Universidade Federal do Paraná

Departamento de Informática

Marcelo L Stival e Roberto A Hexsel

Avaliação de desempenho em aglomerados de PCs interligados por Ethernet

Relatório Técnico
RT-DINF 003/2007

Curitiba, PR
2007

Avaliação de desempenho em aglomerados de PCs interligados por Ethernet

Marcelo L Stival e Roberto A Hexsel

Departamento de Informática
Universidade Federal do Paraná
Centro Politécnico, C Postal 19081 – 81531-990 Curitiba, PR
roberto@inf.ufpr.br

Resumo: *Este trabalho contém uma avaliação de desempenho de um aglomerado de PCs interligados por rede Ethernet, que emprega o padrão MPI para troca de mensagens. A análise objetiva 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 adotado na avaliação de desempenho tem enfoque na comunicação, e é baseado no modelo LogP e em algumas de suas extensões. Dentre os vários parâmetros que caracterizam o sistema, a avaliação apresentada explora a capacidade do sistema em sobrepor comunicação com computação, técnica que tem papel fundamental em ambientes nos quais os custos de comunicação são elevados.*

1 Introdução

Aglomerados de PCs ligados por tecnologia de rede local constituem uma solução de baixo custo e de bom desempenho, além de apresentar facilidade de expansão e atualização do *hardware*. Nestes ambientes os custos de comunicação são relativamente elevados, e têm forte influência no desempenho global das aplicações. Uma compreensão adequada dos custos envolvidos na comunicação é essencial para o projeto de aplicações paralelas eficientes.

O desempenho exposto pela biblioteca de alto nível para as aplicações resulta da combinação de diversos componentes, tais como dispositivos de rede, protocolos e biblioteca de alto nível. Se analisado com uma granularidade mais fina, diversos componentes podem ser adicionados à lista, tornando a análise ainda mais complexa.

O uso de modelos abstrai parte da complexidade do sistema, e auxilia no projeto de aplicações bem como na avaliação e comparação de diferentes sistemas. Na avaliação apresentada neste trabalho, foi adotado um modelo com enfoque na comunicação baseado no *LogP* e em algumas de suas extensões. Dentre os parâmetros que caracterizam o sistema, é avaliada a oportunidade de sobreposição da comunicação com computação exposta pelo sistema.

Na avaliação foram empregados três programas (*microbenchmarks*) para medição dos parâmetros que caracterizam o sistema: *NetPIPE* [15], *LogP Benchmark* [7] e *Perftest* [14]. Novos programas de teste foram implementados e incorporados à estrutura do *Perftest*, em particular para medição da oportunidade de sobreposição da comunicação com computação e sobrecarga de processamento.

Dois *kernels* científicos, *FFT* [2] e *Radix Sort* [10] são usados para avaliar diferentes técnicas de sobreposição da comunicação com computação. Em [9] é avaliado o uso de múltiplos *threads*

para sobrepor fases de comunicação com computação. Neste trabalho foram desenvolvidas novas versões dos *kernels FFT* e *Radix Sort* que empregam primitivas não-bloqueantes para promover sobreposição.

Duas implementações do padrão *MPI* são usadas na avaliação: *MPICH2* e *OPENMPI*. Também é avaliado o desempenho obtido com uso do protocolo LLC em substituição ao TCP/IP como camada de transporte da biblioteca *OPENMPI*.

O texto está organizado da seguinte forma. A Seçào 2 descreve o ambiente no qual foram efetuadas as medições. A Seçào 3 introduz o modelo usado para balizar os experimentos. A Seção 4 discute as métricas sob avaliação, sendo que as Seções 4.1, 4.2 e 4.3 apresentam os resultados das medições relativas a largura de banda efetiva, latência e sobrecarga na comunicação (*overhead*). A Seção 4.4 contém resultados sobre a possibilidade de sobreposição da comunicação com a computação, e a Seção 5 mostra os resultados da execução paralela de dois núcleos de aplicações paralelas. Nossas conclusões são mostradas na Seção 6.

2 Ambiente computacional

O ambiente computacional usado para os experimentos é referenciado como *Nautilus*, e consiste de um aglomerado de PCs mantido pela Universidade Federal do Paraná. Para a análise apresentada no que se segue foi usado um conjunto de 10 estações do Nautilus ligadas por uma rede Ethernet de 1Gbps. Cada estação possui um processador Athlon 64 de 2GHz com 2GB de memória *RAM*, e um adaptador de interface de rede *Marvel Yukon Gigabit Ethernet*. Em todas as estações o dispositivo de rede está ligado ao barramento *PCI* de 32-bits a 66Mhz.

Na avaliação foram usadas as bibliotecas *OPENMPI* (`http://openmpi.org`) e MPICH2 (`http://www-unix.mcs.anl.gov/mpi/mpich2`). 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*.

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 [8] – padrão IEEE 802.2 *Logic Link Control-LLC* [11].

Em [17], 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, que é usado nesta avaliação de desempenho.

3 Modelo

O modelo LogP [6, 5] da execução de programas paralelos 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 como proposto originalmente define parâmetros que representam os custos de comunicação para mensagens pequenas e de tamanho fixo [7]. Extensões ao modelo LogP foram propostas para contemplar mensagens grandes, tais como *LogGP* [1] e *LogP Parametrizado* [13, 14]. O modelo *LogGP* [4] é uma extensão do *LogP* e introduz um novo parâmetro (G) que captura o custo por byte para mensagens grandes, que são transmitidas com algum suporte especial de *hardware* tal como DMA.

Ao usar o modelo para expressar o custo exposto por bibliotecas de alto nível, há diversos fatores que podem alterar o custo de comunicação de acordo com o tamanho da mensagem. Por exemplo, as implementações do padrão MPI usam protocolos distintos para transissão das mensagens "grandes" e "pequenas", criando assim novos intervalos nos quais o custo é diferenciado. O *LogP Parametrizado* [12] representa os parâmetros do modelo em função do tamanho da mensagem $(L(m), o(m), g(m))$, em que m é o tamanho da mensagem. Desta forma o modelo é capaz de representar as variações observadas para diferentes tamanhos de mensagem sem a necessidade de introduzir novos parâmetros.

Em [3] 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 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 empregado nesta avaliação de desempenho usa a abordagem do *LogP Parametrizado* e a latência fim-a-fim (EEL) como definida em [3]. O parâmetro (L) é usado para representar a latência de uma mensagem pequena, e corresponde ao tempo de viagem da mensagem na rede. Os parâmetros do modelo são listados abaixo.

$EEL(m)$ 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 $1/m$ corresponde à largura de banda para mensagens de tamanho m.

L Latência de transporte de uma mensagem pequena.

P Número de processos.

4 Programas de teste

Nesta seção são descritos alguns parâmetros e os testes necessários para sua medição no sistema. Na definição dos testes destaca-se a importância de considerar a semântica e aspectos da implementação das primitivas de comunicação para produzir medidas mais precisas. Os testes comparam o desempenho de diferentes primitivas de comunicação, das bibliotecas *OPENMPI* e *MPICH2*, e também do protocolo LLC em substituição ao TCP/IP. Também é avaliado o

impacto do uso de um *thread* de progresso, recurso da biblioteca *OPENMPI* para permitir o progresso da comunicação com o uso de primitivas não-bloqueantes. Devido a grande variedade de métricas, a seguir são apresentados alguns dos principais resultados. Uma análise mais completa é apresentada em [16].

