envio para todos os parceiros, e então aguarda todos os recebimentos e envios configurados anteriormente, como mostra o algoritmo da Tabela 4.6.

O gráfico da Figura 4.25 mostra o resultado do teste todos-para-todos entre 2 processos e com mensagens pequenas, no qual o desempenho das implementações com primitivas de envio e recebimento conjugados são melhores que os demais. Este cenário muda com o crescimento do número de processos, como pode ser observado no gráfico da Figura 4.26, que mostra os resultados para 8 processadores com mensagens pequenas. Neste caso, as diferenças variam de 50% para mensagens pequenas, até 80% para mensagens de 4kB.

O gráfico da Figura 4.27 mostra os resultados para todo conjunto de mensagens testado, de 1B a 1MB, em escala logarítimica. Com 8 processadores, o teste com mensagens curtas tem melhor desempenho com as implementações que empregam primitivas de envio e recebimento separadas, e com a operação coletiva do MPI. No entanto, na transição entre mensagens curtas e longas (64kB), as implementações com envio bloqueante e não-bloqueante apresentam desempenho pior, e voltam a se aproximar do desempenho das outras implementações para mensagens maiores que 10kB.

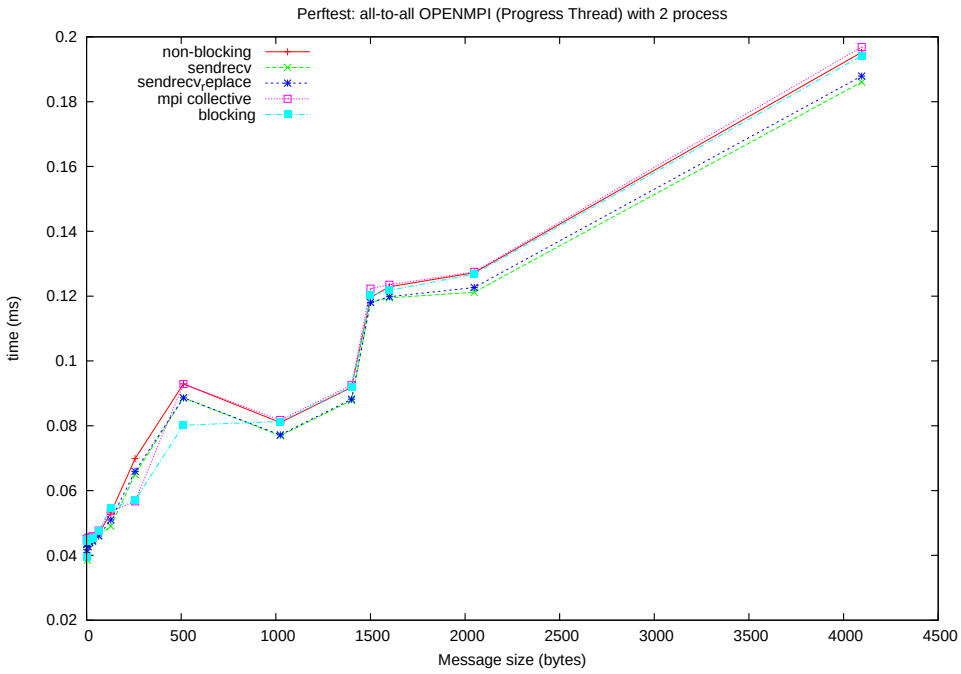


Figura 4.25: Comparação das diferentes implementações do teste todos-para-todos com mensagens pequenas e com 2 processos, obtidos com a biblioteca *OPENMPI*.

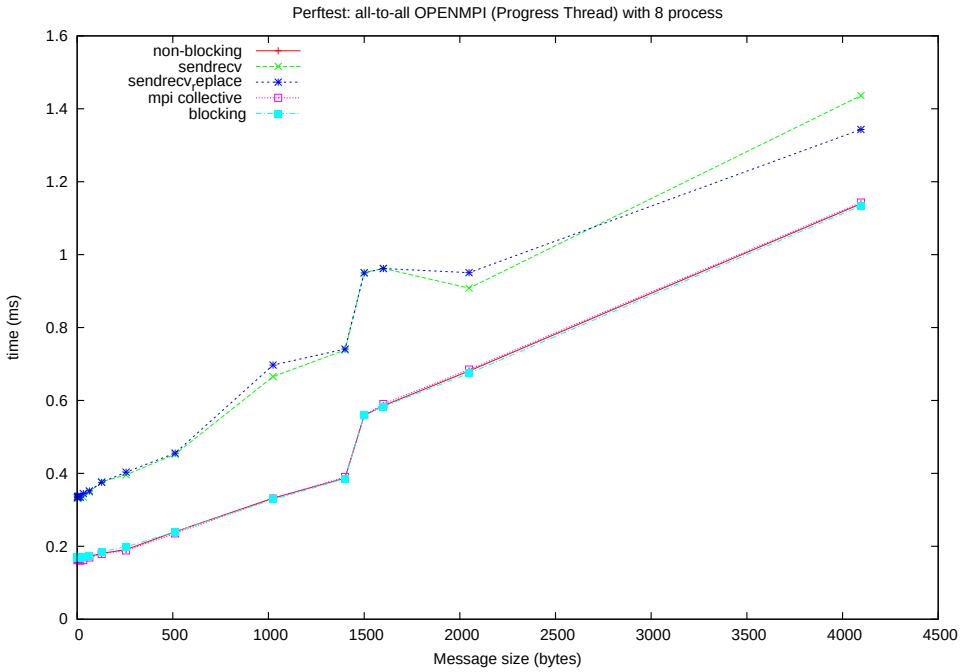


Figura 4.26: Comparação das diferentes implementações do teste todos-para-todos com mensagens pequenas e com 8 processos, obtidos com a biblioteca *OPENMPI*.

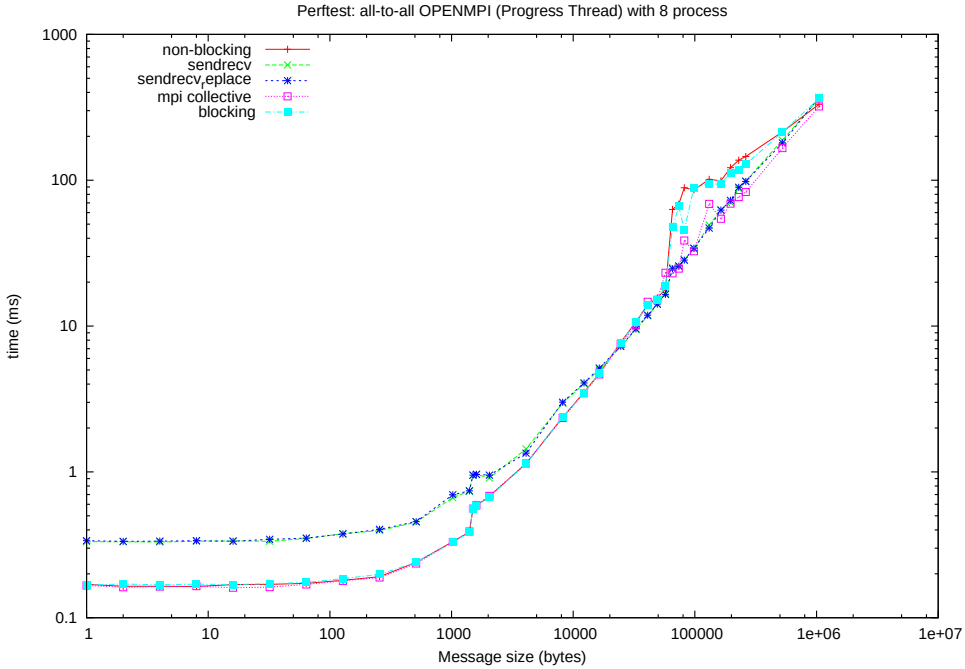


Figura 4.27: Comparação das diferentes implementações do teste todos-para-todos com 8 processos – em escala logarítimica.

4.2.3.5 Sobreposição de comunicação com computação

Os testes de sobreposição de comunicação com computação encontrados na versão oficial do Perftest são baseados em teste do tipo ping-pong. Novos testes de sobreposição foram

Processo A	Algoritmo DAXPY

`start = get_time();`
`if (int i = 0; i < AVG_REPS; i++){`
`  MPI_Send();`
`  OverlapComputation(len);`
`  MPI_Recv();`
`}`
`finish = get_time();`
`return (finish - start);`
	`//cada buffer tem no mínimo 64kB`
	`//chamadas consecutivas usam areas`
	`//differentes do buffer`
	
	`OverlapComputation(len){`
	`  double temp;`
	`  n = len / sizeof(double);`
	`  for ( i=0; i < n; i++) {`
	`  temp+= buffer1[i] * buffer2[i];`
	`  }`
	`}`

Tabela 4.7: Algoritmo do teste de sobreposição com padrão de comunicação ping-pong (à esquerda). Algoritmo *DAXPY* usado nas tarefas de computação (à direita).

implementados com o objetivo de medir a oportunidade de sobreposição em outros padrões de comunicação.

Para medir a oportunidade de sobreposição, os testes implementam um determinado padrão de comunicação, e escalonam tarefas de computação buscando sobrepô-las com a comunicação, como descreve o algoritmo da Tabela 4.7.

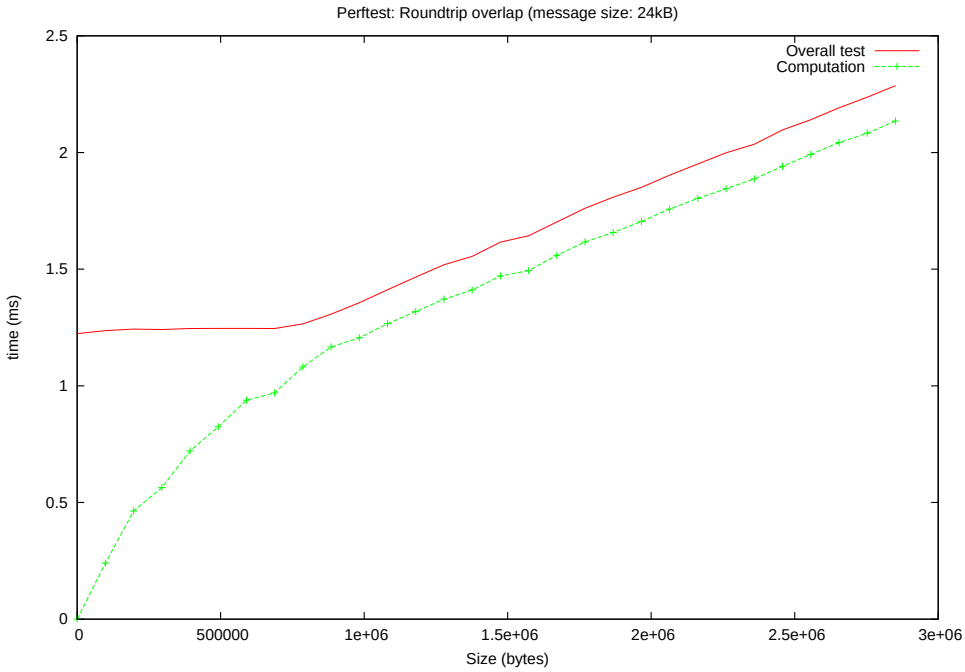


Figura 4.28: Resultados do teste de sobreposição no padrão ping-pong com mensagem de 24kB obtido com a biblioteca *OPENMPI* com protocolo *TCP*.

A quantidade de computação é incrementada gradativamente a cada execução do teste. Enquanto a computação estiver sobreposta com a comunicação, o tempo de execução do

algoritmo não é alterado com o aumento da quantidade de computação. A partir do ponto em que a quantidade de computação ultrapassa a oportunidade de sobreposição, o tempo do teste cresce proporcionalmente à quantidade de computação inserida, como pode ser observado no gráfico da Figura 4.28, que mostra o resultado do teste de sobreposição com padrão ping-pong para mensagens de 24kB. A coordenada do eixo X representa a quantidade de computação inserida na execução do teste, e a coordenada do eixo Y representa o tempo. O gráfico mostra duas linhas, uma representa o tempo total do teste, e a outra o tempo consumido apenas com a computação inserida. Inicialmente, o tempo total do teste não se altera significativamente com o incremento da quantidade de computação (parte horizontal da curva). Quando a quantidade de computação ultrapassa 688128 operações, que corresponde a 0.96 ms de computação, o tempo do teste passa a ser afetado pela quantidade de computação. A oportunidade de sobreposição nesta operação com mensagens de 24kB é de 0.96 ms.

A tarefa de computação inserida consiste em operações do tipo *DAXPY*, como descreve o algoritmo da Tabela 4.7 (à direita). A quantidade de computação corresponde ao número de iterações do algoritmo, que em cada iteração realiza a multiplicação de dois números de ponto flutuante com precisão dupla lidos de dois *buffer*s distintos. Para minimizar o efeito da memória *cache* os *buffer*s são alocados com tamanho mínimo de 64kB, e chamadas consecutivas da função percorrem blocos diferentes dos *buffer*s.

Os testes de sobreposição de comunicação com computação estão descritos a seguir, classificados de acordo com o padrão de comunicação empregado. O teste baseado em comunicação com padrão ping-pong faz parte da ferramenta Perftest. Os demais testes foram implementados como parte deste trabalho para avaliar a oportunidade de sobreposição em outros padrões de comunicação.

Sobreposição Ping-pong No teste de sobreposição com padrão ping-pong, o algoritmo é repetido sempre com o mesmo tamanho de mensagem, e gradativamente insere-se computação entre o envio e a recepção da mensagem, como representado na Tabela 4.7.

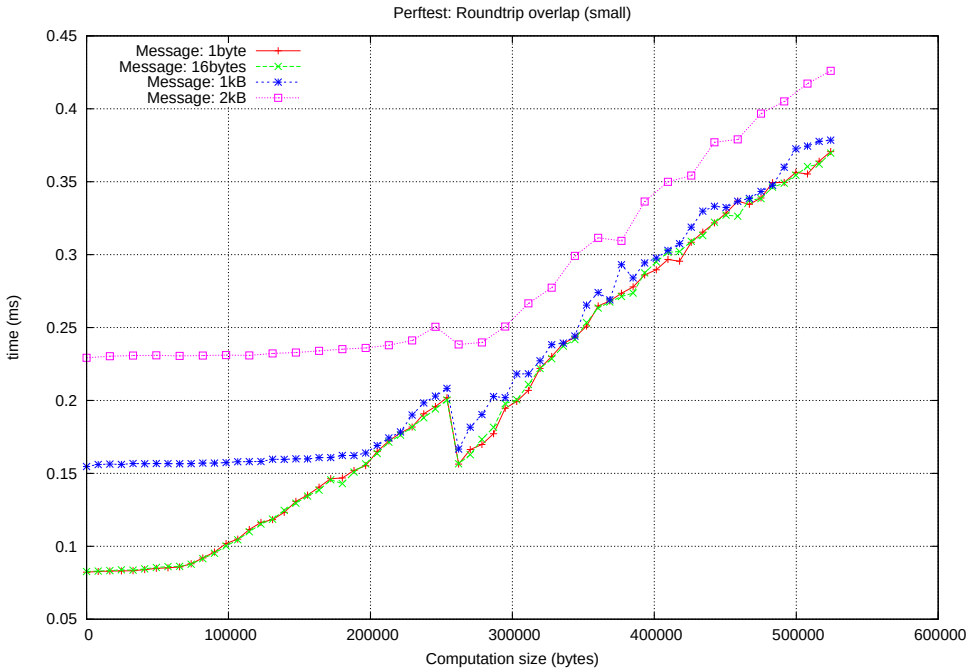


Figura 4.29: Sobreposição com padrão de comunicação ping-pong e mensagens entre 1 byte e 2kB. O eixo X representa a quantidade de computação.

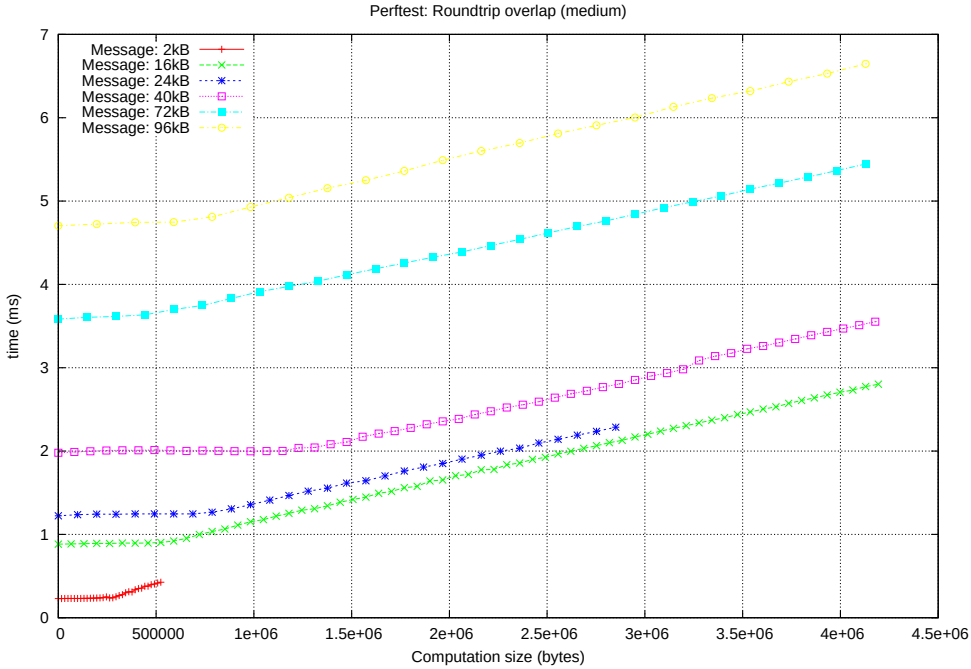


Figura 4.30: Sobreposição com padrão de comunicação ping-pong e mensagens entre 2kB e 96kB. O eixo X representa a quantidade de computação.

O teste de sobreposição implementado pelo Perftest foi alterado para que a computação seja escalonada apenas em um dos processos que participa do teste. Na versão original do teste, a computação é inserida nos dois processos, e o tempo do teste reportado pela

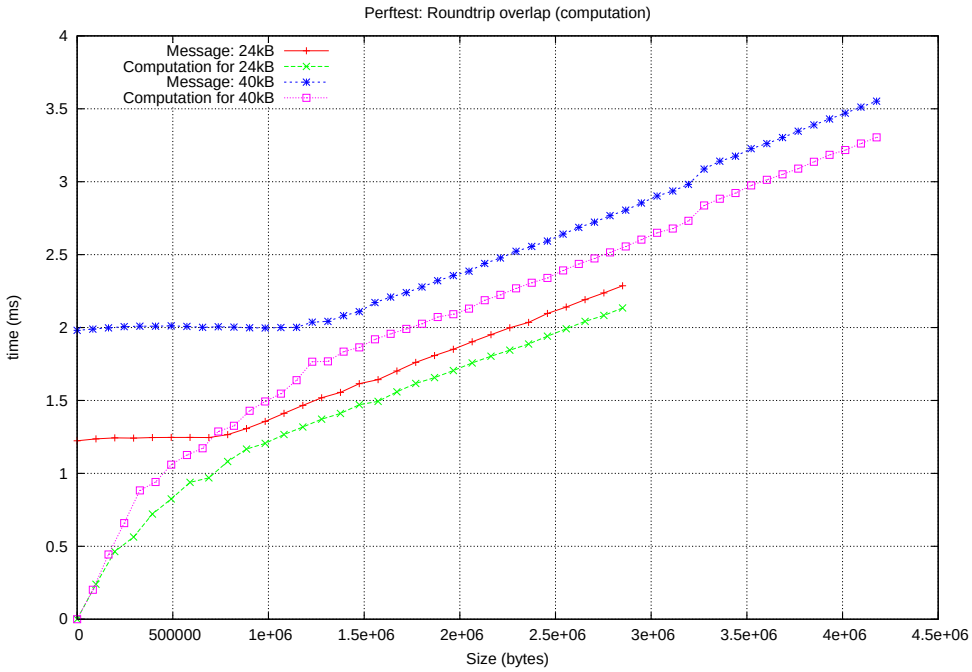


Figura 4.31: Sobreposição com padrão de comunicação ping-pong para mensagens de 24kB e 40kB e o tempo despendido na computação.

ferramenta é dividido por um fator de 2. O novo teste insere computação apenas no processo em que a primitiva de envio precede a de recebimento, e o tempo reportado corresponde ao tempo total do teste, tempo de ida e volta das mensagens.

