



Escola Politécnica da USP

Departamento de Engenharia da Computação e Sistemas Digitais

LI KUAN CHING

ANÁLISE E PREDIÇÃO DE DESEMPENHO DE  
PROGRAMAS PARALELOS EM REDES DE  
ESTAÇÕES DE TRABALHO

Tese Apresentada à Escola Politécnica da  
Universidade de São Paulo para Obtenção  
do Título de Doutor em Engenharia

SÃO PAULO  
2001

LI KUAN CHING

ANÁLISE E PREDIÇÃO DE DESEMPENHO DE  
PROGRAMAS PARALELOS EM REDES DE  
ESTAÇÕES DE TRABALHO

Tese Apresentada à Escola Politécnica da  
Universidade de São Paulo para Obtenção  
do Título de Doutor em Engenharia

SÃO PAULO  
2001

© Copyright by LI KUAN CHING, 2001

LI KUAN CHING

ANÁLISE E PREDIÇÃO DE DESEMPENHO DE  
PROGRAMAS PARALELOS EM REDES DE  
ESTAÇÕES DE TRABALHO

Tese Apresentada à Escola Politécnica da  
Universidade de São Paulo para Obtenção do  
Título de Doutor em Engenharia

Área de Concentração:  
Engenharia de Computação e Sistemas Digitais

Orientadora:  
Prof. Dra. Liria Matsumoto Sato

SÃO PAULO  
2001

**Li, Kuan Ching**

Análise e Predição de Desempenho de Programas Paralelos em Redes de Estação de Trabalho. São Paulo, 2001. 113 p.

Tese (Doutorado) — Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia da Computação e Sistemas Digitais.

1. Análise de Desempenho 2. Predição de Desempenho 3. Sistema de Estações de Trabalho 4. Sistema Distribuído I. Universidade de São Paulo, Escola Politécnica, Departamento de Engenharia da Computação e Sistemas Digitais. II.t.

“Se, em algum cataclisma, todo o conhecimento científico fosse destruído e só uma frase passasse para a próxima geração de criaturas, qual conteria o maior número de informações no menor número de palavras ?”

Richard Feynman (1918-1988).

AOS MEUS PAIS,

*YITA E I-CHIAO*

# Agradecimentos

A **Deus**, pelos incentivos, direcionamento e pelas oportunidades que ELE tem me proporcionado ao longo do caminho que tenho percorrido.

À Prof. Dra. **Liria Matsumoto Sato** pelos acompanhamentos, orientação e dedicação durante todo o período da elaboração do trabalho da minha formação acadêmica. Seus constantes incentivos, amizade e conversas foram fundamentais desde 1992, quando nós nos conhecemos.

Ao Dr. **Jean-Luc Gaudiot**, pelas inúmeras discussões, conversas e incentivos.

Ao Prof. Dr. **João José Neto**, pelas conversas, apoios e sugestões. Ao Dr. **Cyro Patarra** e Prof. Dr. **Siang Wun Song**, pelos incentivos nos anos iniciais da graduação, que foram fundamentais posteriormente para o futuro acadêmico na minha vida.

Ao amigo **Marcelo Mecchi Morales**, pelos inúmeros apoios. Duas pessoas desconhecidas, encontrando num determinado lugar, e um café em novembro/2000, foi suficiente para estabelecer uma grande parceria e amizade.

Ao amigo **José Craveiro da Costa Neto** e **Mário Donato Marino**, pela amizade, conversas e incentivos.

A todos os colegas do laboratório LASB/PCS, em especial **Antônio Misaka**, **Edson Midorikawa**, **Hélio Marci**, **Gisele Craveiro** e **Jean Laine**, pela convivência.

Aos colegas **Deyse Regina Szücs**, **Kátia Guimarães**, **Flávio Keidi Miyazawa**, **Hélio Crestana Guardia** e **José Eduardo Moreira**, pela amizade.

À **Helen Chen**, pela compreensão, estímulo e apoio desde o momento que nós nos conhecemos. Por estar presente em todos os momentos, conversando, compreendendo, dando novos significados à vida.

Aos meus irmãos, **Kuan-Ju** e **Yen-Ying**, pelos incentivos, amizade, apoios e presença em todos os momentos.

Aos meus **pais**, "Mr. Hello" e "Dona Li", pelos apoios familiares e constantes incentivos, sem eles, não teria alcançado mais esta vitória. Parabéns a eles por terem conseguido prover condições para que este trabalho pudesse ser completado.

À **família Lai** e **família Chen** de Taichung, Taiwan (ROC), muito obrigado pelos apoios e constantes incentivos.

A todos aqueles que direta ou indiretamente colaboraram no desenvolvimento deste trabalho.