4.1 Largura de banda

Mesmo um parâmetro tão difundido como a *largura de banda*, ou *vazão*, pode ser interpretado de diferentes maneiras, afetando o resultado da medição. É preciso diferenciar a largura de banda atingida com o envio de uma única mensagem, da largura de banda sustentada com o envio de mensagens consecutivas.

A vazão alcançada com o envio de uma única mensagem, ou mensagens não-consecutivas, pode ser medida com um teste do tipo ping-pong. A metade do tempo do ping-pong corresponde ao custo de envio de uma mensagem, e o inverso do tempo determina a largura de banda nesta medida.

A largura de banda sustentada com mensagens consecutivas pode ser medida com um teste no qual um processo envia muitas mensagens, e um outro processo apenas as recebe. No modelo LogP, a largura de banda sustentada com mensagens consecutivas é a recíproca do parâmetro g. Como no padrão MPI as primitivas de envio podem completar antes da recepção da mensagem pelo destinatário, a medida deve abranger o tempo decorrido até que a última mensagem seja recebida.

4.1.1 Resultados

A ferramenta NetPIPE implementa testes de largura de banda com as duas abordagens. Os testes unidirecional e bi-direcional implementados na ferramenta são baseados no tempo de duração do ping-pong. No teste chamado de *stream* é medida a largura de banda com o envio de mensagens consecutivas.

O gráfico da Figura 1 mostra 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.

Para alguns tamanhos de mensagem ocorrem quedas acentuadas na largura de banda, destacando-se da projeção esperada do gráfico. Estas anomalias estão relacionadas ao uso do dispostivo Marvel Yukon com o *driver* sk98lin para Linux [1]. Desconsiderando as medidas erráticas, o desempenho obtido com o protocolo LLC apresenta ganho em relação ao TCP/IP da ordem de 7% para mensagens com até 256kB e de 12% para mensagens com 16kB.

O *thread* de progresso quando configurado na biblioteca *OPENMPI* é usado apenas na transmissão de mensagens longas, acima de 64kB na configuração padrão. Os testes mostram que o thread acrescenta uma sobrecarga da ordem de 7% para mensagens de 64kB a 1MB. Maiores detalhes podem ser encontrados em [16].

[1] Estas anomalias já foram reportadas anteriormente em estudo disponível em `http://www.digit-life.com/articles2/gig-eth-64bit/gig-eth-64bit-apr2004-p1-3.html`. O driver sk98lin foi substituído nos kernels mais recentes, a partir da versão 2.6.16 pelo driver sky2

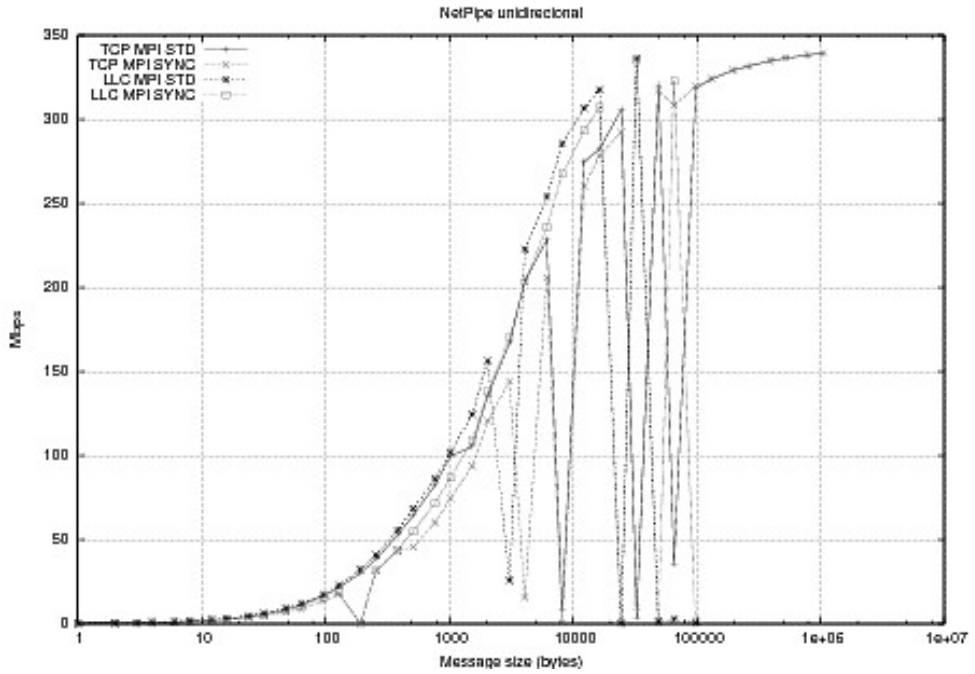


Figura 1: Resultados do NetPIPE para mensagens variando de 1 byte a 1MB.

4.2 Latência

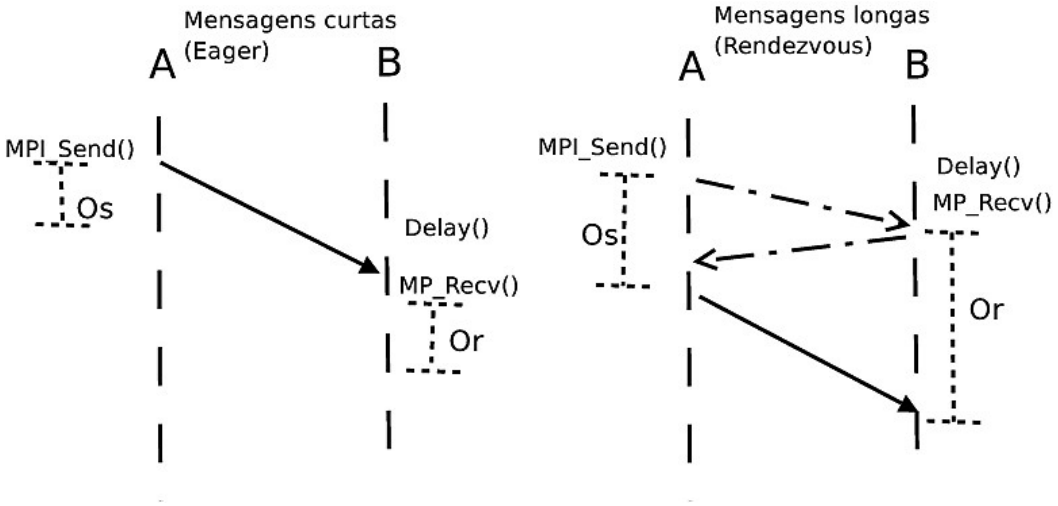
A latência, conforme empregada neste trabalho, corresponde ao tempo consumido na transmissão de uma mensagem entre dois processos. A latência pode ser medida através de um teste do tipo ping-pong, em que a metade da duração do ping-pong corresponde à latência fim-a-fim.

4.3 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. 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* [14]. Nos testes implementados pelo *LogP Benchmark*, o *overhead* é medido tomando a duração da primitiva de comunicação, de envio ou recebimento.