O gráfico da Figura 4.29 mostra os resultados da execução do teste de sobreposição para mensagens pequenas, entre 1 byte e 2kB obtidos com a biblioteca *OPENMPI*. Os resultados para mensagens entre 2kB e 96kB são mostrados no gráfico da Figura 4.30, também obtidos com a biblioteca *OPENMPI*. Os dois gráficos mostram a quantidade de computação (eixo X) que pode ser sobreposta entre o envio da mensagem e a resposta do ping-pong, de acordo com o tamanho da mensagem. O tempo despendido na computação não pode ser diretamente inferido com base na quantidade, pois é influenciado pelo tamanho da mensagem. No gráfico da Figura 4.31 estão representados os tempos do teste para mensagens de 24kB e 40kB, além do tempo despendido apenas com a computação para cada tamanho de mensagem.

Sobreposição Local A sobreposição local, descrita na Seção 3.6, consiste no escalonamento de computação entre a solicitação de envio não-bloqueante e o término

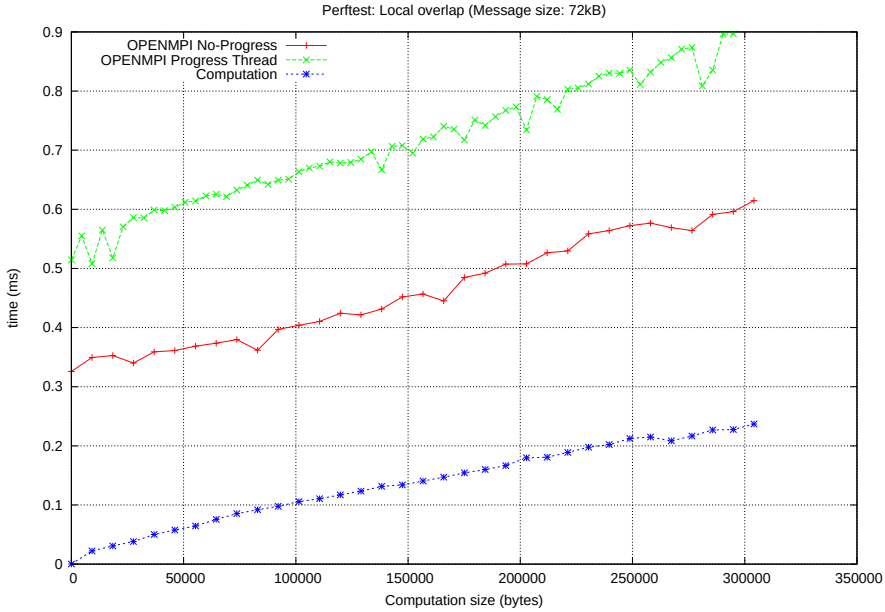


Figura 4.32: Teste de sobreposição local com e sem o *Thread* de progresso da biblioteca *OPENMPI* para mensagens de 72kB.

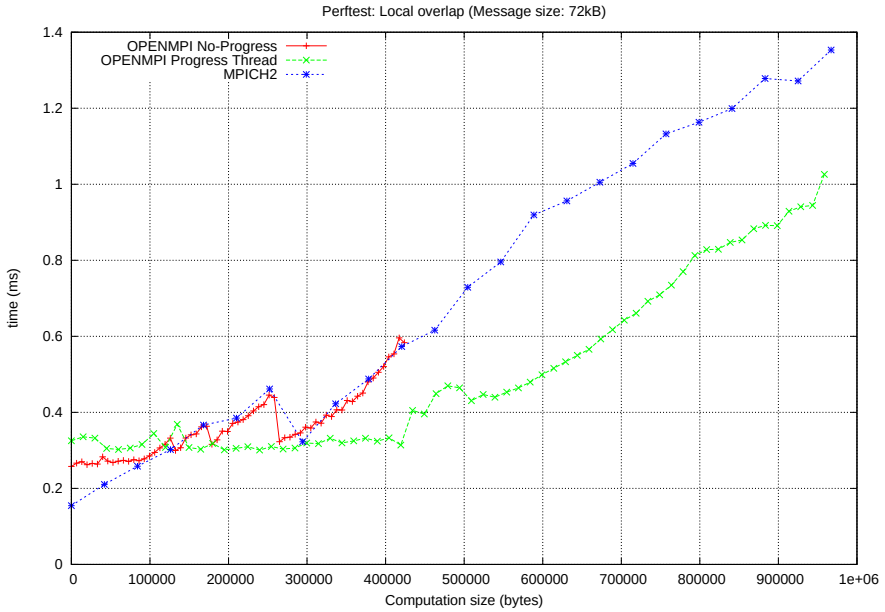
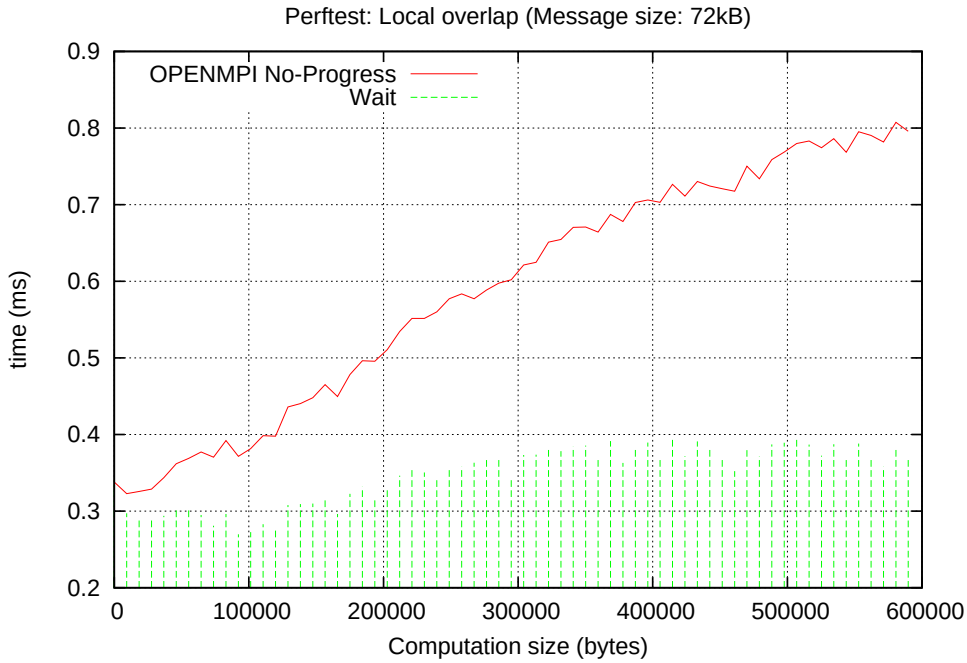


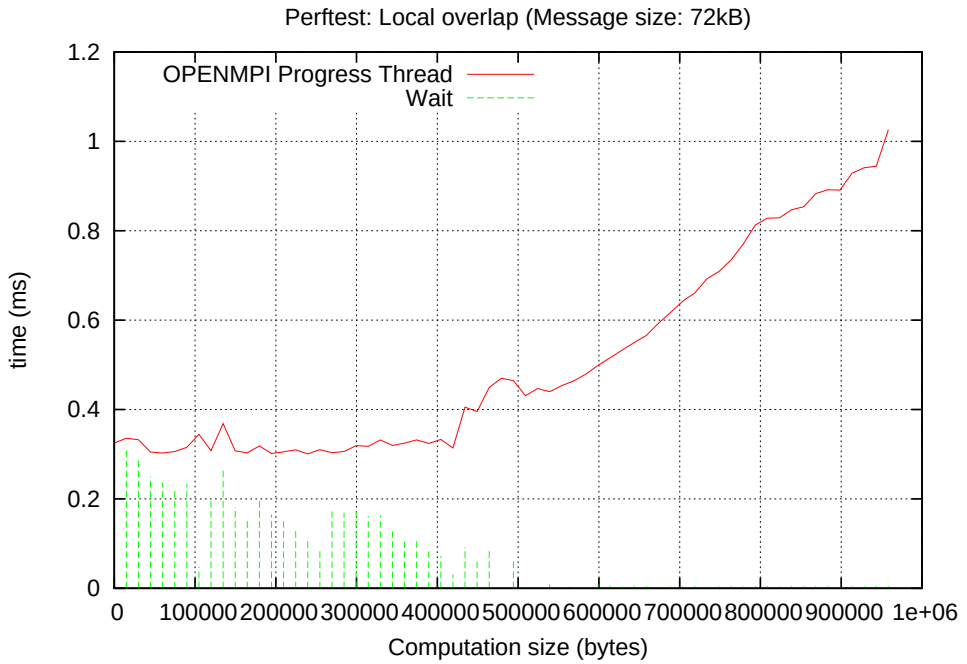
Figura 4.33: Teste de sobreposição local com e sem *Thread* de progresso realizado em um par de estações bi-processadas.

da operação. Foram implementados novos testes de sobreposição local com modo de comunicação síncrono e padrão, tendo como base a estrutura do Perftest.

Para que haja oportunidade de sobreposição local, o tempo para completar uma operação de envio não pode ser dominado pelo *overhead*. Além disso, a biblioteca de comunicação deve permitir o progresso da comunicação mesmo depois que o fluxo de



(a) Sem *Thread* de progresso



(b) Com *Thread* de progresso

Figura 4.34: *Teste de sobreposição local – em um ambiente que não apresenta oportunidade de sobreposição (a), e em um ambiente que apresenta oportunidade de sobreposição (b). Os gráficos não têm a mesma escala no eixo X.*

execução retornar para o programa.

A biblioteca *OPENMPI* oferece uma opção para adicionar suporte a um *thread* de progresso que deve ser definida ao compilar a biblioteca – e que não pode ser definida em

tempo de execução. Entretanto, o *Thread* de progresso implica em alguma sobrecarga de processamento como pode ser observado nas Figuras 4.9 (página 39) e 4.18 (página 49) – para mensagens longas transmitidas com protocolo de comunicação síncrono.

Os testes realizados no Nautilus mostram que, mesmo com o recurso do *Thread* de progresso, a comunicação não progride. O gráfico da Figura 4.32 compara os resultados do teste com e sem o *Thread* de progresso da biblioteca *OPENMPI* – com mensagens de 72kB. O gráfico mostra que a execução com o *Thread* de progresso apresenta maior sobrecarga de processamento, mas não expõe oportunidade de sobreposição.

Embora os testes no Nautilus não apresentem oportunidade de sobreposição local, outras configurações podem prover tal oportunidade. Para ilustrar o potencial da sobreposição local, o mesmo teste foi realizado em um par de estações bi-processadas do C3SL, também ligadas por rede Gigbit Ethernet. Neste par de máquinas há oportunidade de sobreposição quando o *Thread* de progresso é usado, como mostra o gráfico da Figura 4.33. O teste sem o *Thread* de progresso pode esconder aproximadamente 90 mil operações *DAXPY*, que neste teste custou 0.096 ms. Com o uso do *Thread* de progresso, o teste foi capaz de sobrepor aproximadamente 420 mil operações *DAXPY*, que neste teste consumiu 0.26 ms. Inicialmente, quando não há computação intercalada, o tempo despendido no envio da mensagem com o *Thread* de progresso é de 0.33 ms, enquanto que sem o *Thread* de progresso o teste consome 0.26 ms. Entretanto, para realizar o envio da mensagem e mais 420 mil operações *DAXPY* o tempo total sem o uso do *Thread* de progresso é da ordem de 0.60 ms, enquanto que com o uso do *Thread* de progresso o tempo mantém-se em aproximadamente 0.33 ms.

O gráfico da Figura 4.34(a) mostra o resultado obtido no Nautilus e o gráfico da Figura 4.34(b) o resultado obtido no par de estações bi-processadas. Além do tempo do teste, estes gráficos mostram o tempo despendido na execução da primitiva usada para completar a operação de envio. O teste realizado com o par de máquinas e com o *Thread* de progresso (b) mostra que há oportunidade de sobreposição, e o tempo despendido com a primitiva que completa a operação diminui gradativamente – mostrando que a comunicação progride mesmo quando o fluxo de execução retorna ao programa. Por

outro lado, o teste realizado no Nautilus (a) não mostra oportunidade de sobreposição, e o tempo despendido com a primitiva que completa a operação não diminui quando a quantidade de computação é incrementada – o que indica que quando o fluxo retorna para o programa a comunicação não progride, voltando a progredir apenas quando a primitiva para completar a operação é executada.

Primitiva bloqueante	Primitiva não-bloqueante

`for (int i=0; i < REPETICOES; i++){`
`  MPI_Send();`
`  OverlapComputation(len);`
`}`
	`for (int i=0; i < REPETICOES; i++){`
	`  MPI_Isend();`
	`  OverlapComputation(len);`
	`  MPI_Wait();`
	`}`

Tabela 4.8: Teste de sobreposição do *gap* com primitiva de envio bloqueante (à esquerda) e com primitiva de envio não-bloqueante (à direita).

Sobreposição do *gap* O *gap* representa a capacidade de enviar mensagens consecutivas de um processo a outro. Quando o *gap* não é dominado pelo *overhead* de processamento, há oportunidade de sobreposição de computação útil entre as chamadas de envio. Para medir a oportunidade de sobreposição, foi implementado um teste que gradativamente insere computação entre as chamadas de envio – como representa o algoritmo da Tabela 4.8. Quando a quantidade de computação excede a oportunidade de sobreposição, o tempo do teste cresce com o aumento da quantidade de computação.

O teste de sobreposição do *gap* também expõe o *overhead* de processamento, que pode ser representado por $g - C'$, em que C' corresponde à quantidade de computação sobreposta, ou escondida. Os resultados deste teste são apresentados em conjunto com a sobrecarga de processamento na Seção 4.2.3.6.

4.2.3.6 Sobrecarga de processamento

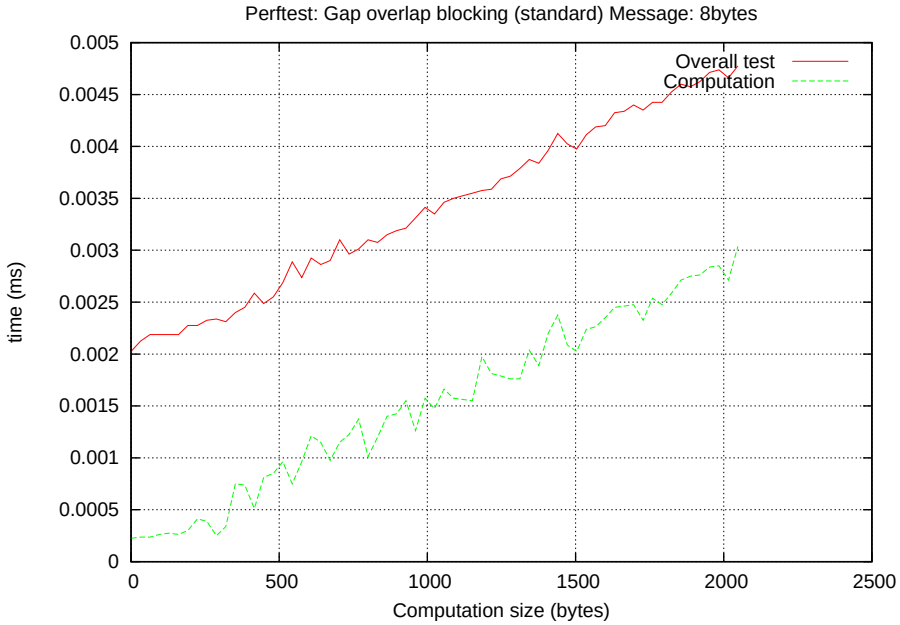
O teste de sobrecarga de processamento, ou *overhead*, foi implementado com base na estrutura do Perftest. Este teste é equivalente ao teste de sobreposição do *gap* com computação. O procedimento para obter a sobrecarga de processamento consiste em medir o *gap* e a quantidade de comunicação que pode ser escondida, ou sobreposta, entre

as chamadas consecutivas da primitiva de comunicação – como mostram os algoritmos da Tabela 4.8 – com primitiva de envio bloqueante (à esquerda) e com primitiva não-bloqueante (à direita). A sobrecarga de processamento corresponde à diferença entre o *gap* e a quantidade de computação escondida, ou sobreposta com a comunicação C', e pode ser modelada como $g - C' = Os$. Para medir a sobrecarga de recebimento Or é usado o mesmo algoritmo, com a primitiva de recebimento no lugar da primitiva de envio, e pode ser representada como $g - C' = Or$.

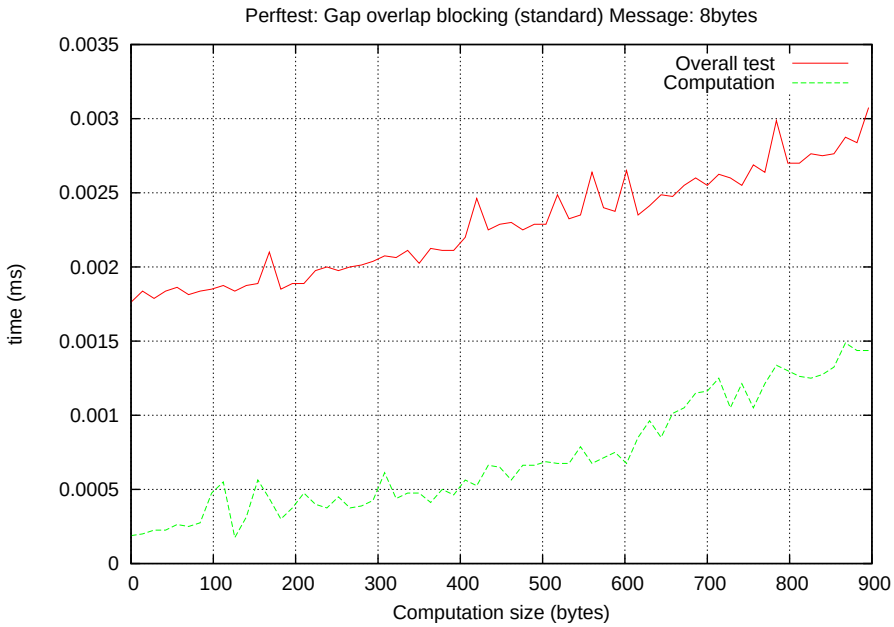
Os testes de sobrecarga e sobreposição do *gap* enviam muitas mensagens consecutivamente. Devido ao erro encontrado na biblioteca *OPENMPI* discutido anteriormente, os testes não puderam ser executados com todo conjunto de dados. Os testes com mensagens pequenas exigem um maior número de mensagens consecutivas para saturar a rede, e são os mais prejudicados. Em particular, os testes com o protocolo *LLC* mostraram-se mais suscetíveis ao problema encontrado no *OPENMPI*, e por este motivo não são apresentados resultados com o *LLC* para mensagens pequenas. Mesmo com um conjunto limitado de resultados válidos, estes são usados para demonstrar o uso dos testes implementados e obter uma avaliação, mesmo que parcial, do ambiente testado.