## Sumário

SUMÁRIO.....	I
LISTA DE FIGURAS .....	V
LISTA DE GRÁFICOS .....	VII
LISTA DE TABELAS.....	VIII
LISTA DE ABREVIATURAS E SIGLAS .....	IX
RESUMO .....	X
ABSTRACT.....	XI
<b>CAPÍTULO 1.....</b>	<b>1</b>
<b>INTRODUÇÃO .....</b>	<b>1</b>
1.1 OBJETIVOS DO TRABALHO .....	4
1.2 MOTIVAÇÃO.....	5
1.3 ORGANIZAÇÃO DA TESE.....	6
<b>CAPÍTULO 2.....</b>	<b>8</b>
<b>CONSIDERAÇÕES INICIAIS.....</b>	<b>8</b>
2.1 COMPUTAÇÃO PARALELA E DISTRIBUÍDA.....	8
2.1.1 Modelos de Fluxo de Controle.....	9
2.1.2 Modelos de Organização de Memória.....	10
2.1.3 Modelos de Programação Paralela.....	10
2.2 INTERFACE DE PASSAGEM POR MENSAGENS MPI.....	11
2.2.1 Implementação do MPI – versão LAM .....	13
2.2.2 Tipos de Dados de Mensagens.....	13
2.2.3 Operações de Comunicação Bloqueantes Ponto a Ponto .....	14
2.2.4 Operações de Comunicação Não-Bloqueantes Ponto a Ponto .....	15
2.2.5 Operações de Comunicação Coletiva.....	15
2.2.5.1 Operações para Movimentação de Dados .....	16

2.2.5.2	Operações para Controle de Processos .....	17
2.2.5.3	Operações para Cálculo Global.....	17
2.3	GRAFO DE TEMPO (TIMING GRAPH / <i>T-GRAPH</i> ) .....	18
2.3.1	<i>Representação de Programas Utilizando T-graph</i> .....	20
2.3.1.1	Representação da Estrutura Estática de Um Programa .....	20
2.3.2.	<i>Caminhos de Execução e Tempo de Execução</i> .....	22
2.3.3.	<i>Caracterização de Caminhos de Execução</i> .....	23
2.3.3.1	Circulação .....	24
2.3.3.2	T-graph e Circulações .....	24
2.4	COMENTÁRIO .....	26
<b>CAPÍTULO 3.....</b>		<b>27</b>
<b>ANÁLISE E PREDIÇÃO DE DESEMPENHO .....</b>		<b>27</b>
3.1	INTRODUÇÃO.....	28
3.2	TÉCNICAS DE AVALIAÇÃO.....	29
3.2.1	<i>Modelagem Analítica</i> .....	29
3.2.2	<i>Modelagem Estatística</i> .....	30
3.2.3	<i>Modelos Empíricos</i> .....	30
3.2.4	<i>Simulação</i> .....	30
3.2.5	<i>Modelos Híbridos</i> .....	31
3.3	TRABALHOS RELACIONADOS .....	31
3.3.1	<i>Trabalho Desenvolvido por P. Puschner e A. Schedl</i> .....	32
3.3.2	<i>Trabalho desenvolvido por M. Gubitoso</i> .....	33
3.3.3	<i>Trabalho Desenvolvido por D. F. Vrsalovic, D. P. Siewiorek, Z. Z. Segall e E. F. Gehringer</i> .....	34
3.3.4	<i>Trabalho Desenvolvido por A.J.C. van Gemund</i> .....	37
3.3.5	<i>Trabalho Desenvolvido por J. Landrum, J. Hardwick e Q.F.Stout</i> .....	38
3.3.6	<i>RSIM (the Rice Simulator for ILP Multiprocessors)</i> .....	39
<b>CAPÍTULO 4.....</b>		<b>42</b>
<b>METODOLOGIA PARA ANÁLISE E PREDIÇÃO DE DESEMPENHO .....</b>		<b>42</b>
4.1	DEFINIÇÃO DE CLASSES DE GRAFOS .....	44

4.1.1	<i>Definição da Classe de Grafos DP*Graph</i> .....	45
4.1.2	<i>Definição da Classe de Grafos T-graph*</i> .....	46
4.2	REPRESENTAÇÃO DE BAIXO NÍVEL .....	47
4.3	REPRESENTAÇÃO DE ALTO NÍVEL.....	49
4.4	ESQUEMA DA METODOLOGIA PROPOSTA .....	50
4.5	MODELAGEM DA APLICAÇÃO PARA O CÁLCULO DOS TEMPOS DE EXECUÇÃO ..	52
4.5.1	<i>Chaveador (Switch)</i> .....	53
4.5.1.1	<i>Arquitetura do Chaveador (Switch)</i> .....	54
4.5.2	<i>Um exemplo de Cálculo do Tempo de Execução</i> .....	55
4.5.3	<i>Análise da Operação de Comunicação Send/Receive</i> .....	58
4.5.3.1	<i>Análises em Função de Tamanho de Mensagem e Número de Nós de Processamento</i> .....	60
4.5.4	<i>Análise da Operação de Comunicação Broadcast</i> .....	61
4.5.4.1	<i>Análises em Função de Tamanho de Mensagem e Número de Nós de Processamento</i> .....	64
4.5.5	<i>Análise da Operação de Comunicação Scatter</i> .....	67
4.5.5.1	<i>Análises com Variação no Tamanho das Mensagens e Nós de Processamento</i> .....	71
4.5.6	<i>Análise da Operação de Comunicação Gather</i> .....	72
4.5.6.1	<i>Análises Variando Número de Mensagens e Número de Nós de Processamento</i> .....	75
4.5.7	<i>Análise da Operação de Comunicação Reduce</i> .....	76
4.5.7.1	<i>Análises Variando Tamanho de Mensagens e Número de Nós de Processamento</i> .....	76
4.5.7.2	<i>Caso Particular: Operação de Comunicação All Reduce</i> .....	77
4.5.8	<i>Análise da Operação de Comunicação All-to-All</i> .....	78
4.6	PREDIÇÃO DE DESEMPENHO.....	79
4.7	ESTUDO DE CASO: MODELAGEM DA APLICAÇÃO IS (NAS) .....	81
4.7.1	<i>Procedimento RANK da aplicação IS</i> .....	84
4.7.1.1	<i>Representação do Procedimento</i> .....	84
4.7.1.2	<i>Análises do Procedimento RANK</i> .....	85
4.7.1.3	<i>Estudos de Predição de Desempenho</i> .....	87