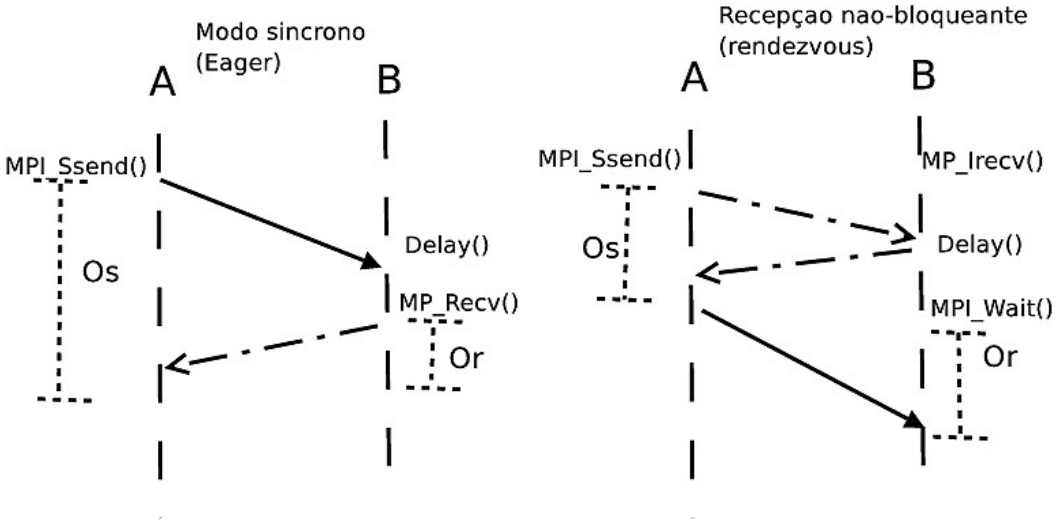
A Figura 2(a) mostra o esquema usado pelo *LogP Benchmark* para medir os parâmetros quando são usadas mensagens curtas (protocolo *eager*), e a Figura 2(b) o esquema para mensagens longas (protocolo *rendezvous*). Quando uma mensagem curta é transmitida com modo de comunicação padrão, a mensagem é copiada 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, o que pode implicar em carga de trabalho da CPU – e portanto em mais *overhead* de comunicação.

Na transmissão de mensagens curtas com modo de comunicação síncrono, a primeira mensagem com a solicitação de envio contém também parte dos dados da mensagem – ou toda mensagem se esta for suficientemente pequena. A Figura 3(a) mostra o esquema usado pelo *LogP Benchmark*



(a) Mensagem curta (b) Mensagem longa

Figura 2: 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.



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

Figura 3: 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.

quando os dados da mensagem são transmitidos com a mensagem de controle do protocolo. O esquema da Figura 3(b) mostra a medida efetuada quando é usada uma primitiva de recepção não-bloqueante e mensagens longas. Nos casos mostrados na Figura 3, o parâmetro Os inclui o tempo de negociação, que não corresponde necessariamente à carga de trabalho da CPU.

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

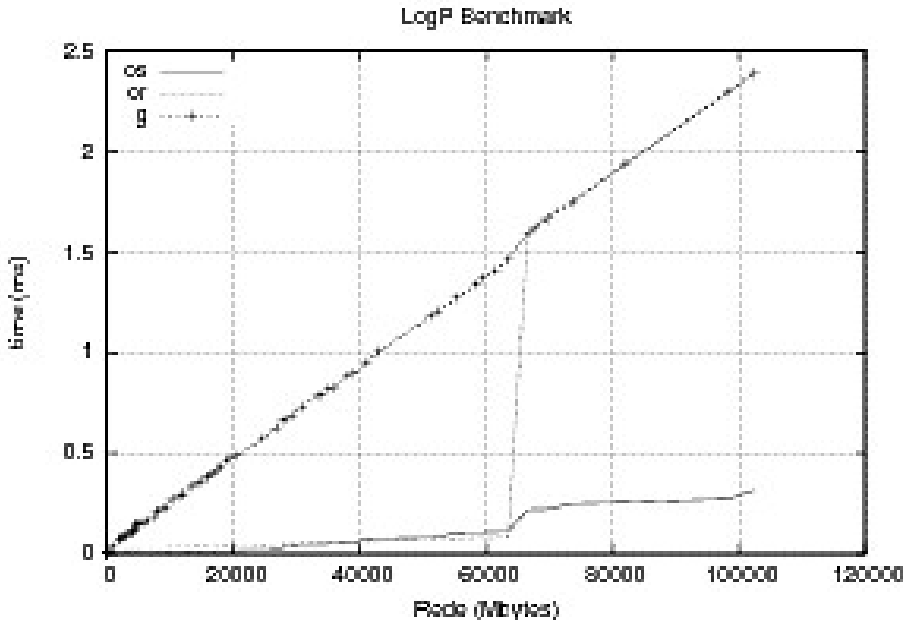


Figura 4: Resultados obtidos com o LogP Benchmark

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* intermediário – tempo que é ignorado pela medida.

A Figura 4 mostra os resultados medidos com primitiva bloqueante – também 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.

Um teste para medir a sobrecarga em operações de envio e recepção é descrito em [3], 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 *de iniciação* (`MPI_Isend`) e a primitiva associada *de conclusão* (`MPI_Wait`) para completar a operação, como descrito na Figura 5. Quando não há computação inserida entre as primitivas de comunicação a medida corresponde ao parâmetro g ou *gap*. A mesma estrutura pode ser usada para medir a sobrecarga de recebimento.

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

Figura 5: Teste de sobrecarga com primitivas não-bloqueantes.

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.

Um novo programa de teste foi desenvolvido com base na estrutura do *Perftest* seguindo esta abordagem para medir a sobrecarga de processamento. Este teste é o mesmo usado para medir a sobreposição da comunicação em mensagens consecutivas, que é apresentado na seção 4.4.3.

4.4 Sobreposição da 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 [9]. 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*.

4.4.1 Ping-pong

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.

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. 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.

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.

O gráfico da Figura 6 mostra os resultados da execução do teste de sobreposição para mensagens pequenas, entre 1 byte e 2kB obtidos com a biblioteca *OPENMPI*. O gráfico mostra no eixo X a quantidade de computação que pode ser sobreposta entre o envio da mensagem e a resposta do ping-pong, em função do tamanho da mensagem.