A Figura 4.35 mostra os resultados para mensagens de 8 bytes para o teste de sobreposição com primitiva de envio bloqueante e modo de comunicação padrão. Os resultados foram obtidos com a biblioteca *OPENMPI* com protocolo *TCP* (4.35(a)) e com a biblioteca *MPICH2* (4.35(b)). Nos dois casos, não houve sobreposição de computação – o tempo do teste é afetado mesmo para pequenas quantidades de computação. Isto indica que o *gap* é dominado pelo *overhead*. O *overhead* medido com a biblioteca *OPENMPI* é de 0.00202 ms, e com a biblioteca *MPICH2* é de 0.00176 ms. A medida obtida com o *LogP Benchmark* para mensagem de 8 bytes com a biblioteca *OPENMPI* é de 0.0017 – menor do que a medida obtida com o teste de sobreposição.

Os gráficos da Figura 4.36 mostram os resultados para mensagens de 56kB obtidos com as bibliotecas *OPENMPI* com protocolo *TCP* e MPICH2. A Tabela 4.9 lista as medidas dos parâmetros *gap*, *overhead* e a quantidade de computação sobreposta, além da medida obtida com o *LogP Benchmark* para comparação. Os resultados com modo



(a) *OPENMPI-TCP*

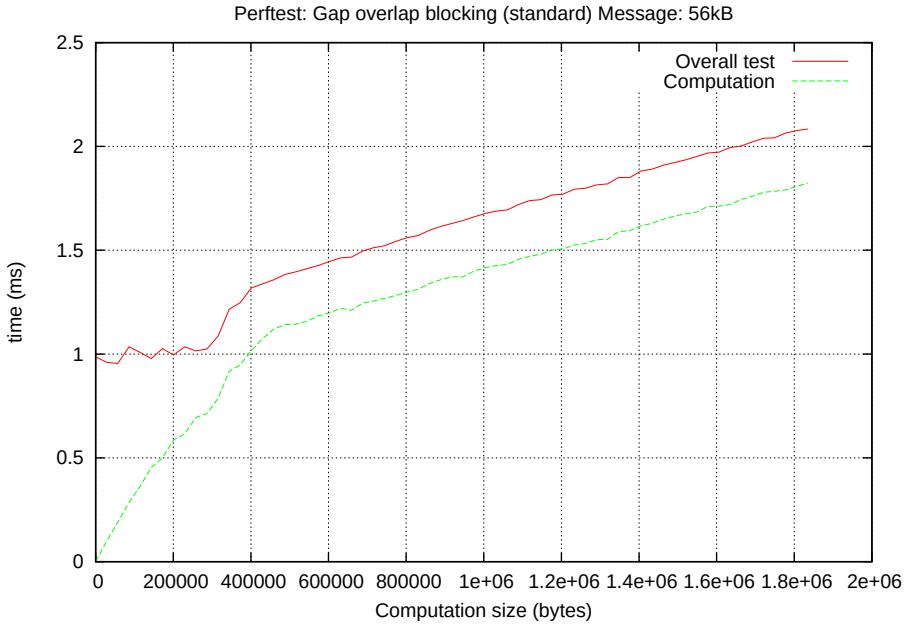


(b) *MPICH2*

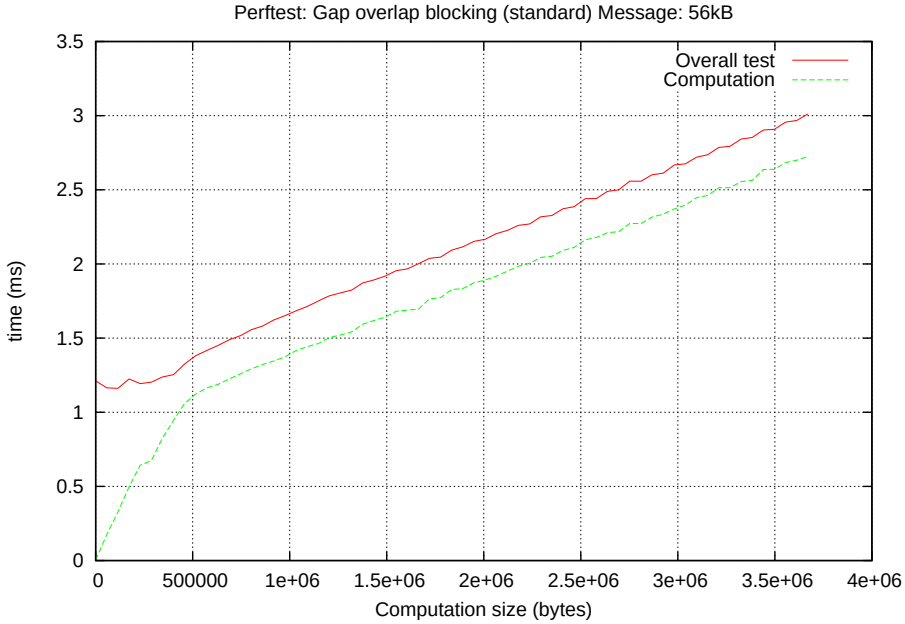
Figura 4.35: *Teste de sobrecarga e sobreposição do gap no envio de mensagens com 8 bytes com primitiva de envio bloqueante e modo de comunicação padrão.*

	gap	C'	Os
OPENMPI-*TCP*	0.98	0.7131	0.2668
MPICH2	1.2124	0.9462	0.662
LogP Bench	1.3155	–	0.1088

Tabela 4.9: Sobrecarga no envio de mensagens de 56kB com biblioteca *OPENMPI (TCP)* e *MPICH2*. A coluna C' representa o tempo consumido com a computação sobreposta.



(a) *OPENMPI* (TCP)



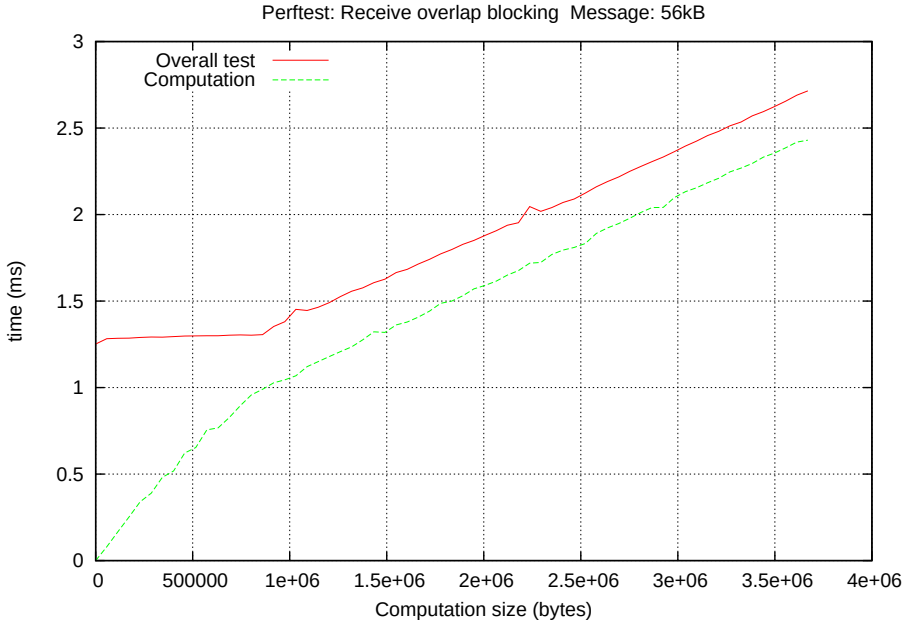
(b) MPICH2

Figura 4.36: *Teste de sobrecarga e sobreposição do gap no envio de mensagens com 56kB – e com primitiva de envio bloqueante e modo de comunicação padrão.*

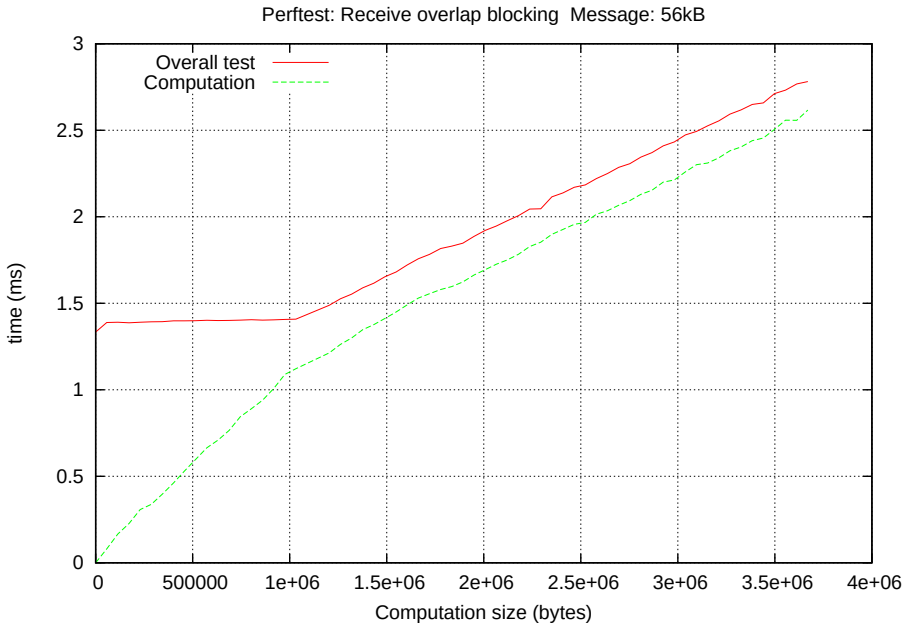
de comunicação padrão são equivalentes aos obtidos com modo de comunicação síncrono (0.2668 ms) – com mensagens de 56kB.

Para mensagens de 56kB, o uso do protocolo *LLC* como camada de transporte do *OPENMPI* apresentou *overhead* de envio maior que o observado com o uso do protocolo TCP. Em tese, o protocolo *LLC* requer menos processamento que o protocolo TCP/IP.

Entretanto, o teste mede o *overhead* exposto para a aplicação – o que depende da implementação do *MPI* e de sua integração como o *LLC*.



(a) *OPENMPI* (TCP)



(b) MPICH2

Figura 4.37: *Teste de sobrecarga no recebimento de mensagens com 56kB – e com primitiva de envio bloqueante e modo de comunicação síncrono.*

Os resultados dos testes de sobrecarga de recebimento são apresentados na Figura 4.37, e foram obtidos com a biblioteca *OPENMPI* com *TCP* (a) e com a biblioteca *MPICH2* (b). A Tabela 4.10 mostra as medidas do *overhead* de recebimento (Or) para mensagens

	T	C'	Or
OPENMPI-*TCP*	1.303	0.95775	0.34525
MPICH2	1.4067	1.0883	0.3184
LogP Bench		–	0.0819

Tabela 4.10: Sobrecarga de recepção em mensagens de 56kB com biblioteca *OPENMPI-TCP* e *MPICH2*. A coluna C' representa o tempo consumido com a computação sobreposta, e T o tempo do teste.

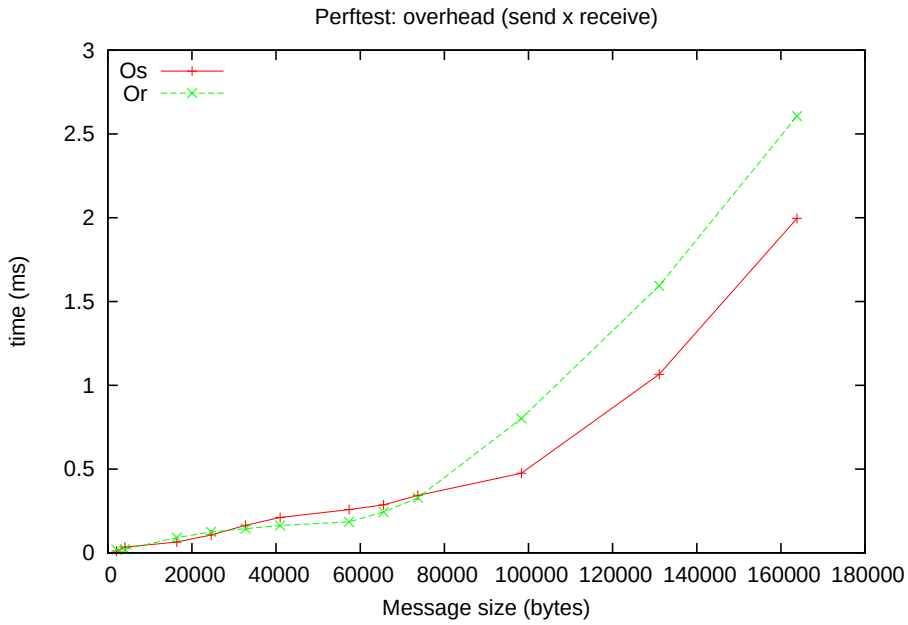
de 56kB. No caso do *overhead* de recebimento, nota-se que a medida obtida pelo *LogP Benchmark* (0.0819 ms) é muito menor que a obtida com o teste de sobreposição (0.34525 ms) – diferença atribuida ao modo com que a medida é realizada, como discutido na Seção 4.2.2.

Os gráficos da Figura 4.38 mostram os resultados do teste de sobrecarga de envio (Os) e de recepção (Or) obtidos no Nautilus com a biblioteca *OPENMPI* (a) e em um par de estações do *C3SL* com a biblioteca *MPICH2* (b). Note que o *overhead* de processamento não é afetado significativamente pela transição entre mensagens curtas e longas, que para o *MPICH2* ocorre em 16kB e para o *OPENMPI* em 64kB. Nos dois casos, o *overhead* de recepção torna-se maior que o de envio para mensagens a partir de 96kB.

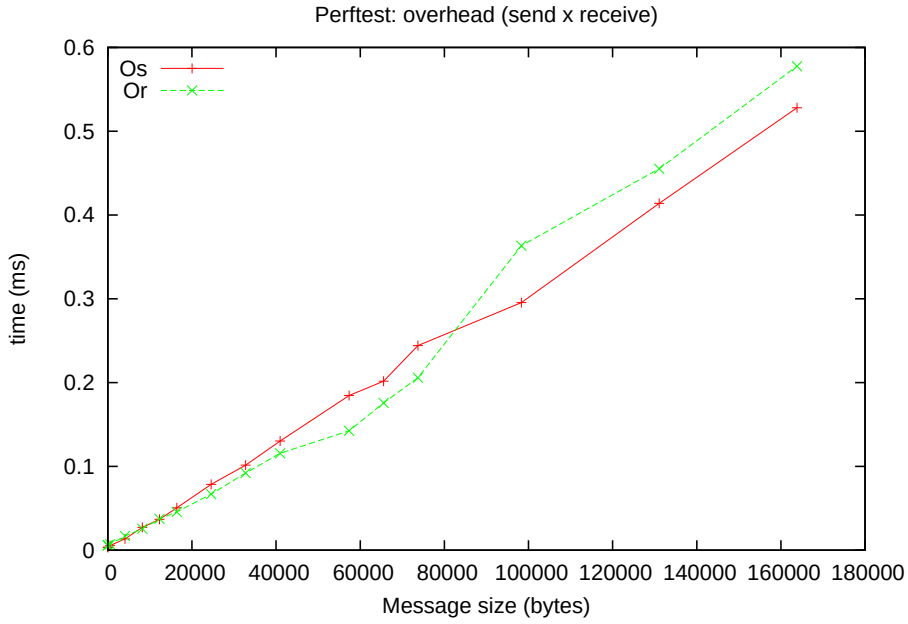
Neste teste é considerado o tempo de processamento envolvido na comunicação, e que não está disponível para processamento da aplicação. O teste baseado na sobreposição de computação mede de forma mais realista a sobrecarga de processamento, quando comparado aos resultados obtidos com o *LogP Benchmark*.

4.3 Avaliação com Kernels Científicos

Os *kernels FFT* e *Radix Sort* foram empregados na avaliação de desempenho, em particular na avaliação de técnicas de sobreposição da comunicação com computação. Foram usadas implementações *multi-threaded* e outras baseadas em primitivas não-bloqueantes para promover a sobreposição da comunicação, e assim melhorar o desempenho. A seguir os *kernels* são descritos e os resultados apresentados.



(a) Nautilus



(b) C3SL

Figura 4.38: *Resultados do teste de sobrecarga de processamento obtidos no Nautilus com a biblioteca OPENMPI (a) e com um par de estações do C3SL com a biblioteca MPICH2 (b).*

4.3.1 FFT

Esta Seção descreve o *kernel* FFT [3], bem como diferentes implementações do algoritmo usadas na avaliação de desempenho. A implementação identificada como *original* foi usada como base para o trabalho desenvolvido em [13], que descreve uma nova implementação

com uma abordagem chamada linha-a-linha, e uma implementação *multi-threaded* – que busca sobrepor comunicação com computação.

Neste trabalho foi desenvolvida uma nova versão que faz uso de primitivas não-bloqueantes para promover sobreposição de comunicação com computação baseado na implementação linha-a-linha desenvolvida em [13]. Os resultados apresentados nesta Seção mostram que a abordagem linha-a-linha apresentou queda no desempenho em relação à implementação original para os testes com problema de tamanho menor, tais como 2^{16}. Entretanto, com o crescimento do tamanho do problema, o algoritmo mostrou-se eficiente, e as implementações que promovem sobreposição de computação com comunicação alcançaram os melhores resultados. A implementação desenvolvida neste trabalho baseada em primitivas não-bloqueantes apresentou o melhor desempenho na grande maioria dos testes realizados.

4.3.1.1 Descrição

O *kernel FFT* implementa o algoritmo *Fast Fourier Transform* [3]. Este algoritmo faz parte de algumas aplicações como processamento de sinais, reconhecimento de voz e dinâmica de fluidos.

O algoritmo tem como entrada uma matriz de n pontos complexos que será transformada, uma matriz com as n raízes complexas do número real 1 que é chamada de *raízes da unidade* e uma estrutura intermediária também com n pontos complexos. O algoritmo distribui blocos da matriz entre os processos como representado na Figura 4.39.

No início do algoritmo a matriz de entrada é transposta na estrutura intermediária. No segundo passo são realizadas FFTs unidimensionais nas linhas da matriz intermediária. No terceiro passo os elementos da matriz raízes da unidade são aplicados aos elementos da matriz intermediária das posições correspondentes. No quarto passo a matriz intermediária é transposta na matriz de entrada. No quinto passo são realizadas FFTs unidimensionais na matriz de entrada, e no sexto e último passo a matriz é transposta na matriz intermediária.

Na fase de transposição da matriz há $P - 1$ blocos que são transmitidos para outros

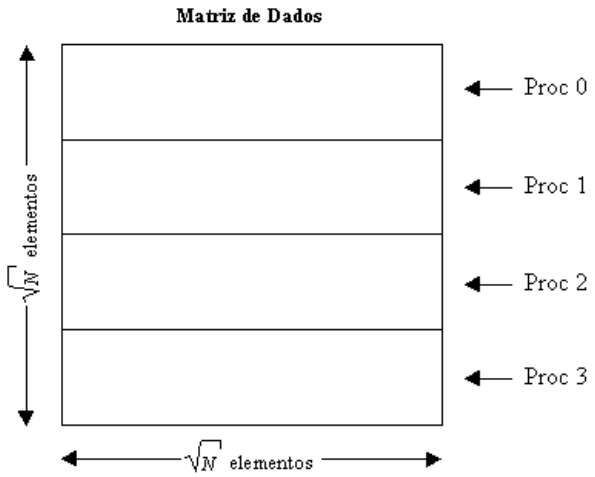


Figura 4.39: Distribuição do conjunto de dados no *kernel* FFT.