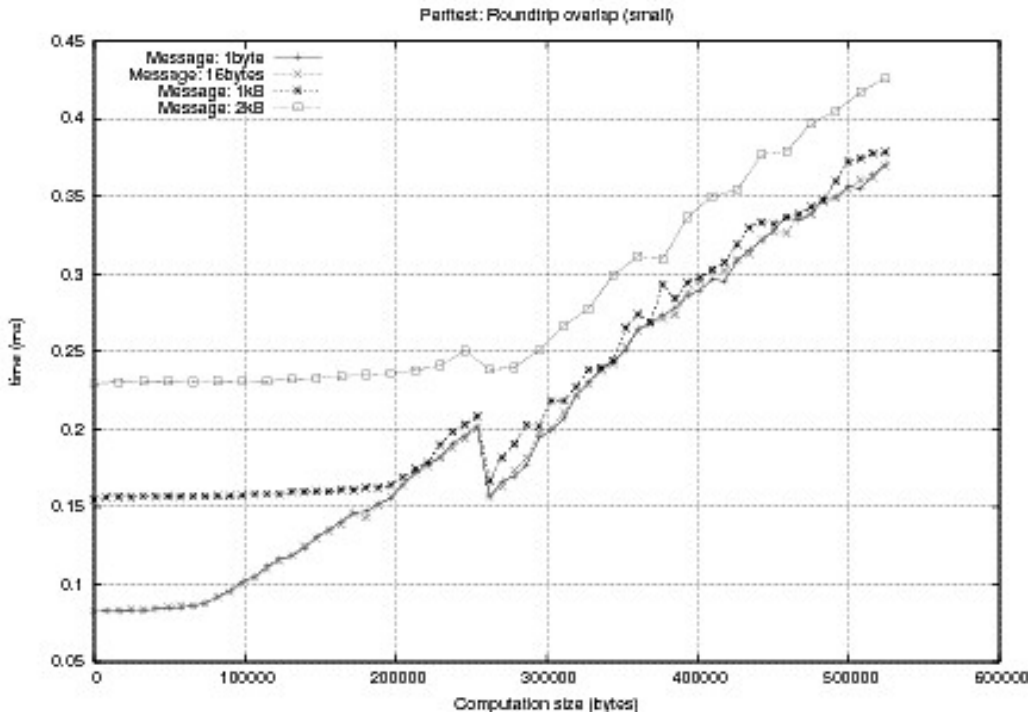


Figura 6: Sobreposição com padrão de comunicação ping-pong e mensagens entre 1 byte e 2kB.

4.4.2 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.

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*. Os resultados mostram que no ambiente analisado a comunicação não progride mesmo com o uso do *thread* de progresso na biblioteca *OPENMPI*, como mostra o gráfico da Figura 7.

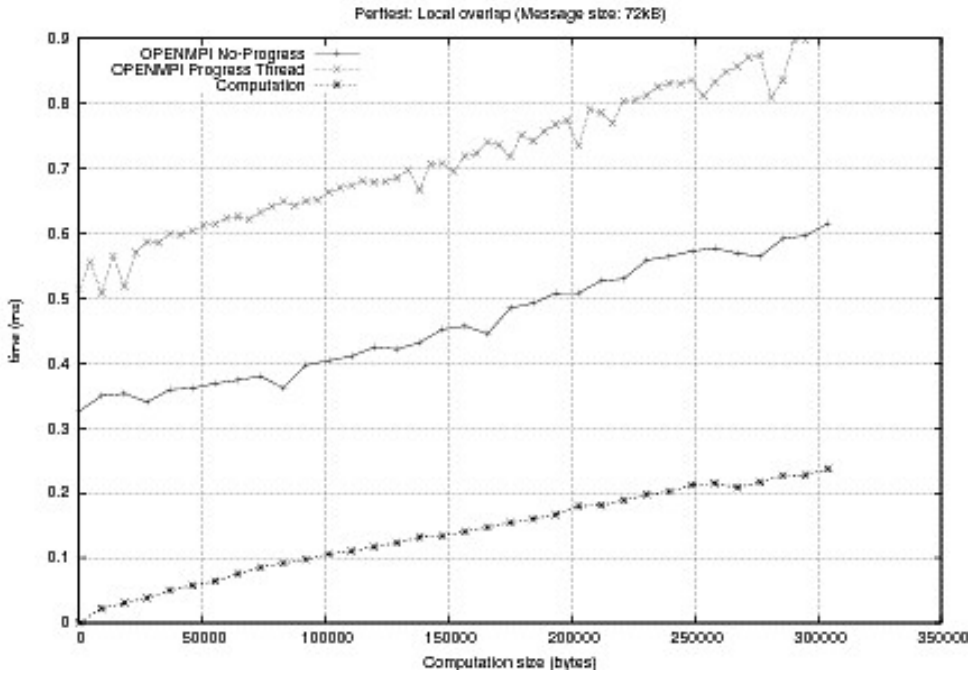


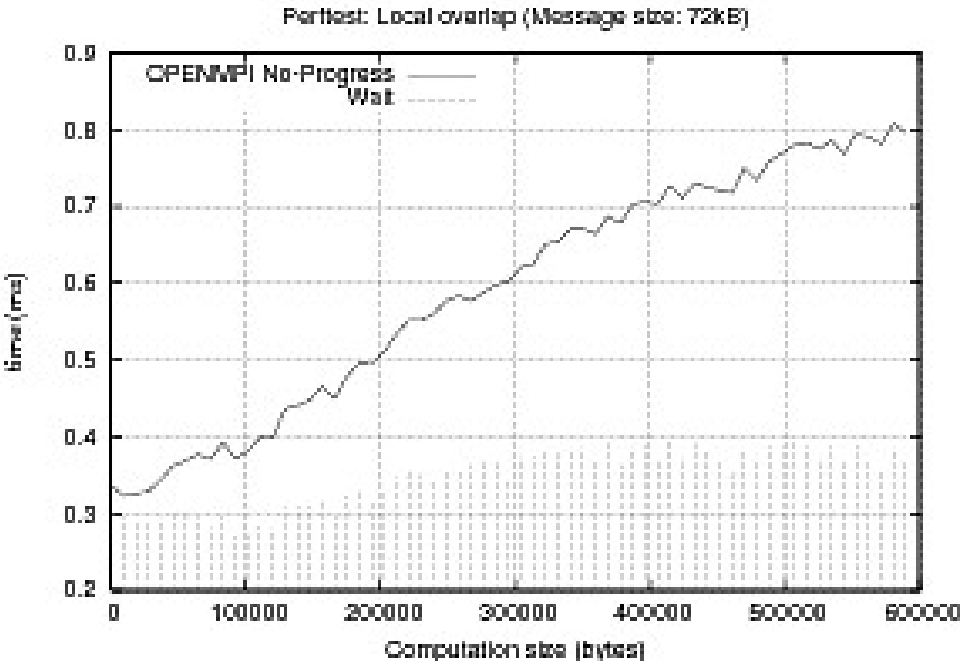
Figura 7: Teste de sobreposição local com e sem o Thread de Progresso da biblioteca OPENMPI para mensagens de 72kB.

Para que seja possível avaliar o efeito do *thread* de progresso, um novo teste foi realizado em outro aglomerado que conta com estações bi-processadas ligadas por rede Ethernet de 1Gbps. Neste outro ambiente há oportunidade de sobreposição local com o uso do *thread* de progresso, como mostra a Figura 8. As barras verticais representam o tempo gasto na primitiva que espera o término do envio da mensagem. No gráfico da Figura 8(a) o tempo consumido pela primitiva de término não diminui, o que indica que a comunicação não progrediu em paralelo com a computação inserida entre as primitivas de envio e de término. Na Figura 8(b) observa-se que o tempo consumido na primitiva de término diminui gradativamente quando é usado o *thread* de progresso.