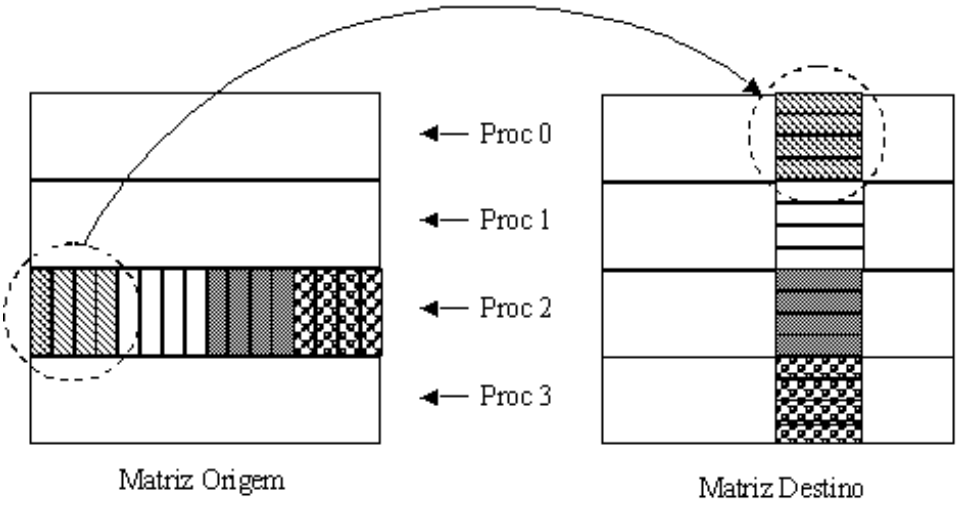


Figura 4.40: Transposição da matriz.

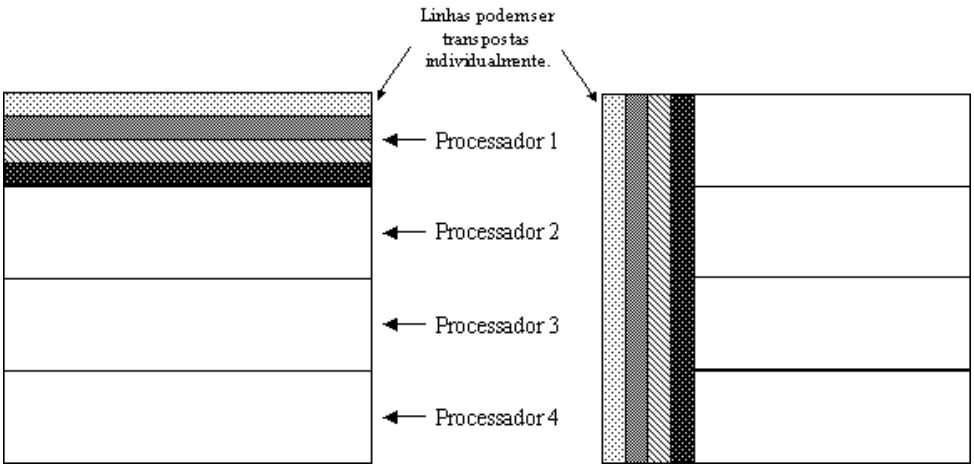


Figura 4.41: Transposição linha-a-linha da matriz.

processos, e um bloco que é transposto localmente. Na transposição cada processo troca blocos da matriz com todos os outros – padrão de comunicação todos-para-todos. A Figura 4.40 descreve o passo de transposição da matriz.

original	linha-a-linha

`Transpose();`
`FFT1DOnce();`
`TwiddleOneCol();`
 	
`Transpose();`
`FFT1Donce();`
 	
`Transpose();`
	`for (linha = 0; linha < linhas_por_processo; linha++){`
	`  Transpose(linha);`
	`  FFT1DOnce(linha);`
	`  TwiddleOneCol(linha);`
	`}`
	`for (linha = 0; linha < linhas_por_processo; linha++){`
	`  Transpose(linha);`
	`  FFT1Donce(linha);`
	`}`
	`for (linha = 0; linha < linhas_por_processo; linha++){`
	`  Transpose(linha);`
	`}`

Tabela 4.11: Passos do algoritmo *FFT* na versão original (à esquerda) e na implementação linha-a-linha (à direita).

Na implementação do algoritmo, a função que realiza a transposição é chamada de *Transpose*. As FFTs unidimensionais são realizadas pela função *FFT1DOnce*, e a aplicação das matriz raízes da unidade é realizada pela função *FFTTwiddleOneCol*.

As Figuras 4.39, 4.40 e 4.41 foram retiradas de [13].

Implementação linha-a-linha Em [13] os autores propõem uma reestruturação do algoritmo na qual as linhas da matriz são transpostas individualmente, e após a

transposição de cada linha são aplicadas as FFTs unidimensionais – e as raízes da unidade dependendo do estágio do algoritmo. A Figura 4.41 mostra um esquema da transposição linha-a-linha, e a Tabela 4.11 mostra os passos do algoritmo original e a implementação linha-a-linha desenvolvida pelos autores.

Implementação multi-threaded Com a nova estrutura que faz a transposição linha-a-linha pode-se sobrepor fases de comunicação (transposição da matriz) com fases de computação (aplicação das FFTs unidimensionais e raízes da unidade). A fase de computação (FFT1DOnce e TwiddleOneCol) pode ser iniciada após a transposição da linha, em paralelo com a transposição das linhas subsequentes. Em [13] os autores descrevem a implementação de uma versão *multi-threaded* do algoritmo que executa concorrentemente as fases de computação e comunicação. Um *thread* é responsável pela transposição da matriz, e outro pela aplicação das FFTs unidimensionais e raizes da unidade. Os *threads* são sincronizados para que a fase de computação que incide sobre uma linha inicie apenas depois que esta estiver transposta – já na posse do processo.

Implementação com primitiva não-bloqueante Neste trabalho foi criada uma nova versão do algoritmo que faz uso de primitivas não-bloqueantes para promover a sobreposição das fases de comunicação e computação. Esta nova implementação segue o mesmo algoritmo descrito na Tabela 4.11. Ao invés de usar *threads* separados para as fases de computação e comunicação, a função Transpose foi modificada para não bloquear o fluxo de execução, e apenas iniciar a comunicação da linha. O algoritmo solicita a transposição da linha $i + 1$ e continua com a fase de computação sobre a linha i, como descrito na Tabela 4.12.

4.3.1.2 Avaliação de Desempenho

Os resultados apresentados a seguir foram obtidos no conjunto A de estações do Nautilus. Os testes foram executados com 2, 4 e 8 processos. O tamanho do problema N também foi escalado. Considerando $N = 2^m$, foram executados testes com os seguintes valores de m: 16, 18, 20, 22 *e* 24.

não-bloqueante

```
Transpose(linha=0);
for (linha = 0; linha < linhas_por_processo; linha++){
   Transpose( linha +1 );

   //aguarda transposição da linha
   Wait(linha);
   FFT1DOnce(linha);
   TwiddleOneCol(linha);
}
Wait(linha);

Transpose(linha=0);
for (linha = 0; linha < linhas_por_processo; linha++){
   Transpose( linha + 1);

   //aguarda transposicao da linha
   Wait(linha);
   FFT1Donce(linha);
}
for (linha = 0; linha < linhas_por_processo; linha++){
   Transpose(linha);
}
Wait(linha);
```

Tabela 4.12: Algoritmo da implementação baseada em primitivas não-bloqueantes de comunicação.

O algoritmo do *FFT* faz uso intensivo de comunicação, que representa a maior parte do tempo consumido. Além disso, o algoritmo faz uso de mensagens grandes, e portanto o desempenho é fortemente influenciado pela largura de banda do sistema. Nos testes de largura de banda, a biblioteca *OPENMPI* com protocolo *LLC* apresentou os melhores resultados. A Figura 4.43 mostra o desempenho obtido com diferentes bibliotecas, em que a biblioteca *OPENMPI* com protocolo *LLC* apresentou o melhor desempenho. No teste com entrada de tamanho 2^{20} e com 8 processadores, o protocolo *LLC* apresentou ganho da ordem de 10% em relação ao TCP (também com OPENMPI).

As mensagens trocadas entre os processos durante a fase de transposição têm sempre o mesmo tamanho, que depende do tamanho da entrada e do número de processos envolvidos. O tamanho das mensagens usadas pelos algoritmos que implementam a transposição linha-a-linha pode ser calculado por $2 * (\frac{\sqrt{N}}{P}) * 8$, em que N é o tamanho do problema, P o número de processos e 8 corresponde ao tamanho em bytes de cada número

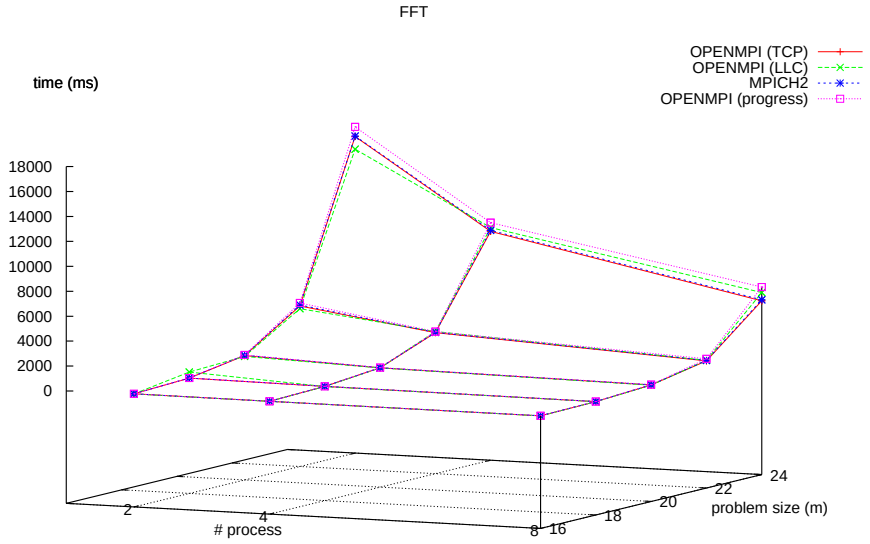


Figura 4.42: Resultados do *FFT* com as bibliotecas OPENMPI-TCP, OPENMPI-LLC, *OPENMPI* com *Thread* de progresso e *MPICH2*.

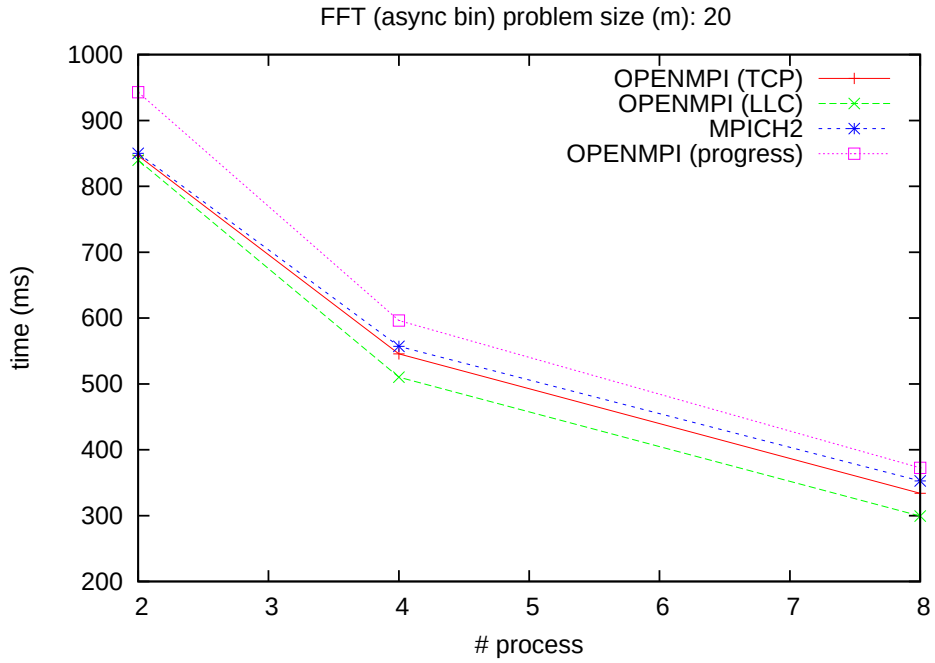


Figura 4.43: Resultado do *FFT* versão linha-a-linha com primitiva não-bloqueante e problema de tamanho 2^{20} números complexos.

complexo da entrada. No algoritmo original, em que é realizada a transposição completa da matriz, as mensagens são maiores do que na abordagem linha-a-linha, e o tamanho das mensagens é dado por $(\frac{\sqrt{N}}{P}) * 2 * (\frac{\sqrt{N}}{P}) * 8$. A Tabela 4.13 lista os tamanhos das mensagens

usadas na transposição de acordo com o tamanho da entrada 2^m e o número de processos envolvidos – tanto para os algoritmos de transposição linha-a-linha quanto para os que implementam a transposição da matriz completa.

	linha-a-linha			completa		
$m \backslash P$	2	4	8	2	4	8
16	2k	1k	512B	256k	64k	16k
18	4k	2k	1k	1M	256k	64k
20	8k	4k	2k	4M	1M	256k
22	16k	8k	4k	16M	4M	1M
24	32k	16k	8k	64M	16M	4M

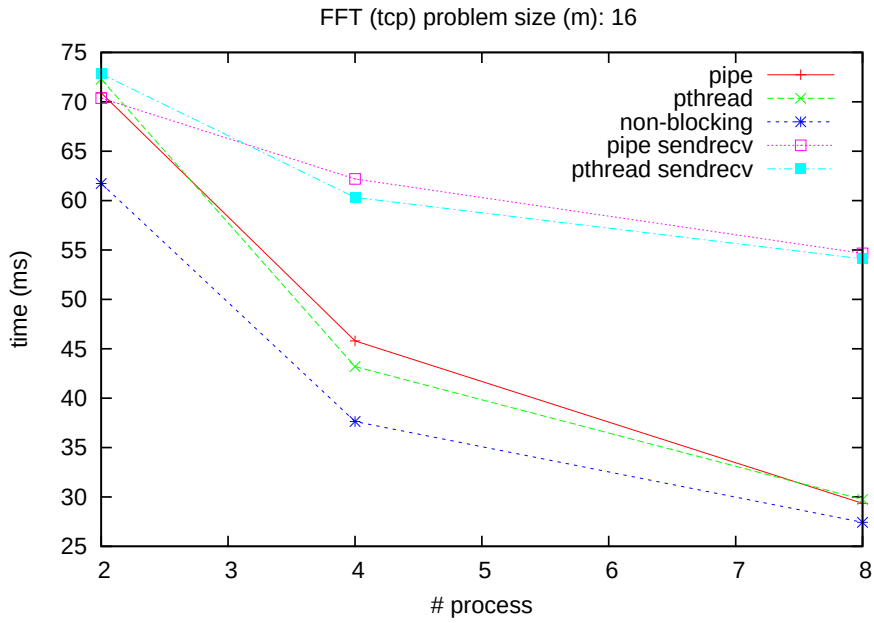
Tabela 4.13: *Tamanho das mensagens (em bytes) usadas no algoritmo do FFT de acordo com o tamanho da entrada* (2^m).

O gráfico da Figura 4.42 mostra os resultados dos *FFT* escalando o número de processadores (eixo X) e também o tamanho da entrada (eixo Y). Pode-se observar que quanto maior o conjunto de dados, maior o ganho obtido com o acréscimo do número de processadores. A Tabela 4.14 mostra os tempos de execução do *FFT* e o percentual consumido nas fases de computação para o algoritmo com transposição linha-a-linha (pipe). Pode-se notar que quanto maior o tamanho do problema, maior é o ganho alcançado com o acréscimo do número de processadores. Com uma entrada de tamanho 2^{16}, o ganho ao passar de 2 para 8 processadores é de 58.5%, enquanto que com entrada de tamanho 2^{24} o ganho é de 69.8%. O tempo consumido com as fases de computação representa menos da metade do tempo total para o conjunto de testes apresentado. A fração do tempo despendido com computação diminui quando se aumenta o número de processadores. Por outro lado, quando o tamanho da entrada cresce a fração do tempo

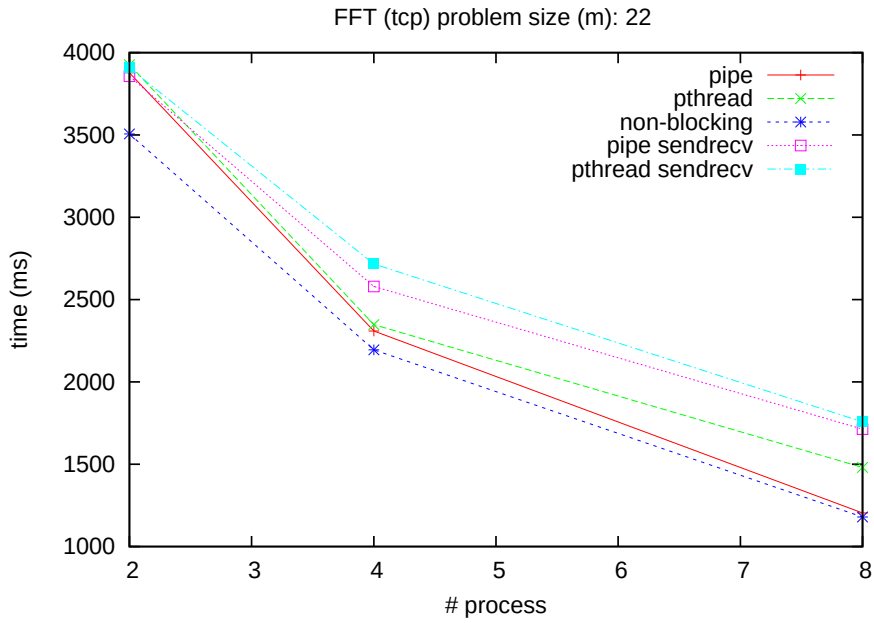
$m \backslash P$	2		4		8		ganho ($2 \Rightarrow 8$)
	T	C	T	C	T	C	
16	70.90	21%	45.79	16%	29.35	14%	58.5%
20	976.76	30%	574.23	25%	349.01	22%	64.2%
24	16378.45	33%	9347.03	29%	4930.22	28%	69.8%

Tabela 4.14: Tempo de execução do *FFT* em *ms* obtidos com a biblioteca *OPENMPI* (TCP) com a implementação linha-a-linha (pipe), com entrada de tamanho 2^{16}, 2^{20} *e* 2^{24}. As colunas identificadas por T mostram o tempo total do teste, e as colunas identificadas com C o percentual do tempo total consumido nas fases de computação. A coluna da direita mostra o ganho relativo ao passar de 2 para 8 processadores.

consumido com computação também aumenta.