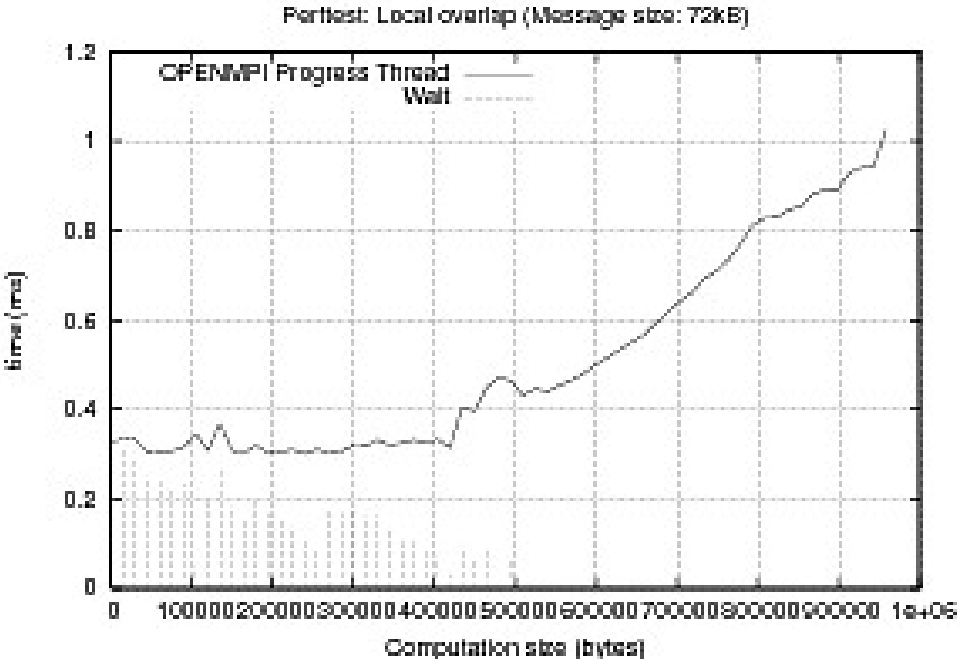
4.4.3 Gap

Quando um processo envia mensagens consecutivas, ele fica sujeito à capacidade de injetar mensagens na rede, que é representada pelo *gap* (g) no modelo. Quando o parâmetro g não é dominado pelo *overhead* (Os), existe 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



(a) Sem *Thread* de progresso



(b) Com *Thread* de progresso

Figura 8: Teste de sobreposição local – em um ambiente que não apresenta oportunidade de sobreposição (a), e em um ambiente que apresenta oportunidade de sobreposição (b). Escalas no eixo X são diferentes.

e acréscimos subsequentes na quantidade de computação aumentem o tempo do teste. O teste de sobreposição do *gap* também expõe o *overhead* de processamento, que pode ser representado por $g - C'$, onde C' corresponde à quantidade de computação sobreposta, ou escondida.

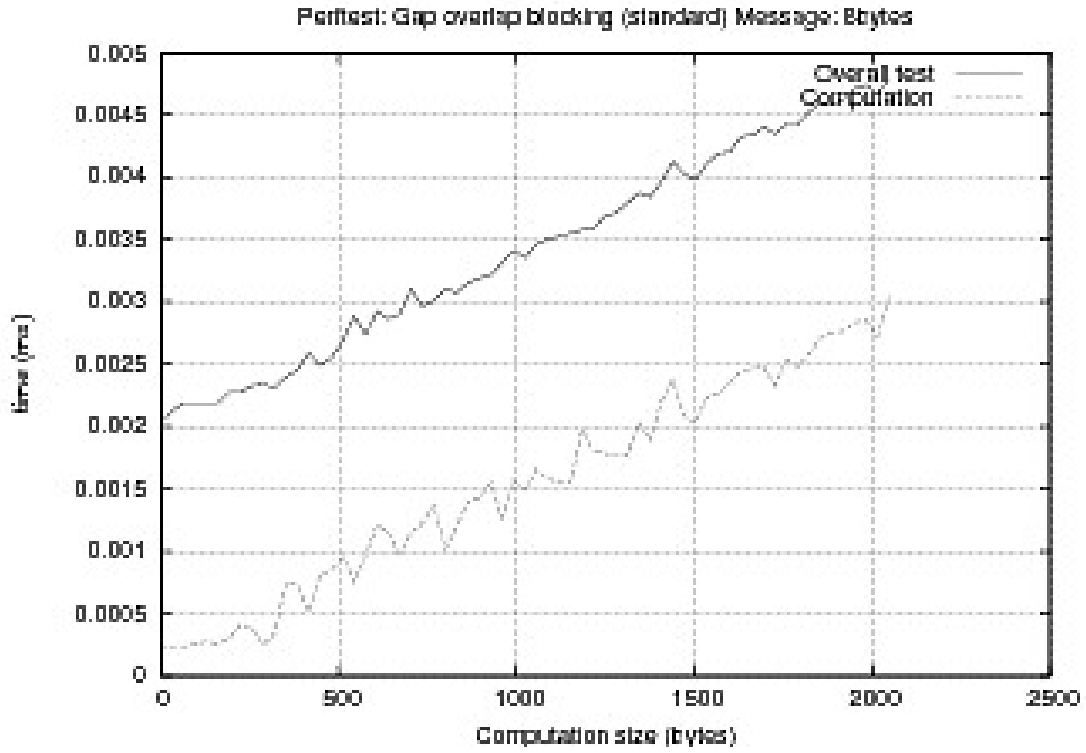


Figura 9: Teste de sobrecarga e sobreposição do gap no envio de mensagens com 8 bytes com primitiva de envio bloqueante e modo de comunicação padrão.

As Tabelas 1 e 2 mostram o resultado do teste de sobrecarga de envio e de recepção, respectivamente, para mensagens de 56kB. São mostrados os resultados obtidos com as bibliotecas *OPENMPI* e MPICH2, além do resultado que ignora a sobreposição obtido com o LogP Benchmark para fins de comparação.

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

Tabela 1: Sobrecarga no envio de mensagens de 56kB com biblioteca OPENMPI (TCP) e MPICH2. A coluna C' representa o tempo consumido com a computação sobreposta.

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

Tabela 2: 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.

O gráfico da Figura 9 mostra o resultado obtido com mensagens de 8kB usando a biblioteca *OPENMPI*. Para mensagens de 8kB não há sobreposição, o que indica que o gap é dominado