(a) $m = 16$



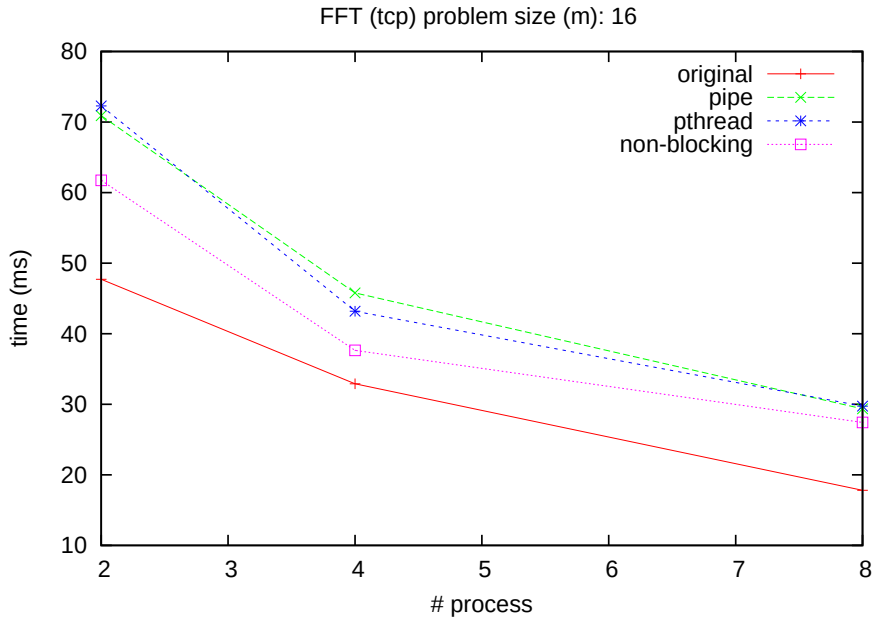
(b) $m = 22$

Figura 4.44: *Comparação do desempenho do algoritmo de transposição com primitiva de envio e recebimento conjugadas obtidos com a biblioteca OPENMPI (TCP) e com entrada de tamanho* 2^{16} *(a) e* 2^{22} *(b).*

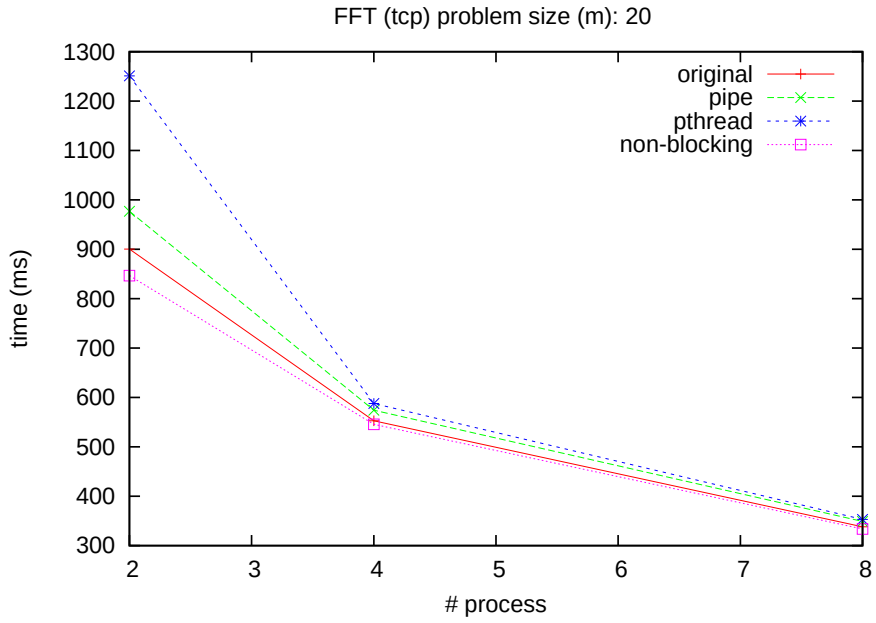
Primitivas de envio e recebimento conjugadas A implementação descrita em [13] emprega primitivas de envio e recebimento conjugadas (MPI_Sendrecv) na fase de transposição da matriz. O algoritmo de transposição segue o padrão

todos-para-todos, que foi avaliado na Seção 4.2.3. Como pode ser verificado no gráfico da Figura 4.27 (página 59), o desempenho do algoritmo com padrão todos-para-todos baseado em primitivas de envio e recepção conjugadas é pior do que as demais implementações testadas. A implementação foi alterada para usar primitivas não conjugadas na fase de transposição da matriz. Com o uso de primitivas não conjugadas, pode-se fazer uso de primitivas não-bloqueantes, e assim buscar sobreposição da comunicação com computação. Os gráficos da Figura 4.44 mostram os resultados do *FFT* com diferentes implementações do algoritmo de transposição – com e sem o uso de primitivas conjugadas. Nota-se que em todos os algoritmos o desempenho foi melhorado com a substituição da primitiva conjugada. Além do ganho de desempenho obtido com a substituição direta da primitiva, a versão baseada em primitiva não-bloqueante teve um desempenho ainda melhor. Note que na avaliação do teste com padrão todos-para-todos o desempenho das implementações com primitiva bloqueante e não-bloqueante é equivalente. Entretanto, a implementação do *FFT* baseada em primitiva não-bloqueante foi capaz de coordenar a comunicação de forma mais eficiente – e assim obter melhor desempenho. No teste com entrada de tamanho 2^{16}, o ganho obtido com a substituição direta da primitiva conjugada por primitivas bloqueantes é de 0.7%, 26% e 46%, para 2, 4 e 8 processos respectivamente. O ganho de desempenho com o uso de primitivas não-bloqueantes em relação a implementação com primitivas bloqueantes é de 14%, 21% e 7% para 2, 4 e 8 processos respectivamente.

Abordagem linha-a-linha A versão original tem melhor desempenho com entrada de tamanho 2^{16}, como pode ser observado na Figura 4.45(a). Com entradas de tamanho maior, tais como 2^{22} e 2^{24}, o desempenho piora em relação às outras implementações (que realizam a transposição linha-a-linha) – como mostram os gráficos das Figuras 4.46(a) e 4.46(b). Os resultados dos testes mostrados na Seção 4.2 indicam que quanto maior o tamanho da mensagem, maior é a largura de banda atingida. Isto beneficia a implementação original, que usa mensagens maiores na fase de transposição. Entretanto, o ganho com o aumento do tamanho da mensagem é maior para mensagens pequenas. Assim, quanto maior o tamanho das mensagens usadas pelo algoritmo, menor



(a) $m = 16$



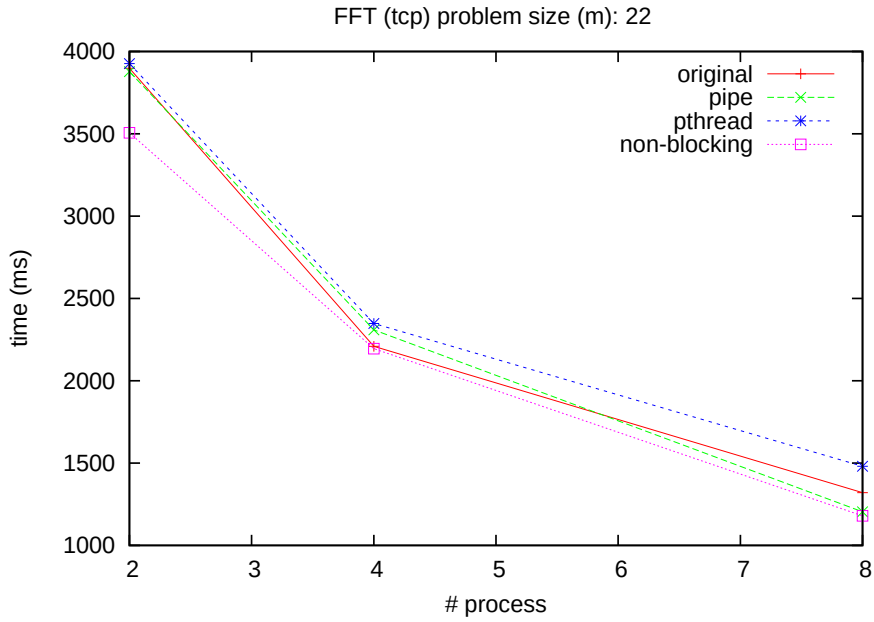
(b) $m = 20$

Figura 4.45: *Resultados da execução do FFT com variação do número de processadores obtidos com a biblioteca OPENMPI (TCP) para entrada de tamanho* 2^{16} *(a) e* 2^{20} *(b).*

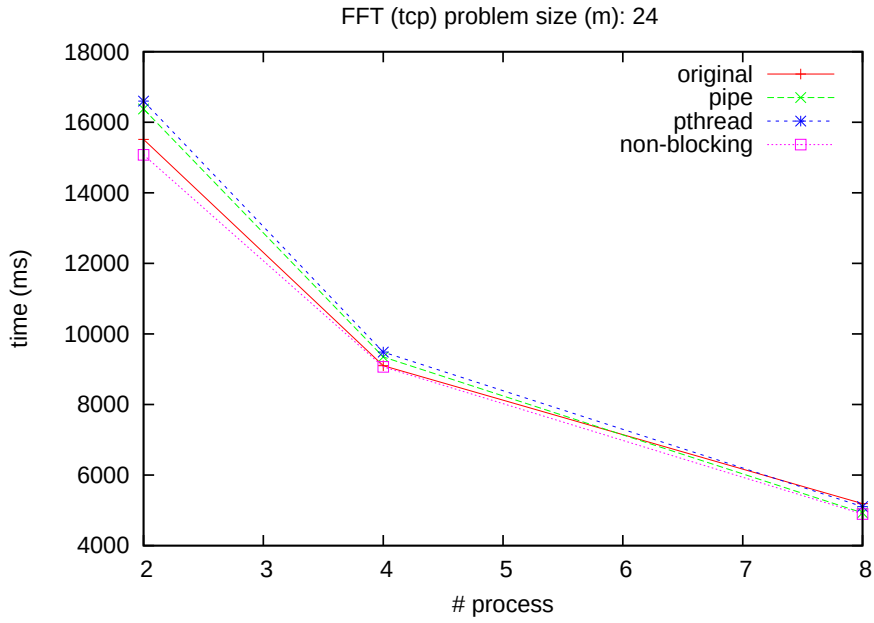
o ganho relativo da implementação original em relação às demais.

Implementações com primitiva não-bloqueante e multi-threaded

Das implementações com transposição linha-a-linha, a versão com primitivas não-bloqueantes apresentou o melhor desempenho na grande maioria dos casos. Tanto a



(a) $m = 22$

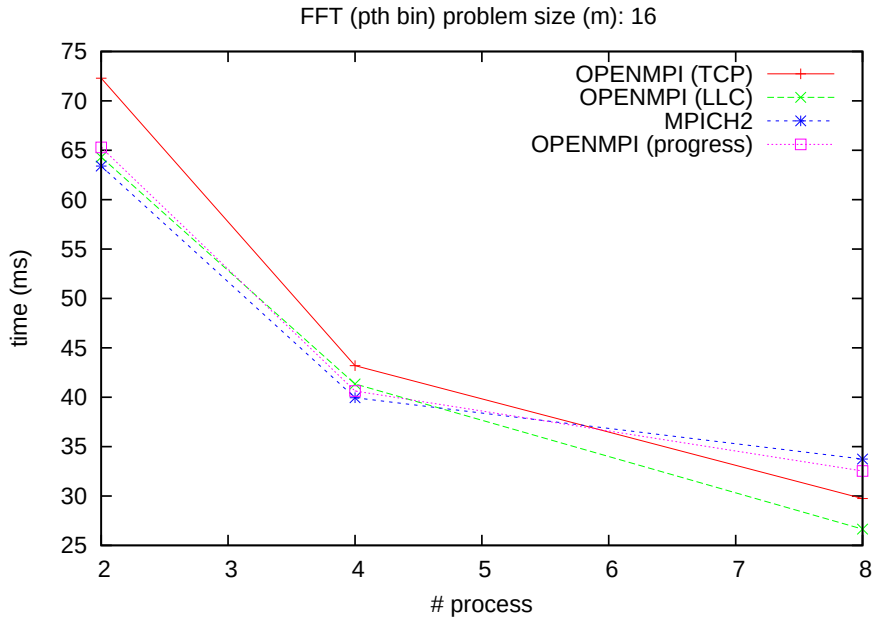


(b) $m = 24$

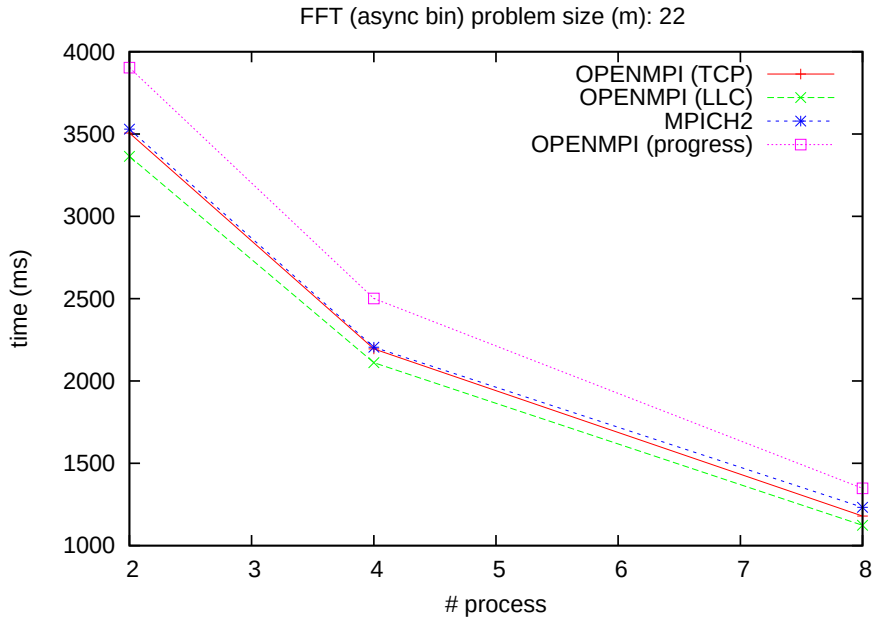
Figura 4.46: *Resultados da execução do FFT com variação do número de processadores obtidos com a biblioteca OPENMPI (TCP) para entrada de tamanho* 2^{22} *(a) e* 2^{24} *(b).*

implementação com primitivas não-bloqueantes, quanto a implementação *multi-threaded* buscam sobrepor comunicação com computação. A Figura 4.47 compara o desempenho da implementação *multi-threaded* e com primitiva não-bloqueante com diferentes bibliotecas.

Os resultados mostram que estas implementações apresentaram ganho de desempenho em relação à implementação linha-a-linha que não promove sobreposição. A Figura 4.48



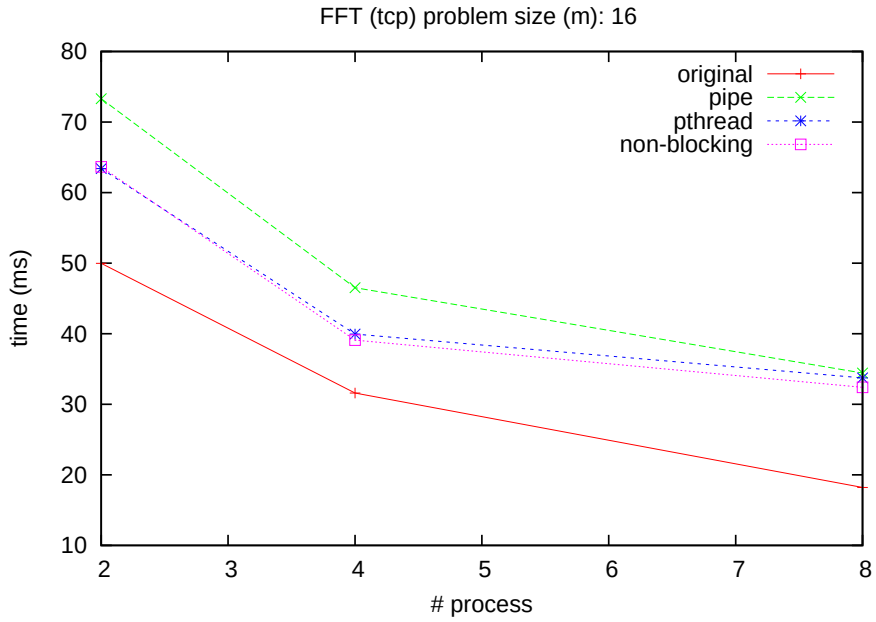
(a) $m = 16$



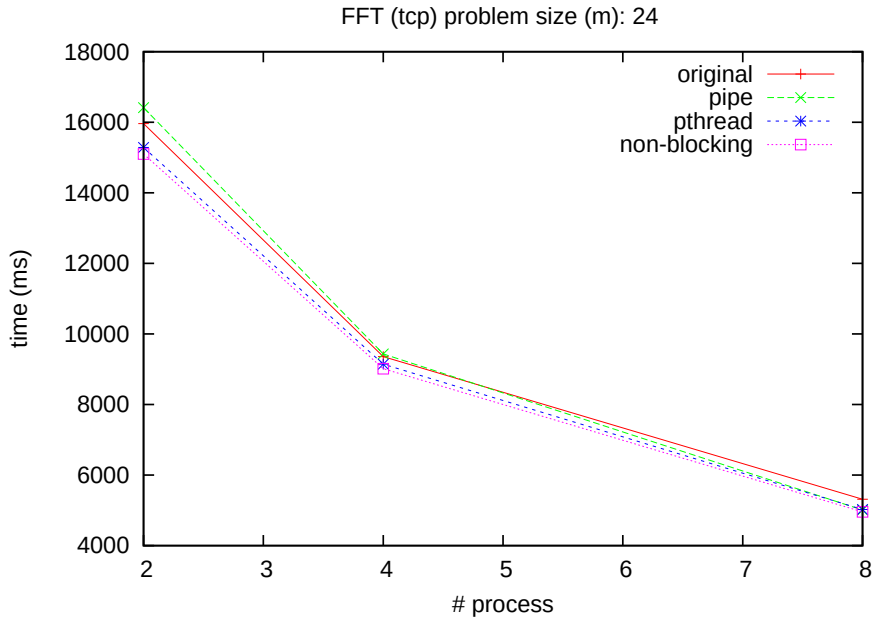
(b) $m = 22$

Figura 4.47: *Comparação dos resultados da implementação do FFT multi-threaded (a) e baseado em primitiva não-bloqueante (b) obtidos com as bibliotecas OPENMPI-TCP, OPENMPI-LLC, OPENMPI com Thread de progresso e MPICH2 – com entrada de tamanho* 2^{16} *(a) e* 2^{22} *(b).*

mostra os resultados obtidos com a biblioteca *MPICH2* com entradas de tamanho 2^{16} e 2^{24}. Pode-se notar que na execução com *MPICH2*, o desempenho da implementação com *Thread* ficou próximo ao da implementação com primitiva não-bloqueante. No teste realizado no *C3SL*, a implementação *multi-threaded* também apresentou desempenho



(a) $m = 16$

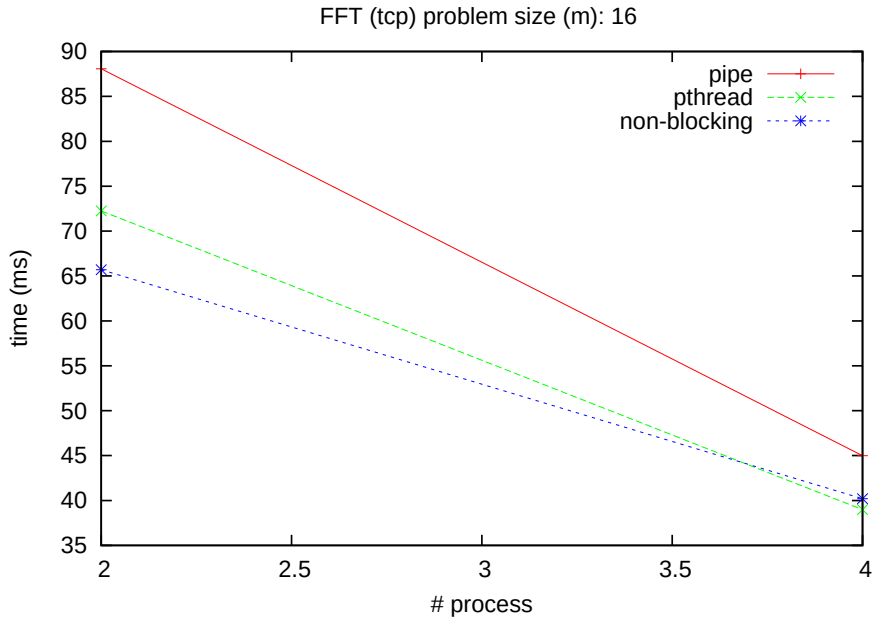


(b) $m = 24$

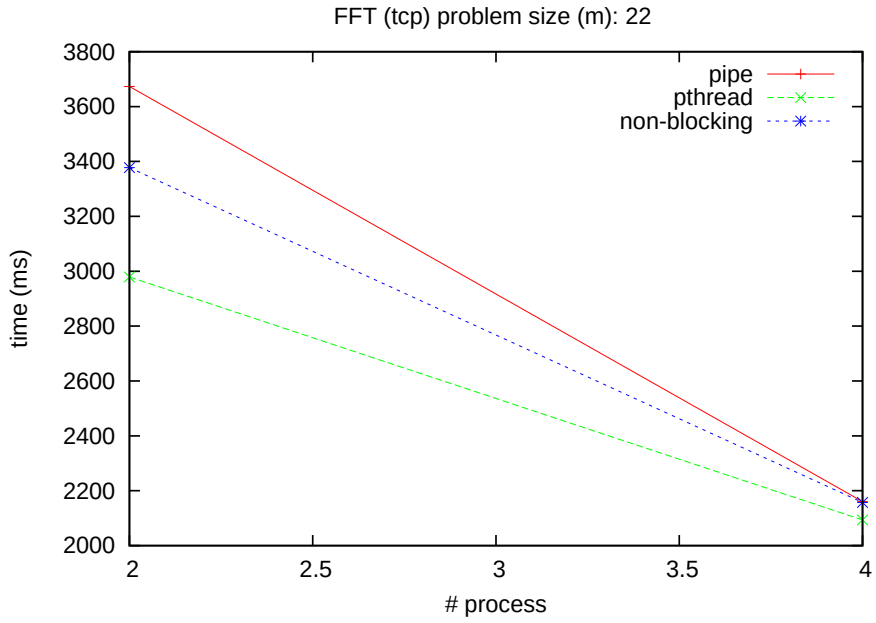
Figura 4.48: *Resultados da execução do FFT com variação do número de processadores obtidos com a biblioteca MPICH2 para entrada de tamanho* 2^{16} *(a) e* 2^{24} *(b).*

próximo da implementação com primitiva não-bloqueante, e em alguns casos melhor – como mostram os gráficos da Figura 4.49 com 2 e 4 processadores.