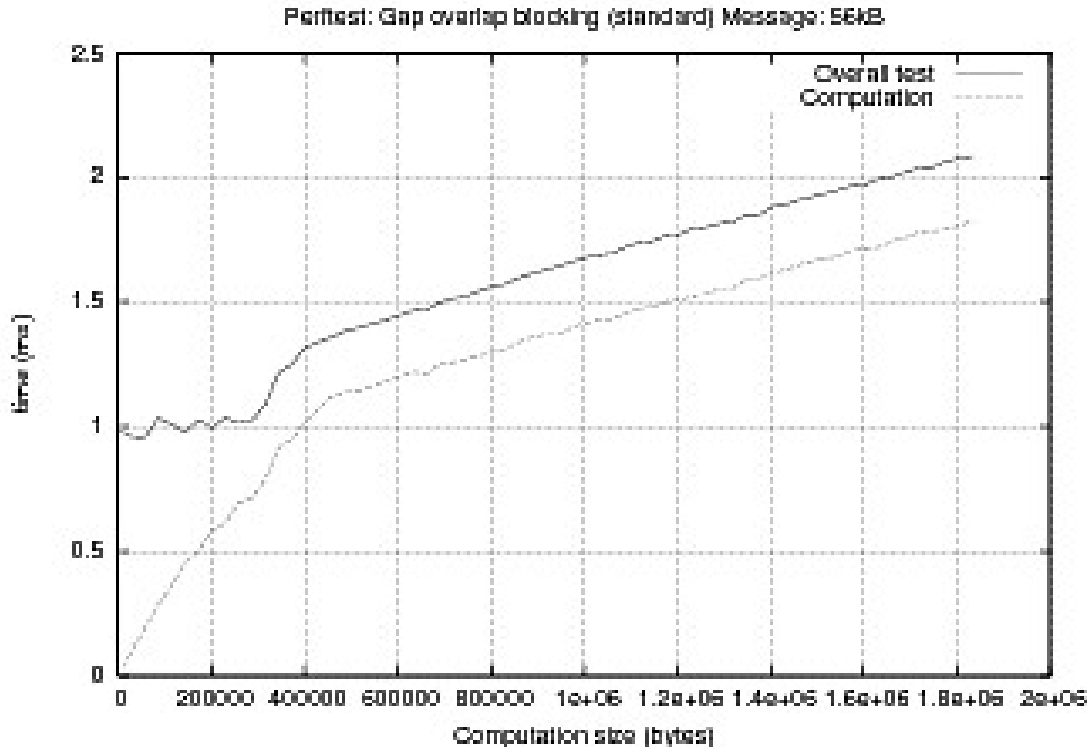


Figura 10: Teste de sobrecarga e sobreposição do gap no envio de mensagens com 56kB

pelo *overhead*. Nos resultados obtidos com mensagens de 56kB, mostrados na Figura 10, há sobreposição da comunicação, de forma que o *overhead* não domina a medida do gap.

5 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.

5.1 FFT

A implementação do *kernel* FFT identificada como *original* foi usada como base para o trabalho desenvolvido em [9], que descreve uma nova implementação com uma abordagem chamada aqui de 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 [9].

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. No início do algoritmo a matriz de entrada é transposta na estrutura intermediária. No segundo passo são realizadas FFTs unidi-

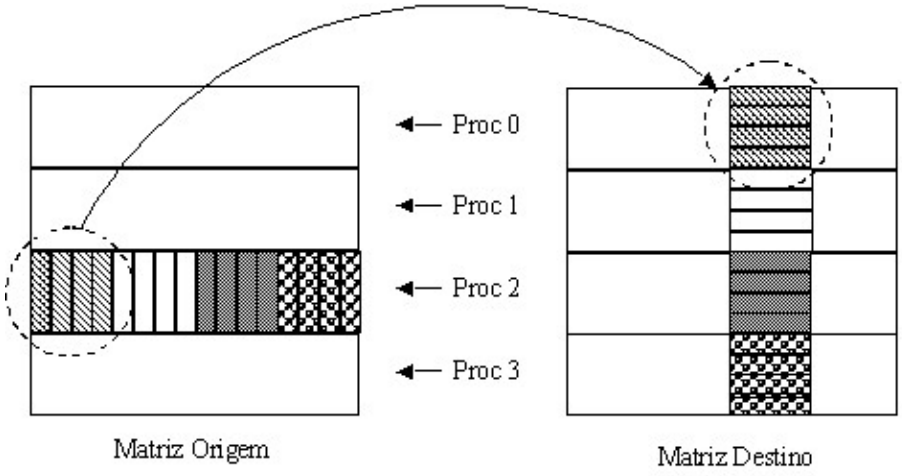


Figura 11: Transposição da matriz.

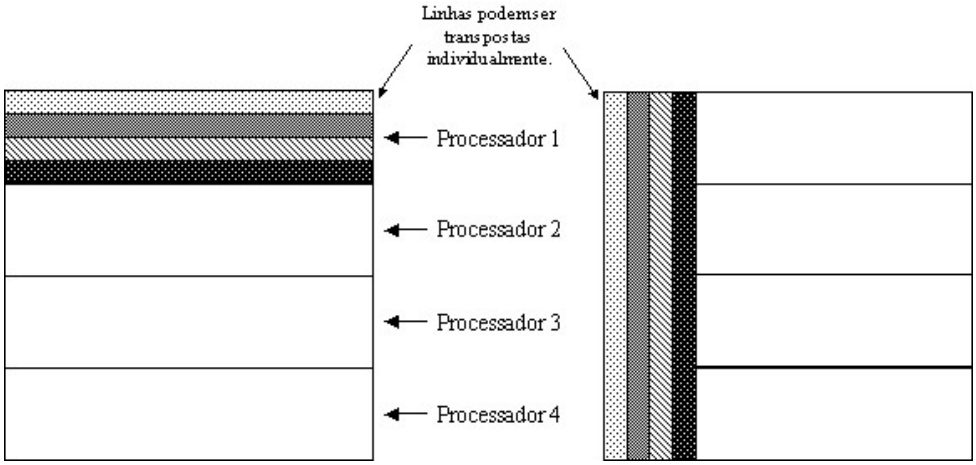


Figura 12: Transposição linha-a-linha da matriz.

mensionais 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 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 11 mostra o passo de transposição da matriz[2].

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*. Em [9], o autor propõe 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

[2]As Figuras 11 e 12 foram retiradas de [9].

original

```
Transpose();
FFT1DOnce();
TwiddleOneCol();

Transpose();
FFT1Donce();

Transpose();
```

linha-a-linha

```
for(linha=0; linha < linhas_proc; linha++)
    Transpose(linha);
    FFT1DOnce(linha);
    TwiddleOneCol(linha);

for(linha=0; linha < linhas_proc; linha++)
    Transpose(linha);
    FFT1Donce(linha);

for(linha=0; linha < linhas_proc; linha++)
    Transpose(linha);
```

Figura 13: Passos do algoritmo FFT na versão original (à esquerda) e na implementação linha-a-linha (à direita).

do estágio do algoritmo, como representado na Figura 12. Ainda em [9] o autor descreve a implementação de uma versão *multi-threaded* do algoritmo que executa concorrentemente as fases de computação e comunicação, com base na abordagem linha-a-linha.

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, como descrito na Figure 13. 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.

5.1.1 Resultados

Os testes foram executados com 2, 4 e 8 processos, e o tamanho N do problema foi escalado. Considerando $N = 2^m$, foram executados testes com os seguintes valores de m: 16, 18, 20, 22 e 24. O algoritmo faz uso de mensagens grandes na fase de transposição, que na abordagem linha-a-linha é dado por $2 * (\frac{\sqrt{N}}{P}) * 8$, em que N é o tamanho do problema, P é o número de processos e 8 corresponde ao tamanho em bytes de cada número complexo da entrada.

A Figura 14 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).

A versão original tem melhor desempenho com entrada de tamanho 2^{16}, como pode ser observado na Figura 15. Com entradas de tamanho maior o desempenho piora em relação às outras implementações (que realizam a transposição linha-a-linha) – como mostra o gráfico da Figura 16.

5.2 Radix Sort

O *kernel Radix Sort* [10] implementa a ordenação de números inteiros, que é usada em uma grande quantidade de aplicações, como sistemas de gerenciamento de banco de dados. Cada

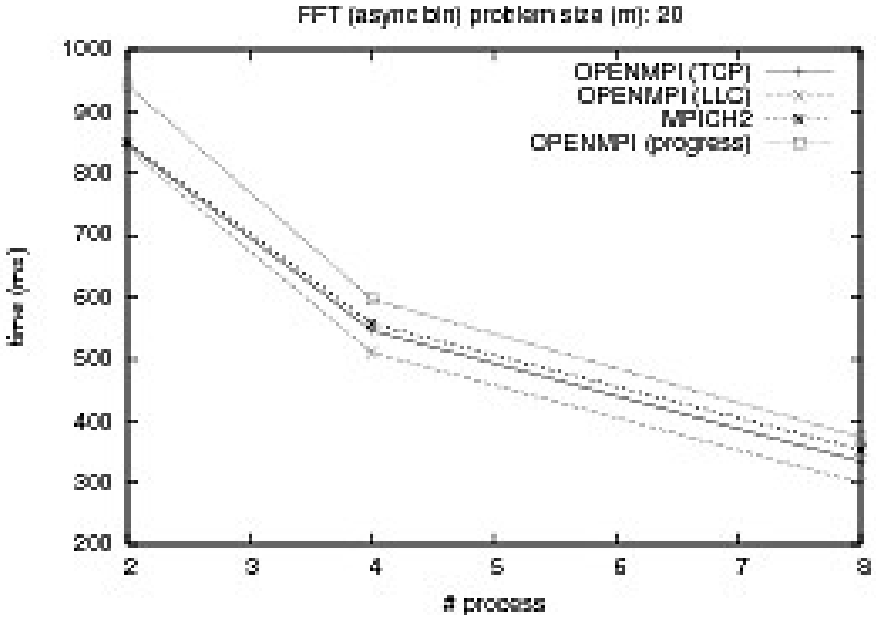


Figura 14: Resultado do FFT versão linha-a-linha com primitiva não-bloqueante e problema de tamanho 2^{20} números complexos.

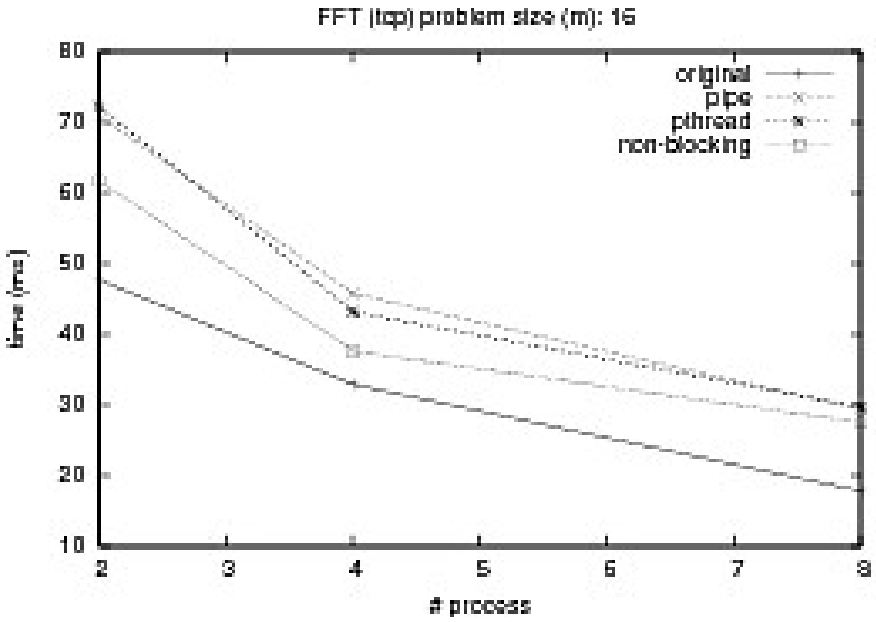


Figura 15: Resultados da execução do FFT com variação do número de processadores para entrada de tamanho 2^{16}.

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 17. 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

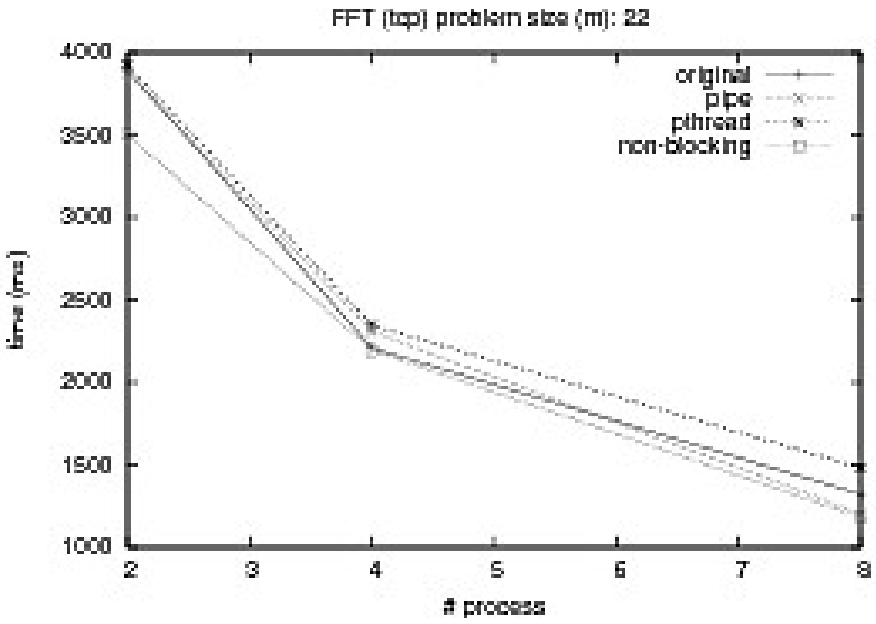


Figura 16: Resultados da execução do FFT com variação do número de processadores para entrada de tamanho 2^{22}.

distribuídas entre os processos.

Em [9], o autor descreve 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 é responsável pelo cálculo da distribuição das chaves e transmissão das chaves aos destinos, e um thread auxiliar é dedicado a recepçãodas chaves enviadas por outros processos. Neste algoritmo pode haver sobreposição de comunicação com comunicação, bem como de comunicação com computação.

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.

5.2.1 Resultados

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 18 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.