A sobreposição ocorre nas fases 1 e 2 do algoritmo, que tenta sobrepor a comunicação com computação nas implementações com primitiva não-bloqueante e *multi-threaded*. Os resultados na Figura 4.48(a) mostram que para 2 processos e com entrada de tamanho 2^{16}



(a) $m = 16$



(b) $m = 22$

Figura 4.49: *Resultados da execução do FFT em um par de estações do C3SL para 2 e 4 processadores obtidos com a biblioteca OPENMPI com Thread de progresso. Execução com entrada de tamanho* 2^{16} *(a) e* 2^{22} *(b).*

as implementações que promovem sobreposição de computação com comunicação têm melhor desempenho que a implementação linha-a-linha sem sobreposição. A Tabela 4.15 mostra os tempos (em *ms*) consumidos na fase 1 (F1) do algoritmo com entrada de tamanho 2^{16} com 2 processos, e também o tempo referente à transposição (T) e à

computação (C) para as implementações original e linha-a-linha sem sobreposição.

	transposição completa	transposição linha-a-linha		
	original	sem sobreposição	não-bloqueante	*multi-threaded*
T	11	19	–	–
C	8	8	–	–
$F1$	20	27	22	23

Tabela 4.15: Tempo de execução da primeira fase do algoritmo *FFT*.

Pode-se notar que o tempo consumido com a transposição completa da matriz implementada na versão original tem melhor desempenho do que na implementação linha-a-linha sem sobreposição. Neste teste, a implementação original usa mensagens de 256kB, enquanto a implementação linha-a-linha usa mensagens de 2kB, e sabe-se pelos resultados dos testes de largura de banda que neste intervalo a vazão é maior com mensagens maiores. Na implementação linha-a-linha sem sobreposição, a fase 1 consome 27 ms, dos quais 19 ms são despendidos na transposição da matriz e 8 ms com computação. Nas implementações não-bloqueante e *multi-threaded* o tempo consumido na fase 1 é menor do que a soma dos tempos despendidos na transposição (T) e na computação (C) medidos isoladamente na implementação sem sobreposição. Neste teste o algoritmo linha-a-linha permitiu a sobreposição de computação com comunicação, mas este ganho não foi suficiente para superar a queda de desempenho com o uso de mensagens menores na fase de transposição.

Para alguns casos, o desempenho da implementação linha-a-linha supera o desempenho da implementação original, como nos resultados mostrados na Figura 4.46(a) com entrada de tamanho 2^{22} e com 8 processadores. Neste teste a transposição da fase 1 na implementação original consumiu 346.76 ms, enquanto que na implementação linha-a-linha consumiu 296.28 ms. Este resultado mostra que para entradas maiores e com mais processos envolvidos, a transposição linha-a-linha pode ser mais eficiente.

A abordagem linha-a-linha permite empregar as técnicas de sobreposição da comunicação, que pelos resultados apresentados aqui são mais eficientes no uso dos recursos com o acréscimo do número de processos e do tamanho da entrada. Testes com mais processos e com entradas ainda maiores são necessários para validar a vantagem do

uso desta abordagem no *FFT*.

É importante notar que estas técnicas dependem do patamar de evolução da tecnologia, e também de características do sistema de comunicação. Aplicações projetadas com o objetivo de promover sobreposição da comunicação, podem se adaptar com mais eficiencia a novos patamares de evolução, ou mesmo a mudanças no sistema.

4.3.2 Radix Sort

Esta Seção descreve o *kernel Radix Sort* e diferentes implementações usadas na avaliação de desempenho.

Em [13], os autores descrevem uma versão *multi-threaded* do algoritmo com base em uma versão que é identificada como original aqui. Foi implementada uma nova versão *multi-threaded* e uma nova versão baseada em primitivas não-bloqueantes, que também é baseada na implementação original.

4.3.2.1 Descrição

O *kernel Radix Sort* [15] implementa a ordenação de números inteiros, que é usada em uma grande quantidade de aplicações, como sistemas de gerenciamento de banco de dados.

Cada processo recebe um subconjunto das chaves que serão ordenadas. Cada iteração do algoritmo ordena as chaves com base em um bloco de r bits. Para chaves de b bits, são necessárias b/r iterações do algoritmo. Cada iteração é composta por três fases, representadas na Figura 4.50. Na primeira fase cada processo contrói um histograma local, que representa a distribuição dos dígitos das chaves em sua posse. Na segunda fase, os histogramas locais são combinados em um histograma global. Na terceira fase, as chaves são permutadas com base no histograma global e distribuídas entre os processos.

A comunicação neste algoritmo ocorre com mais intensidade na terceira fase. Durante esta fase, o programa percorre as chaves e as agrupa em um *buffer* de acordo com o destino, como representa a Figura 4.51. Quando a quantidade de chaves acumuladas atinge um determinado limite, estas são enviadas para o destino – e o algoritmo prossegue. Esta fase apresenta um padrão irregular de comunicação. Os processos recebem mensagens que não

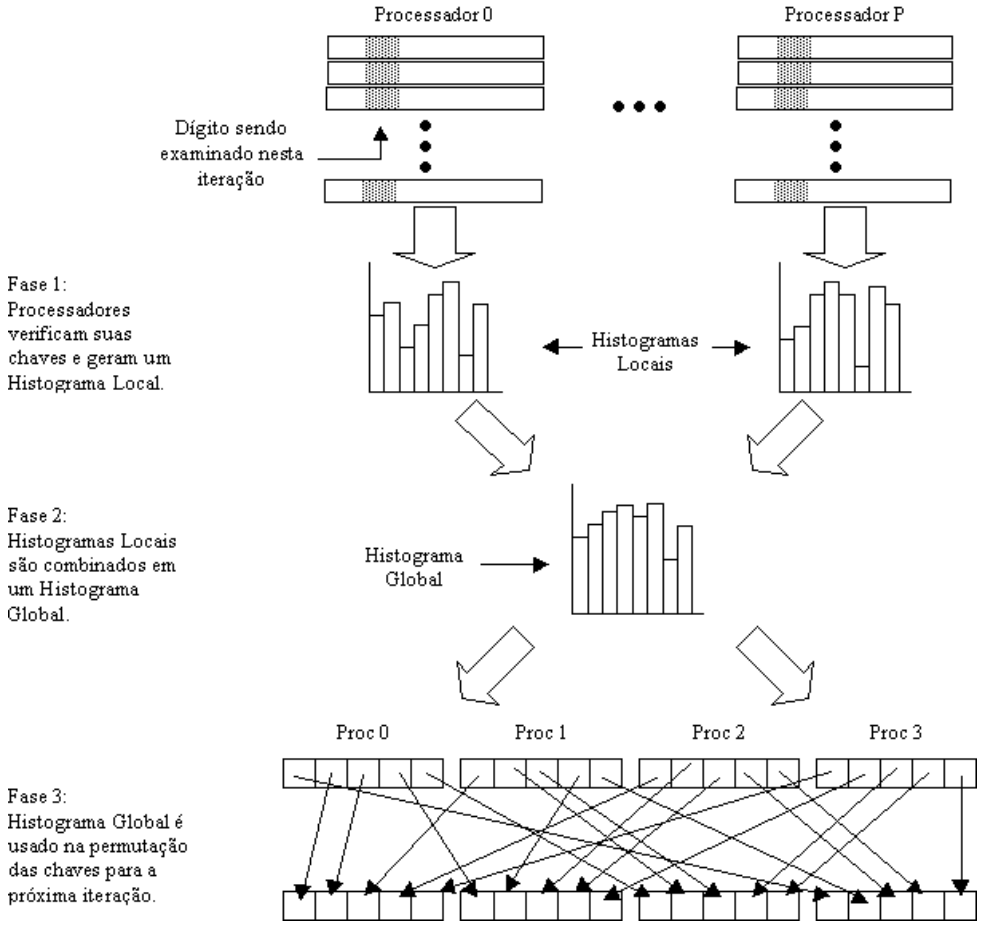


Figura 4.50: Fases do algoritmo do *Radix Sort*.

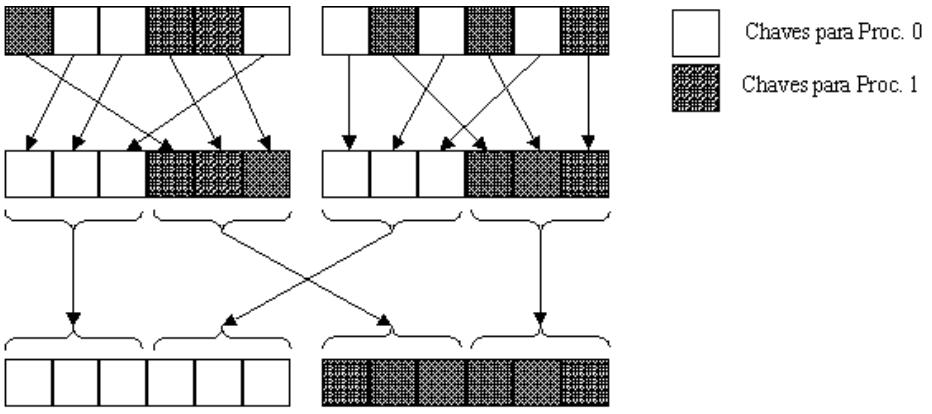


Figura 4.51: Fase de permutação das chaves para distribuição.

são esperadas.

As Figuras 4.50 e 4.51 foram retiradas de [13].

Implementação com primitivas de recepção não-bloqueante Quando são usadas mensagens longas na fase de distribuição das chaves na versão original, se dois processos tentam enviar chaves um para o outro ao mesmo tempo, o programa pode entrar em *deadlock*. Isto pode ocorrer quando dois processos ficam bloqueados em operações de envio um para o outro, que não podem completar enquanto um e outro não iniciarem a recepção.

A implementação não-bloqueante é semelhante à versão original, mas usa primitivas de recepção não-bloqueantes para configurar operações de recebimento antecipadamente. Esta implementação configura uma operação de recepção para cada processo, e sempre que uma mensagem é recebida, uma nova operação de recepção é configurada – assim sempre há uma operação de recepção já configurada quando um processo envia uma mensagem.

Assim como na versão original, nesta implementação o algoritmo deve verificar se há mensagens disponíveis. O teste é realizado através da primitiva MPI_Test, que não bloqueia a execução e retorna um valor que indica se há ou não mensagens prontas para recepção. Na implementação original, há um laço de repetição na fase de cálculo da distribuição das chaves que percorre todas as chaves na posse do processo. Tanto na implementação original, quanto na implementação com primitiva não-bloqueante, o teste de mensagens disponíveis é efetuado a cada iteração deste laço.

Implementação multi-threaded Em [13], os autores descrevem uma implementação *multi-threaded* na qual são usados três *threads* distintos. Um *Thread* é responsável pela criação do histograma local, criação do histograma global e cálculo da distribuição das chaves. Os outros *threads* são responsáveis pela função de envio das chaves aos destinos, e outro *thread* pela recepção das chaves enviadas por outros processos.

A versão *multi-threaded* criada neste trabalho emprega múltiplos *threads* apenas na fase de distribuição das chaves. O *thread* principal continua responsável pelo cálculo da distribuição das chaves e transmissão das chaves aos destinos. Somente um *thread* auxiliar

existe para receber as chaves enviadas por outros processos. Nesta implementação, a recepção das chaves ocorre concorrentemente com as fases de cálculo da distribuição e envio das chaves. Sempre que uma mensagem é enviada ao processo, o *thread* recebe a mensagem e armazena as chaves concorrentemente com o *thread* principal que realiza o cálculo da distribuição e o envio das mensagens. Assim, pode haver sobreposição de comunicação com comunicação, bem como de comunicação com computação.

Para que a aplicação possa chamar funções de comunicação do MPI concorrentemente, deve-se usar a opção MPI_MULTIPLE_THREAD na inicialização do MPI. A versão 1.0 do *OPENMPI* não suporta múltiplos *threads*, e esta funcionalidade foi incluída a partir da versão 1.1. A versão do *OPENMPI* que possui o protolo *LLC* integrado é 1.0, e por isso não foi possível executar testes da versão *multi-threaded* com o protocolo *LLC*.

Tamanho das mensagens Na fase de distribuição das chaves, o algoritmo percorre as chaves na posse do processo e as armazena em um buffer de acordo com o destino. Quando o buffer atinge uma determinada quantidade de chaves acumuladas para um processo, estas são enviadas para o destino. Na implementação original o tamanho do *buffer* é de 1403 bytes, para que a mensagem possa ser transmitida em um pacote Ethernet. Neste trabalho o programa foi modificado para receber o tamanho do *buffer* como argumento, e assim foram testados diversos tamanhos de mensagem.

4.3.2.2 Avaliação de Desempenho

Todos os testes apresentados nesta Seção foram executados com 524288 chaves por processo. Cada chave é um inteiro de 4 bytes dividida em blocos de 8 bits.

A Figura 4.52 mostra os resultados do *kernel* Radix com variação do número de processos e do tamanho do *buffer* usado na fase de distribuição das chaves. Pode-se notar que a versão *multi-threaded* mostrou melhor desempenho que as demais, principalmente com o acréscimo do número de processos envolvidos.

Os resultados mostram também que usar um *buffer* pequeno para que a mensagem possa ser transmitida em um único pacote Ethernet, como implementado originalmente,

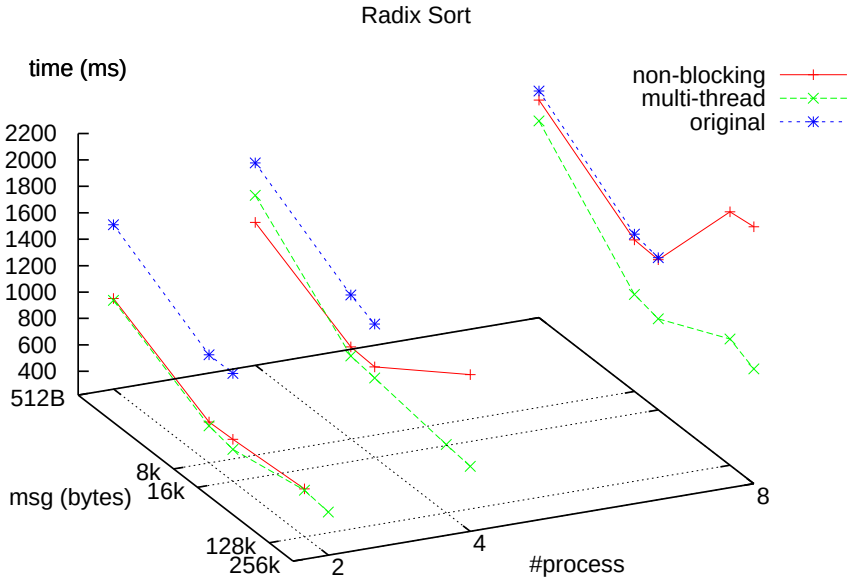
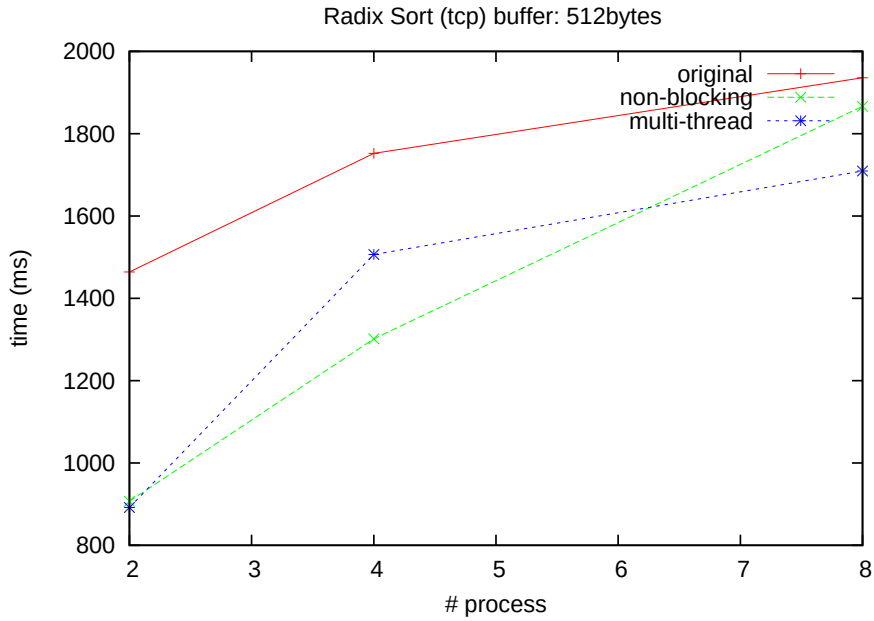


Figura 4.52: *Comparação do desempenho das implementações do Radix Sort para 2, 4 e 8 processos, e mensagens de tamanho 128, 2048, 4096.*

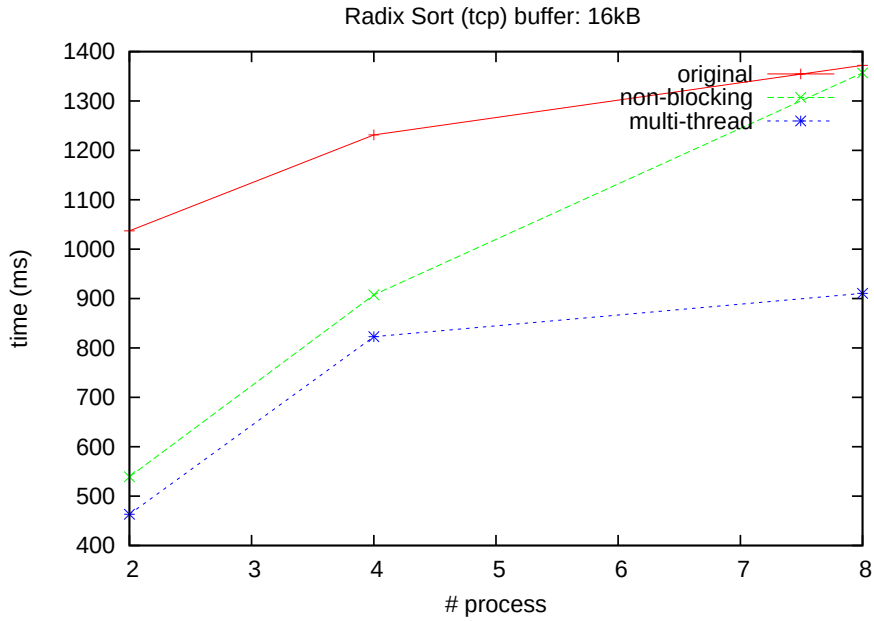
não apresenta o melhor desempenho. Neste teste, o melhor desempenho foi alcançado com mensagens de 16kB.

A Figura 4.53 mostra o desempenho do *kernel* Radix com *buffer* de 512 bytes (a) e com *buffer* 16kB (b), obtidos com 2, 4, e 8 processos. O tamanho da entrada é de 512k chaves por processo. A implementação *multi-threaded* foi a que apresentou melhor desempenho e melhor escalabilidade. Com 8 processadores o ganho em relação à implementação original foi da ordem de 50%.

No caso do Radix, o uso de múltiplos threads além de contribuir para a sobreposição da comunicação também melhora o mecanismo usado para receber mensagens que não são esperadas. Na implementação original, durante a execução do laço de repetição responsável por calcular a distribuição das chaves, o programa tem que executar periodicamente a função que trata as mensagens recebidas.



(a) $512B$



(b) $16k$

Figura 4.53: *Desempenho do Radix Sort variando o número de processos com mensagens de 512 bytes (a) e 16k (b) obtidos com a biblioteca OPENMPI.*

CAPÍTULO 5

CONCLUSÃO

Neste trabalho foram discutidos diversos parâmetros que devem compor a análise de um sistema de comunicação, bem como guiar o projeto de aplicações paralelas. Também foram avaliadas algumas variantes do modelo *LogP*, e descritas extensões para ajustar o modelo ao ambiente estudado. Foi discutida a importância de considerar outros parâmetros além da largura de banda e latência na avaliação dos sistemas de comunicação. Também foram descritos testes usados para medir os parâmetros que caracterizam um sistema baseado no padrão *Message Passing interface (MPI)*, mostrando a necessidade de considerar semântica das primitivas de comunicação para implementar testes mais precisos. Para complementar a análise, foram implementados novos testes que permitem medir alguns parâmetros do sistema, tais como a oportunidade de sobreposição em diferentes padrões de comunicação e também a sobrecarga adicional de processamento.

Os resultados mostram ganho de desempenho do protocolo *LLC* em relação ao *TCP/IP* da ordem de 7% para mensagens pequenas e de 12% com mensagens de 16kB – o que mostra que o *LLC* é uma boa solução para aglomerados ligados por rede local. Problemas encontrados na implementação da biblioteca *OPENMPI*, que suporta o protocolo *LLC*, e as medidas erráticas observadas no sistema dificultaram a avaliação do *LLC*, bem como a medição de alguns parâmetros. Portanto, os resultados para o protocolo *LLC* estão incompletos. Apesar de ter mostrado maior vazão, não foi possível determinar com precisão o *overhead* de processamento, e nem o modo como o protocolo *LLC* afeta a oportunidade de sobreposição.

Os resultados mostram também que as técnicas de sobreposição de comunicação apresentaram ganho de desempenho nos testes com os *kernels FFT* e *Radix*. A abordagem *multi-threaded* mostrou-se eficiente, principalmente com o padrão irregular de comunicação do *kernel* Radix – que teve ganho da ordem de 50%. No caso do *FFT*, que apresenta

padrão de comunicação regular, o uso de primitivas não-bloqueantes apresentou melhor desempenho na maioria dos casos.

A oportunidade de sobreposição depende do patamar de evolução da tecnologia, bem como de características do sistema de comunicação – como a capacidade de progredir a comunicação iniciada por primitivas não-bloqueantes. Aplicações projetadas com as técnicas de sobreposição da comunicação, mesmo que não apresentem ganho de desempenho em uma determinada configuração, podem adaptar-se com mais eficiência a novos patamares de evolução da tecnologia, e também à mudanças no sistema de comunicação.

Tecnologias de rede mais recentes, tais como Infiniband [1] e Myrinet [5] são baseadas em dispositivos com maior poder de processamento. Estes dispositivos podem realizar parte do trabalho que normalmente é atribuido à CPU principal – e assim diminuir a sobrecarga de processamento e aumentar a oportunidade de sobreposição. Este trabalho foi baseado em aglomerados ligados por rede Ethernet, que é uma solução de baixo custo. Entretanto, a metodologia e os testes descritos neste trabalho podem ser usados na avaliação de outros sistemas.

Trabalhos Futuros

O protocolo *LLC* mostrou bom desempenho, mas para ser avaliado de forma mais completa é necessário reproduzir os testes que foram prejudicados pelos erros encontrados na biblioteca OPENMPI e pelas medidas erráticas obtidas no sistema. O problema das medidas erráticas observadas com os dispositivos de rede *Marvell Yukon* disponíveis no ambiente avaliado pode ser resolvido com a atualização do *software* [1]. O erro encontrado na biblioteca OPENMPI ainda não foi resolvido, mas quando houver uma atualização será preciso integrar o *LLC* à versão mais recente.

O ambiente analisado usa o tamanho padrão do pacote Ethernet, que é de 1500 bytes. Também pode ser avaliado o efeito que o tamanho do pacote Ethernet tem no desempenho do sistema de comunicação. Os parâmetros que influenciam o desempenho das bibliotecas

[1] http://gentoo-wiki.com/HARDWARE_sk98lin

MPI também podem ser analisados, como o limite usado para classificar mensagens curtas e longas pela implementação do MPI. O conjunto de testes pode incluir também uma análise das funções do padrão MPI-2, que não foram testadas neste trabalho.

Os testes com os *kernels FFT* e *Radix* mostraram que a sobreposição de comunicação pode melhorar o desempenho das aplicações. Os dois *kernels* são sensíveis à largura de banda, por usarem mensagens grandes. Aplicações sensíveis à latência ou com outros padrões de comunicação podem ser incluídas na análise.

A oportunidade de sobreposição da comunicação depende do patamar de evolução da tecnologia, bem como de características do sistema. O projeto de aplicações paralelas pode buscar sobrepor a comunicação com a computação para construir aplicações que se adaptam com mais eficiência à novos patamares de evolução, e também a mudanças no sistema de comunicação. Os testes com os *kernels* podem ser reproduzidos com o objetivo de avaliar a eficiência destas técnicas ao mudar o patamar de evolução ou outros parâmetros do sistema. Esta capacidade de adaptação a evolução da tecnologia é importante para que as aplicações se beneficiem de atualizações do sistema, e também para aumentar a vida útil da aplicação.

APÊNDICE A

LISTAGEM DE CÓDIGO FONTE

Neste apêndice são apresentados os códigos fonte de alguns dos testes implementados neste trabalho. Os testes foram desenvolvidos com base na ferramenta Perftest. A ferramenta está disponível separadamente [1] e em conjunto com a biblioteca MPICH, que é distribuída sob a licença listada a seguir.

Licença MPICH

[1] http://www-unix.mcs.anl.gov/mpi/mpich1/perftest/

Fonte 1 Teste do padrão todos-para-todos com primitivas de envio e recepção conjugadas.

```
double all2all_sendrecv_replace(int reps, int len, PairData ctx){
  int *destid;
  double elapsed_time;
  int  p, i, to = ctx->destination, from = ctx->source;
  int  recv_from;
  char *sbuffer, *rbuffer;
  GET_TIME_TYPE t0, t1;
  MPI_Status status;

  destid = (int *) malloc(__NUMNODES * sizeof(int));
  if (destid == NULL) {
      fprintf(stderr,"Could not malloc memory for destid\n");
      MPI_Abort(MPI_COMM_WORLD,1); exit(1);
  }

  for (i = 0; i < __NUMNODES; i++) destid[i] = i;
  dest_sort(destid, __NUMNODES);

  sbuffer = (char *)malloc(len);
  rbuffer = (char *)malloc(len);
  memset( sbuffer, 0, len );
  memset( rbuffer, 0, len );

  SetupTest( from );
  ConfirmTest( reps, len, ctx );
  //warmup
  if(ctx->is_master){
    recv_from = MPI_ANY_SOURCE;
    if (source_type == SpecifiedSource) recv_from = to;
    MPI_Recv(rbuffer,len,MPI_BYTE,recv_from,0,MPI_COMM_WORLD,&status);
  }
  if(ctx->is_slave){
    recv_from = MPI_ANY_SOURCE;
    if (source_type == SpecifiedSource) recv_from = to;
    MPI_Send(sbuffer,len,MPI_BYTE,from,0,MPI_COMM_WORLD);
  }

  elapsed_time = 0;
  t0 = get_time();
  for(i=0;i<reps;i++){
     for (p=0; p < __NUMNODES; p++){
      if (destid[p] != __MYPROCID){
            MPI_Sendrecv_replace(sbuffer, len, MPI_BYTE, destid[p], MSG_TAG(i), destid[p],
                MSG_TAG(i), MPI_COMM_WORLD, &status);

      }
     }
  }
  t1 = get_time();
  elapsed_time = t1-t0;

  FinishTest();
  free(sbuffer);
  free(rbuffer);
  free(destid);
  return(elapsed_time);

}
```

Fonte 2 Teste do padrão todos-para-todos usando primitivas não-bloqueantes. As operações de recepção são pré-configuradas, e então é efetuada a troca de mensagens entre os processos participantes.

```
double all2all_nb_send(int reps, int len, PairData ctx){
  int *destid;
  int r=0;
  double elapsed_time;
  int  p, i, to = ctx->destination, from = ctx->source;
  int  recv_from;
  char *sbuffer, *rbuffer;
  GET_TIME_TYPE t0, t1;
  MPI_Status status;
  MPI_Request *request_recv, *request_send;
  destid = (int *) malloc(__NUMNODES * sizeof(int));
  if (destid == NULL) {
      fprintf(stderr,"Could not malloc memory for destid\n");
      MPI_Abort(MPI_COMM_WORLD,1); exit(1);
  }
  request_recv = (MPI_Request *) malloc((__NUMNODES-1) * sizeof(MPI_Request));
  if (request_recv == NULL) {
      fprintf(stderr,"Could not malloc memory for request_recv\n");
      MPI_Abort(MPI_COMM_WORLD,1); exit(1);
  }
  request_send = (MPI_Request *) malloc((__NUMNODES-1) * sizeof(MPI_Request));
  if (request_send == NULL) {
      fprintf(stderr,"Could not malloc memory for request_send\n");
      MPI_Abort(MPI_COMM_WORLD,1); exit(1);
  }

  for (i = 0; i < __NUMNODES; i++) destid[i] = i;
  dest_sort(destid, __NUMNODES);

  sbuffer = (char *)malloc(len * (__NUMNODES ));
  rbuffer = (char *)malloc(len * (__NUMNODES ));
  memset( sbuffer, 0, len * __NUMNODES);
  memset( rbuffer, 0, len * __NUMNODES);

  SetupTest( from );
  ConfirmTest( reps, len, ctx );
  //warmup
  if(ctx->is_master){
    recv_from = MPI_ANY_SOURCE;
    if (source_type == SpecifiedSource) recv_from = to;
    MPI_Recv(rbuffer,len,MPI_BYTE,recv_from,0,MPI_COMM_WORLD,&status);
  }
  if(ctx->is_slave){
    recv_from = MPI_ANY_SOURCE;
    if (source_type == SpecifiedSource) recv_from = to;
    MPI_Send(sbuffer,len,MPI_BYTE,from,0,MPI_COMM_WORLD);
  }

  elapsed_time = 0;
  t0 = get_time();
  for(i=0;i<reps;i++){
     r=0;
     for (p=0; p < __NUMNODES; p++){
      if (destid[p] != __MYPROCID){
            MPI_Irecv(sbuffer, len, MPI_BYTE, destid[p], MSG_TAG(i),
               MPI_COMM_WORLD, &request_recv[r++] );
      }
     }
     r=0;
     for (p=0; p < __NUMNODES; p++){
      if (destid[p] != __MYPROCID){
             MPI_Isend(sbuffer, len,  MPI_BYTE, destid[p], MSG_TAG(i),
                  MPI_COMM_WORLD,  &request_send[r++] );
      }
     }
     //wait for sends and receives to be completed
     MPI_Waitall(__NUMNODES-1, request_recv, MPI_STATUS_IGNORE);
     MPI_Waitall(__NUMNODES-1, request_send, MPI_STATUS_IGNORE);
  }//reps

  t1 = get_time();
  elapsed_time = t1-t0;

  FinishTest();
  free(sbuffer);
  free(rbuffer);
  free(destid);
  free(request_recv);
  free(request_send);
  return(elapsed_time);

}
```

Fonte 3 Teste do padrão todos-para-todos usando a operação coletiva MPI_Alltoall.

```
double all2all_mpi(int reps, int len, PairData ctx){
  int *destid;
  double elapsed_time;
  int  i, to = ctx->destination, from = ctx->source;
  int  recv_from;
  char *sbuffer, *rbuffer;
  GET_TIME_TYPE t0, t1;
  MPI_Status status;

  destid = (int *) malloc(__NUMNODES * sizeof(int));
  if (destid == NULL) {
      fprintf(stderr,"Could not malloc memory for destid\n");
      MPI_Abort(MPI_COMM_WORLD,1); exit(1);
  }

   for (i = 0; i < __NUMNODES; i++) destid[i] = i;
   dest_sort(destid, __NUMNODES);

  sbuffer = (char *)malloc(len * __NUMNODES);
  rbuffer = (char *)malloc(len * __NUMNODES);
  memset( sbuffer, 0, len * __NUMNODES);
  memset( rbuffer, 0, len * __NUMNODES);

  SetupTest( from );

  ConfirmTest( reps, len, ctx );
  //warmup
  if(ctx->is_master){
    recv_from = MPI_ANY_SOURCE;
    if (source_type == SpecifiedSource) recv_from = to;
    MPI_Recv(rbuffer,len,MPI_BYTE,recv_from,0,MPI_COMM_WORLD,&status);
  }
  if(ctx->is_slave){
    recv_from = MPI_ANY_SOURCE;
    if (source_type == SpecifiedSource) recv_from = to;
    MPI_Send(sbuffer,len,MPI_BYTE,from,0,MPI_COMM_WORLD);
  }

  elapsed_time = 0;
  t0 = get_time();
  for(i=0;i<reps;i++){
    MPI_Alltoall(sbuffer, len, MPI_BYTE, rbuffer, len, MPI_BYTE, MPI_COMM_WORLD);
  }
  t1 = get_time();
  elapsed_time = t1-t0;

  FinishTest();
  free(sbuffer);
  free(rbuffer);
  free(destid);
  return(elapsed_time);

}
```

Fonte 4 Teste de sobreposição da comunicação com padrão ping-pong.

```
double round_trip_nb_overlap( int reps, int len, void *vctx)
{
    double elapsed_time=0;
    OverlapData *ctx = (OverlapData *)vctx;
    int  i,myproc,
    proc1=ctx->proc1,proc2=ctx->proc2,MsgSize=ctx->MsgSize;
    char *rbuffer,*sbuffer;
    GET_TIME_TYPE t0, t1, time0, time1;
    MPI_Request rid, sid;
    MPI_Status  status;

    /* If the MsgSize is negative, just do the floating point computation. */
    if (MsgSize < 0) {
        return computation(reps, len , vctx);
    }

    //setts the send function based on communication mode
    ISendFunctionPtr send = GetMPI_ISendFunction(ctx->comm_mode);

    myproc = __MYPROCID;
    sbuffer = (char *)malloc(MsgSize);
    rbuffer = (char *)malloc(MsgSize);
    SetupOverlap(len,ctx);
    elapsed_time = 0;
    if(myproc==proc1){
    MPI_Recv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,&status);
    t0=get_time();
    for(i=0;i<reps;i++){
        MPI_Irecv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&(rid));
        send(sbuffer,MsgSize,MPI_BYTE,proc2,1,MPI_COMM_WORLD,&(sid));

        OverlapComputation(len,ctx);

        MPI_Wait(&(rid),&status);
        MPI_Wait(&(sid),&status);
    }
    t1=get_time();
    elapsed_time = t1 -t0;
    }

    if(myproc==proc2){
    MPI_Irecv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&(rid));
    send(sbuffer,MsgSize,MPI_BYTE,proc1,0,MPI_COMM_WORLD,&(sid));
    for(i=0;i<reps-1;i++){
        MPI_Wait(&(rid),&status);
        MPI_Wait(&(sid),&status);
        MPI_Irecv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&(rid));
        send(sbuffer,MsgSize,MPI_BYTE,proc1,1,MPI_COMM_WORLD,&(sid));
    }
    MPI_Wait(&(rid),&status);
    MPI_Wait(&(sid),&status);
    MPI_Send(sbuffer,MsgSize,MPI_BYTE,proc1,1,MPI_COMM_WORLD);
    }

    free(sbuffer);
    free(rbuffer);
    return(elapsed_time);
}
```

Fonte 5 Teste de sobreposição local. Este teste faz medidas de tempo de partes do teste isoladamente, tais como tempo consumido apenas com computação e também o tempo consumido na primitiva MPI_Wait. Desta forma é possível verificar o progresso da comunicação, apesar de introduzir alguma sobrecarga com o grande número de chamadas a função que recupera o tempo corrente.

```
double local_nb_overlap( int reps, int len, void *vctx)
    double elapsed_time;
    OverlapData *ctx = (OverlapData *)vctx;
    int  i,myproc,
    proc1=ctx->proc1,proc2=ctx->proc2,MsgSize=ctx->MsgSize;
    char *rbuffer,*sbuffer;
    GET_TIME_TYPE t0, t1, time0, time1;
    MPI_Request rid, sid;
    MPI_Status  status;
    register double elapsed_send=0, elapsed_swait=0, elapsed_comp=0;

    /* If the MsgSize is negative, just do the floating point computation.  */
    if (MsgSize < 0) {
    return computation(reps, len, vctx);
    }

    //get the send function (based on communication mode)
    ISendFunctionPtr send = GetMPI_ISendFunction(ctx->comm_mode);

    myproc = __MYPROCID;
    sbuffer = (char *)malloc(MsgSize);
    rbuffer = (char *)malloc(MsgSize);
    SetupOverlap(len,ctx);
    elapsed_time = 0;
    if(myproc==proc1){
    t0=get_time();
    for(i=0;i<reps;i++){

        MPI_Recv(rbuffer, 1, MPI_BYTE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
        t0 = time0 = get_time();

        send(sbuffer,MsgSize,MPI_BYTE,proc2,1,MPI_COMM_WORLD,&(sid));
        time1 = get_time();
        elapsed_send += (time1 - time0);

        OverlapComputation(len,ctx);
        time0 = get_time();
        elapsed_comp += (time0 - time1);

        MPI_Wait(&(sid),&status);
        time1 = get_time();
        elapsed_swait += (time1 - time0);
        elapsed_time += (time1 - t0);
    }
    }
    if(myproc==proc2){
    for(i=0;i<reps-1;i++){
        MPI_Irecv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&rid);
        MPI_Send(sbuffer, 1, MPI_BYTE, proc1, 0, MPI_COMM_WORLD);
        MPI_Wait(&(rid),&status);
    }
    MPI_Irecv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&rid);
    MPI_Send(sbuffer, 0, MPI_BYTE, proc1, 0, MPI_COMM_WORLD);
    MPI_Wait(&(rid),&status);
    }
    tresult.values[0] = elapsed_send;
    tresult.values[1] = elapsed_swait;
    tresult.values[2] = elapsed_comp;
    free(sbuffer);
    free(rbuffer);
    return(elapsed_time);
}
```