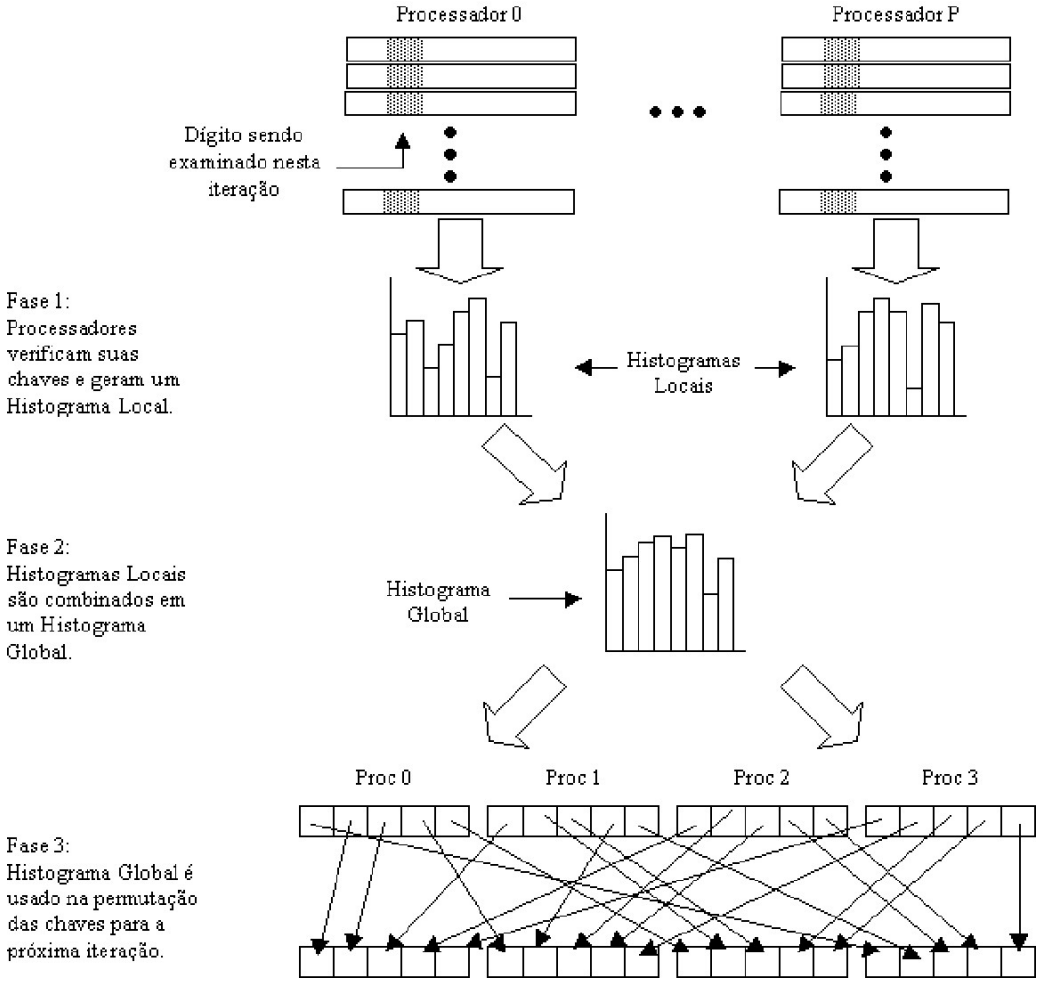


Figura 17: Fases do algoritmo do Radix Sort.

Os resultados mostram também que usar um *buffer* pequeno para que a mensagem possa ser transmitida em um único pacote Ethernet, como implementado originalmente, não apresenta o melhor desempenho. Neste teste, o melhor desempenho foi alcançado com mensagens de 16kB. 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*, por este motivo o gráfico não mostra resultados da versão original com mensagem maior que 8kB.

O Radix apresenta um padrão de comunicação irregular. O uso de um *thread* dedicado a recepção de mensagens apresentou ganho da ordem de 50% em relação a implementação original no teste com 8 processadores.

6 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

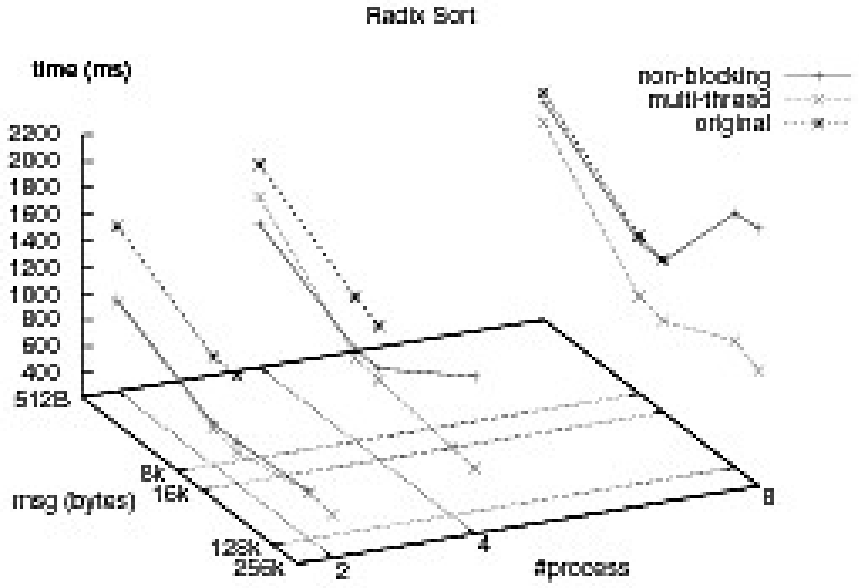


Figura 18: Comparação do desempenho das implementações do Radix Sort para 2, 4 e 8 processos, e mensagens de tamanho 128, 2048, 4096.

estudado.

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.

Embora no ambiente analisado o sistema não seja muito eficiente em expor oportunidade de sobreposição às aplicações, os resultados mostram que as técnicas de sobreposição proporcionaram ganho de desempenho nos testes com os *kernels FFT* e *Radix Sort*. Espera-se que em outros ambientes o sistema seja capaz de expor maior oportunidade de sobreposição, aumentando o ganho obtido com o uso das técnicas de sobreposição da comunicação.

Referências

[1] A Alexandrov, M F Ionescu, K E Schauser, and C Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

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

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

[4] 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.

[5] D E Culler, R M Karp, D A Patterson, A Sahay, E E Santos, K E Schauser, R Subramonian, and T von Eicken. LogP: a practical model of parallel computation. *Commun. ACM*, 39(11):78–85, 1996.

[6] D E Culler, R M Karp, D A Patterson, A Sahay, K E Schauser, E E Santos, R Subramonian, and T von Eicken. LogP: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[7] D E Culler, L T Liu, R P Martin, and C Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, February 1996.

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

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

[10] G E Blelloch and others. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc ACM Symp on Parallel Algorithms and Architectures*, pages 3–16, 1991.

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

[12] F Ino, N Fujimoto, and K Hagihara. LogGPS: a parallel computational model for synchronization analysis. In *Proc of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 133–142. ACM Press, 2001.

[13] T Kielmann, H E Bal, and S Gorlatch. Bandwidth-efficient collective communication for clustered wide area systems. In *Proc 14th Int Symposium on Parallel and Distributed Processing*, page 492. IEEE Computer Society, 2000.

[14] T Kielmann, H E Bal, and K Verstoep. Fast measurement of LogP parameters for message passing platforms. In *Proc IPDPS'00 Workshop on Parallel and Distributed Processing*, pages 1176–1183. Springer-Verlag, 2000. `http://www-unix.mcs.anl.gov/mpi/mpptest/`.

[15] Q Snell, A Mikler, and J Gustafson. Netpipe: A network protocol independent performace evaluator. In *In IASTED International Conference on Intelligent Information Management and Systems*, June 1996. `http://www.scl.ameslab.gov/netpipe/`.

[16] Marcelo Loyola Stival. Avaliação de desempenho em aglomerados de PCs ligados por ethernet. Dissertação de mestrado, Universidade Federal do Paraná (UFPR) - Depto de Informática, Setembro 2006. `http://www.inf.ufpr.br/roberto/dissMarcelo.pdf`.

[17] 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 2006. `http://www.inf.ufpr.br/~roberto/dissLeslie.pdf`.