Fonte 6 Teste de sobrecarga de processamento no envio de mensagens e de sobreposição do gap com primitiva não-bloqueante.

```
double gap_nb_overlap( int reps, int len, void *vctx)
{
    double elapsed_time;
    OverlapData *ctx = (OverlapData *)vctx;
    int  i,myproc,
    proc1=ctx->proc1,proc2=ctx->proc2,MsgSize=ctx->MsgSize;
    char *rbuffer,*sbuffer;
    GET_TIME_TYPE t0, t1, time0, time1;
    MPI_Request rid, sid;
    MPI_Status  status;
   register double elapsed_swait=0, elapsed_comp=0;
   myproc = __MYPROCID;
    /* If the MsgSize is negative, just do the floating point computation. */
    if (MsgSize < 0) {
        return computation(reps, len , vctx);
    }

    //get the send function (based on communication mode)
    ISendFunctionPtr send = GetMPI_ISendFunction(ctx->comm_mode);

    sbuffer = (char *)malloc(MsgSize);
    rbuffer = (char *)malloc(MsgSize);
    SetupOverlap(len,ctx);
    if(myproc==proc1){
    MPI_Recv(rbuffer, 1, MPI_BYTE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    t0=get_time();
    for(i=0;i<reps;i++){
        send(sbuffer,MsgSize,MPI_BYTE,proc2,1,MPI_COMM_WORLD, &sid);
        time0 = get_time();
        OverlapComputation(len,ctx);
        time1 = get_time();
        elapsed_comp += (time1 - time0);

        MPI_Wait(&sid, MPI_STATUS_IGNORE);
        elapsed_swait += (get_time() - time1);
    }
    t1 = get_time();
    elapsed_time = (t1 - t0);
    }

    if(myproc==proc2){
    MPI_Send(sbuffer, 1, MPI_BYTE, proc1, 0, MPI_COMM_WORLD);
    for(i=0;i<reps-1;i++){
        MPI_Irecv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&rid);
        MPI_Wait(&(rid),&status);
    }
        MPI_Irecv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&rid);
    MPI_Wait(&(rid),&status);
    }

    tresult.values[1] = elapsed_swait;
    tresult.values[2] = elapsed_comp;
    free(sbuffer);
    free(rbuffer);
    return(elapsed_time);
}
```

Fonte 7 Teste de sobrecarga de processamento no envio de mensagens e de sobreposição do gap com primitiva bloqueante.

```
double gap_b_overlap( int reps, int len, void *vctx)
{
    double elapsed_time;
    OverlapData *ctx = (OverlapData *)vctx;
    int  i,myproc,
    proc1=ctx->proc1,proc2=ctx->proc2,MsgSize=ctx->MsgSize;
    char *rbuffer,*sbuffer;
    GET_TIME_TYPE t0, t1, time0, time1;
    MPI_Request rid, sid;
    MPI_Status  status;
    register double elapsed_comp=0;

    /* If the MsgSize is negative, just do the floating point computation. */
    if (MsgSize < 0) {
        return computation(reps, len , vctx);
    }
    //get the send function (based on communication mode)
    SendFunctionPtr send = GetMPI_SendFunction(ctx->comm_mode);

    myproc = __MYPROCID;
    sbuffer = (char *)malloc(MsgSize);
    rbuffer = (char *)malloc(MsgSize);
    SetupOverlap(len,ctx);
    elapsed_time = 0;
    if(myproc==proc1){
    MPI_Recv(rbuffer, 0, MPI_BYTE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    t0=get_time();
    for(i=0;i<reps;i++){
        send(sbuffer,MsgSize,MPI_BYTE,proc2,1,MPI_COMM_WORLD);
        time1 = get_time();
        OverlapComputation(len,ctx);
        time0 = get_time();
        elapsed_comp += (time0 - time1);
    }
    t1 = get_time();
    elapsed_time = (t1 - t0);
    }
    if(myproc==proc2){
    MPI_Send(sbuffer, 0, MPI_BYTE, proc1, 0, MPI_COMM_WORLD);
    for(i=0;i<reps-1;i++){
        MPI_Irecv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&rid);
        MPI_Wait(&(rid),&status);
    }
        MPI_Irecv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&rid);
    MPI_Wait(&(rid),&status);
    }
    tresult.values[0] = tresult.values[1] = 0;
    tresult.values[2] = elapsed_comp;
    free(sbuffer);
    free(rbuffer);
    return(elapsed_time);
}
```

Fonte 8 Teste de sobrecarga de processamento na recepção de mensagens com primitiva não-bloqueante.

```
double gap_nb_recv_overlap( int reps, int len, void *vctx)
{
    double elapsed_time;
    OverlapData *ctx = (OverlapData *)vctx;
    int  i,myproc,
    proc1=ctx->proc1,proc2=ctx->proc2,MsgSize=ctx->MsgSize;
    char *rbuffer,*sbuffer;
    GET_TIME_TYPE t0, t1, time0, time1;
    MPI_Request rid, sid;
    MPI_Status  status;
    register double elapsed_recv=0, elapsed_rwait=0, elapsed_comp=0;
    myproc = __MYPROCID;
    if (MsgSize < 0) {
        return computation(reps, len , vctx);
    }
    //get the send function (based on communication mode)
    ISendFunctionPtr send = GetMPI_ISendFunction(ctx->comm_mode);

    sbuffer = (char *)malloc(MsgSize);
    rbuffer = (char *)malloc(MsgSize);
    SetupOverlap(len,ctx);
    if(myproc==proc1){
    MPI_Recv(rbuffer, 1, MPI_BYTE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    t0=get_time();
    for(i=0;i<reps;i++){
        MPI_Irecv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,1,MPI_COMM_WORLD,&rid);
        time0 = get_time();
        OverlapComputation(len,ctx);
        time1 = get_time();
        elapsed_comp += (time1 - time0);

        MPI_Wait(&(rid),&status);
        elapsed_rwait += (get_time() - time1);
    }
    t1 = get_time();
    elapsed_time = (t1 - t0);
    }
    if(myproc==proc2){
    MPI_Send(sbuffer, 1, MPI_BYTE, proc1, 0, MPI_COMM_WORLD);
    for(i=0;i<reps;i++){
        send(sbuffer,MsgSize,MPI_BYTE,proc1,1,MPI_COMM_WORLD, &sid);
        MPI_Wait(&sid, MPI_STATUS_IGNORE);
    }
    }
    tresult.values[0] = elapsed_recv;
    tresult.values[1] = elapsed_rwait;
    tresult.values[2] = elapsed_comp;
    free(sbuffer);
    free(rbuffer);
    return(elapsed_time);
}
```

Fonte 9 Teste de sobrecarga de processamento na recepção de mensagens com primitiva bloqueante.

```
double gap_b_recv_overlap( int reps, int len, void *vctx)
{
    double elapsed_time;
    OverlapData *ctx = (OverlapData *)vctx;
    int  i,myproc,
    proc1=ctx->proc1,proc2=ctx->proc2,MsgSize=ctx->MsgSize;
    char *rbuffer,*sbuffer;
    GET_TIME_TYPE t0, t1, time0, time1;
    MPI_Request rid, sid;
    MPI_Status  status;
   register double elapsed_comp=0;
   myproc = __MYPROCID;
    if (MsgSize < 0) {
        return computation(reps, len , vctx);
    }

    //get the send function (based on communication mode)
    SendFunctionPtr send = GetMPI_SendFunction(ctx->comm_mode);

    sbuffer = (char *)malloc(MsgSize);
    rbuffer = (char *)malloc(MsgSize);
    SetupOverlap(len,ctx);
    if(myproc==proc1){
    MPI_Recv(rbuffer, 1, MPI_BYTE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
    t0=get_time();
    for(i=0;i<reps;i++){
        MPI_Recv(rbuffer,MsgSize,MPI_BYTE,MPI_ANY_SOURCE,1,MPI_COMM_WORLD, &status);
        time0 = get_time();
        OverlapComputation(len,ctx);
        time1 = get_time();
        elapsed_comp += (time1 - time0);
    }
    t1 = get_time();
    elapsed_time = (t1 - t0);
    }
    if(myproc==proc2){
    MPI_Send(sbuffer, 1, MPI_BYTE, proc1, 0, MPI_COMM_WORLD);
    for(i=0;i<reps;i++){
        send(sbuffer,MsgSize,MPI_BYTE,proc1,1,MPI_COMM_WORLD);
    }
    }
    tresult.values[0] = elapsed_recv;
    tresult.values[1] = 0;
    tresult.values[2] = elapsed_comp;
    free(sbuffer);
    free(rbuffer);
    return(elapsed_time);
}
```

Fonte 10 Processo mestre do teste de largura de banda usando filas.

```
double bandwidth_nb_queue( int reps, int len, PairData ctx)
{
  double         elapsed_time = 0;
  int            i,j,k,r, to = ctx->destination, from = ctx->source;
  int            recv_from;
  MPI_Request *request;//array
  MPI_Status status;
  char         *sbuffer,*rbuffer;
  register double   t0, t1;

  int chunk = 0;
  int queue_len = g_queue_len;

  sbuffer = (char *)malloc(len);
  rbuffer = (char *)malloc(len);
  memset( sbuffer, 0, len);
  memset( rbuffer, 0, len );

  request = (MPI_Request *) malloc(queue_len * sizeof(MPI_Request));

  SetupTest( from );
  ConfirmTest( reps, len, ctx );

  if ( (len % queue_len) != 0){
      fprintf(stderr, "warn: bandwidth test: message len must be multiple of queue len!\n");
  }
  //chose total bytes and number of messages adaptively
  //large messages will flood the network earlier
  double total_bytes = 0;
  if (len <= 1024)
      total_bytes = (1024 * len);
  else if (len <= 4096)
      total_bytes = (512 * len);
  else if (len <= 32768)
      total_bytes = (64 * len);
  else if (len <= 131072)
      total_bytes = (24 * len);
  else
      total_bytes = (16 * len);
  //iterations to send total_bytes bytes with selected message len
  int iters = total_bytes /len;

  //each iteration send queue_len/2 messages (except for queue_len==1)
  if (queue_len > 1 ){
    iters = iters * 2;
  }
  //redefine len (each message len)
  len = len / queue_len;

  if(ctx->is_master){
    recv_from = MPI_ANY_SOURCE;
    if (source_type == SpecifiedSource) recv_from = to;
    MPI_Recv(rbuffer,len,MPI_BYTE,recv_from,0,MPI_COMM_WORLD,&status);
    for (r = 0; r < reps; r++){
        //post q/2 sends
        chunk = 0; //i=0;
        for (j = 0; j < queue_len/2; j++){
           MPI_Isend( &sbuffer[chunk+j], len, MPI_BYTE,to,MSG_TAG(j),
                  MPI_COMM_WORLD, &request[chunk + j]);
        }
        t0=get_time();
        for(i=1;i < iters; i++){
           if (queue_len == 1){
                MPI_Isend(&sbuffer[0],len,MPI_BYTE,to,MSG_TAG(i),
                      MPI_COMM_WORLD, &request[0]);
        MPI_Wait(&request[0], MPI_STATUS_IGNORE);
           }else{//queue_len > 1
          //post q/2 send
        chunk = ( i % 2 == 0)?0:(queue_len / 2);
        for (k = 0; k < queue_len/2; k++){
                MPI_Isend(&sbuffer[chunk+k],len,MPI_BYTE,to,MSG_TAG(k),
                      MPI_COMM_WORLD, &request[chunk+k]);
        }
        chunk = ( (i-1) % 2 == 0)?0:(queue_len / 2);
        MPI_Waitall(queue_len/2, &request[chunk], MPI_STATUS_IGNORE);
           }//queue_len
       }//iters
       if (queue_len > 1){
        chunk = ( (i-1) % 2 == 0)?0:(queue_len / 2);
        MPI_Waitall(queue_len/2, &request[chunk], MPI_STATUS_IGNORE);
       }
       MPI_Recv(rbuffer,1,MPI_BYTE,recv_from,0,MPI_COMM_WORLD,&status);
       t1=get_time();
       elapsed_time += t1-t0;
    }//reps
    //time to send 1024 bytes
    elapsed_time = elapsed_time / (total_bytes / 1024.0);
  }


 FinishTest();
 free(sbuffer);
 free(rbuffer);
 free(request);
 return(elapsed_time);
}
```

Fonte 11 Processo escravo do teste de largura de banda usando filas.

```
double bandwidth_nb_queue( int reps, int len, PairData ctx)
{
  double         elapsed_time = 0;
  int            i,j,k,r, to = ctx->destination, from = ctx->source;
  int            recv_from;
  MPI_Request *request;//array
  MPI_Status status;
  char         *sbuffer,*rbuffer;
  register double   t0, t1;

  int chunk = 0;
  int queue_len = g_queue_len;

  sbuffer = (char *)malloc(len);
  rbuffer = (char *)malloc(len);
  memset( sbuffer, 0, len);
  memset( rbuffer, 0, len );

  request = (MPI_Request *) malloc(queue_len * sizeof(MPI_Request));

  SetupTest( from );
  ConfirmTest( reps, len, ctx );

  if ( (len % queue_len) != 0){
      fprintf(stderr, "warn: bandwidth test: message len must be multiple of queue len!\n");
  }
  //chose total bytes and number of messages adaptively
  //large messages will flood the network earlier
  double total_bytes = 0;
  if (len <= 1024)
      total_bytes = (1024 * len);
  else if (len <= 4096)
      total_bytes = (512 * len);
  else if (len <= 32768)
      total_bytes = (64 * len);
  else if (len <= 131072)
      total_bytes = (24 * len);
  else
      total_bytes = (16 * len);
  //iterations to send total_bytes bytes with selected message len
  int iters = total_bytes /len;

  //each iteration send queue_len/2 messages (except for queue_len==1)
  if (queue_len > 1 ){
    iters = iters * 2;
  }
  //redefine len (each message len)
  len = len / queue_len;

  if(ctx->is_slave){
    recv_from = MPI_ANY_SOURCE;
    if (source_type == SpecifiedSource) recv_from = to;
    MPI_Send(sbuffer,len,MPI_BYTE,from,0,MPI_COMM_WORLD);
    for(r = 0; r < reps; r++){
        chunk = 0; //i=0;
    for (j = 0; j < queue_len/2; j++){
        MPI_Irecv(&rbuffer[chunk+j],len,MPI_BYTE,recv_from,MSG_TAG(j),
                MPI_COMM_WORLD,&request[chunk+j]);
        }

        for(i=1;i < iters; i++){
        if (queue_len == 1){
                MPI_Irecv(&rbuffer[0],len,MPI_BYTE,recv_from,MSG_TAG(i),
                MPI_COMM_WORLD,&request[0]);
                MPI_Wait(&request[0], MPI_STATUS_IGNORE);
        }else{//queue_len > 1
                chunk = ( i % 2 == 0)?0:(queue_len / 2);
                for (k = 0; k < queue_len/2; k++){
                    MPI_Irecv(&rbuffer[chunk+k],len,MPI_BYTE,recv_from,
                    MSG_TAG(k), MPI_COMM_WORLD,&request[chunk+k]);
            }
            chunk = ( (i-1) % 2 == 0)?0:(queue_len / 2);
                MPI_Waitall(queue_len/2, &request[chunk], MPI_STATUS_IGNORE);
        }//queue_len
        }//iters (total_bytes)
    if (queue_len > 1){
        chunk = ( (i-1) % 2 == 0)?0:(queue_len / 2);
        MPI_Waitall(queue_len/2, &request[chunk], MPI_STATUS_IGNORE);
    }
    MPI_Send(sbuffer,1,MPI_BYTE,from,0,MPI_COMM_WORLD);
     }//reps
 }
 FinishTest();
 free(sbuffer);
 free(rbuffer);
 free(request);
 return(elapsed_time);
}
```

BIBLIOGRAFIA

[1] *Infiniband specification. Release 1.0*, 2000. InfiniBand Trade Association, `http://www.infinibandta.org/specs`.

[2] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, e Chris Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[3] David H Bailey. FFTs in external or hierarchical memory. *Journal of Supercomputing*, 4:23–25, 1990.

[4] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, e K. Yelick. An evaluation of current high-performance networks, 2003. In IPDPS 2003.

[5] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, e Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[6] Michael J. Brim. Predicting MPI-Based parallel application performance on workstation clusters using LogGP. `http://www.cs.wisc.edu/~mjbrim/personal/classes/747/research.html`, 2002.

[7] D. E. Culler, L. T. Liu, R. P. Martin, e C. O. Yoshikawa. Assessing Fast Network Interfaces. *IEEE Micro*, 16(1):35–43, February de 1996.

[8] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, e Thorsten von Eicken. LogP: a practical model of parallel computation. *Commun. ACM*, 39(11):78–85, 1996.

[9] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, e Thorsten von Eicken. LogP:

Towards a realistic model of parallel computation. *Principles Practice of Parallel Programming*, páginas 1–12, 1993.

[10] David E. Culler, Lok T. Liu, Richard P. Martin, e Chad Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, February de 1996.

[11] Arnaldo Carvalho de Melo. TCPfying the poor cousins. Linux Symposium, 2004. `http://www.linuxsymposium.org/proceedings/reprints/Reprint-Melo-OLS2004.pdf`.

[12] Thomas Fahringer, Matthew Haines, e Piyush Mehrotra. On the utility of threads for data parallel programming. *International Conference on Supercomputing*, páginas 51–59, 1995.

[13] Sérgio Luiz Marques Filho. Uma abordagem multithread em aplicações paralelas utilizando mpi. Dissertação de mestrado, Universidade Federal do Paraná (UFPR) - Depto de Informática, Março de 2005. `http://www.inf.ufpr.br/~roberto/dissSergio.pdf`.

[14] Message Passing Interface Forum. MPI: A message-passing interface standard. Relatório Técnico UT-CS-94-230, 1994.

[15] G E Blelloch and C E Leiserson and B M Maggs and C G Plaxton and S J Smith and M Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. *ACM Proc Symp on Parallel Algorithms and Architectures*, páginas 3–16, 1991.

[16] P. B. Gibbons. A more practical PRAM model. *SPAA '89: Proceedings of the first annual ACM Symposium on Parallel Algorithms and Architectures*, páginas 158–168. ACM Press, 1989.

[17] IEEE. IEEE standards association. ANSI IEEE 802.3 standard., maio de 1998. `http://standards.ieee.org/getieee802/download/802.3ae-2002.pdf`.

[18] Fumihiko Ino, Noriyuki Fujimoto, e Kenichi Hagihara. LogGPS: a parallel computational model for synchronization analysis. *PPoPP '01: Proceedings of*

the eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, páginas 133–142. ACM Press, 2001.

[19] T. Kielmann, H. E. Bal, e S. Gorlatch. Bandwidth-efficient collective communication for clustered wide area systems. *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, páginas 492, Washington, DC, USA, 2000. IEEE Computer Society.

[20] Thilo Kielmann, Henri E. Bal, e Kees Verstoep. Fast measurement of LogP parameters for message passing platforms. *IPDPS '00: Proceedings of IPDPS 2000 Workshops on Parallel and Distributed Processing*, páginas 1176–1183. Springer-Verlag, 2000.

[21] Jens Mache. An assessment of gigabit ethernet as cluster interconnect. *IWCC '99: Proceedings of the 1st IEEE Computer Society International Workshop on Cluster Computing*, páginas 36. IEEE Computer Society, 1999.

[22] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[23] David A. Patterson e John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., 2002.

[24] Rafael H. Saavedra e Alan J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14(4):344–384, 1996.

[25] R. H. Saavedra-Barrera, A. J. Smith, e E. Miya. Machine characterization based on an abstract high-level language machine. *IEEE Trans. Comput.*, 38(12):1659–1679, 1989.

[26] Q. Snell, A. Mikler, e J. Gustafson. Netpipe: A network protocol independent performace evaluator. 1996. In IASTED International Conference on Intelligent Information Management and Systems, June 1996.

[27] Volker Strumpen. Software-based communication latency hiding for commodity workstation networks. *IEEE International Conference on Parallel Processing, Vol. 1*, páginas 146–153, 1996.

[28] Dave Turner e Xuehua Chen. Protocol-dependent message-passing performance on linux clusters. *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, páginas 187, Washington, DC, USA, 2002. IEEE Computer Society.

[29] Leslie Harlley Watter. Avaliação de desempenho do protocolo IEEE 802.2-LLC no kernel do linux. Dissertação de mestrado, Universidade Federal do Paraná (UFPR) - Depto de Informática, Março de 2006. `http://www.inf.ufpr.br/~roberto/dissLeslie.pdf`.

MARCELO LOYOLA STIVAL

AVALIAÇÃO DE DESEMPENHO EM AGLOMERADOS DE PCS INTERLIGADOS POR ETHERNET

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Roberto André Hexsel

CURITIBA

2006