<b>CAPÍTULO 5.....</b>	<b>89</b>
<b>IMPLEMENTAÇÃO, RESULTADOS E DISCUSSÃO .....</b>	<b>89</b>
5.1 AMBIENTE DE TESTE .....	90
5.2 RESULTADOS COM OPERAÇÃO DE COMUNICAÇÃO <i>SEND</i> .....	90
5.3 RESULTADOS COM OPERAÇÃO <i>BROADCAST</i> .....	93
5.3.1 <i>Outros Gráficos</i> .....	96
5.4 RESULTADOS COM OPERAÇÃO DE COMUNICAÇÃO <i>SCATTER</i> .....	96
5.5 RESULTADOS OBTIDOS COM PROCEDIMENTO RANK, DO PROGRAMA IS/NAS	98
<b>CAPÍTULO 6.....</b>	<b>101</b>
<b>CONCLUSÕES E TRABALHOS FUTUROS .....</b>	<b>101</b>
6.1 CONTRIBUIÇÕES .....	102
6.2 TRABALHOS E PROJETOS FUTUROS .....	103
<b>REFERÊNCIAS BIBLIOGRÁFICAS .....</b>	<b>105</b>
<b>APÊNDICE I.....</b>	<b>1</b>
<b>PROGRAMAS DE BENCHMARK.....</b>	<b>1</b>
1. PROGRAMA DE BENCHMARK NAS .....	1
2. PROGRAMA DE BENCHMARK MPBENCH.....	3
Referência .....	3
<b>APÊNDICE II.....</b>	<b>4</b>
<b>PROGRAMA IS / NAS.....</b>	<b>4</b>

## Lista de Figuras

<i>Figura 1. Classificação de Ambientes Paralelos.....</i>	<i>9</i>
<i>Figura 2. Arquitetura de um Sistema por Passagem de Mensagens. ....</i>	<i>12</i>
<i>Figura 3. Algumas operações coletivas.....</i>	<i>18</i>
<i>Figura 4. Notação de T-graph. ....</i>	<i>19</i>
<i>Figura 5. Representação em T-graph de uma construção típica de linguagem de programação.....</i>	<i>21</i>
<i>Figura 6. Modelagem de um processador no RSIM. ....</i>	<i>40</i>
<i>Figura 7.Arquitetura do sistema paralelo. ....</i>	<i>41</i>
<i>Figura 8. Metodologia de Análise e Predição Proposta Neste Trabalho.....</i>	<i>44</i>
<i>Figura 9. Elementos de Representação da classe DP*Graph.....</i>	<i>45</i>
<i>Figura 10. Elementos de Representação da Classe T-graph*.....</i>	<i>46</i>
<i>Figura 11(a). Listagem de um programa paralelo-Caso1. ....</i>	<i>47</i>
<i>Figura 11(b). Representação do programa - Caso1 - apresentado na figura 11(a). ....</i>	<i>47</i>
<i>Figura 12(a). Listagem de um programa paralelo-Caso2. ....</i>	<i>48</i>
<i>Figura 12(b). Representação do programa - Caso2 - utilizando DP*Graph.....</i>	<i>48</i>
<i>Figura 13(a). Listagem de um programa paralelo-Caso3. ....</i>	<i>49</i>
<i>Figura 13(b). Representação do programa - Caso3 - utilizando DP*Graph.....</i>	<i>49</i>
<i>Figura 14. Representação em Alto Nível da listagem apresentada na figura 11(a). ....</i>	<i>50</i>
<i>Figura 15. Representação em Alto Nível das listagens apresentadas nas figuras 12(a) e 13(a).....</i>	<i>50</i>
<i>Figura 16. Tempo máximo de execução de um programa paralelo com 5 processos. ....</i>	<i>53</i>
<i>Figura 17. Arquitetura Interna de um Switch.....</i>	<i>55</i>
<i>Figura 18. Cálculo do Tempo de Execução.....</i>	<i>55</i>
<i>Figura 19. Explicação da nomenclatura fornecida. ....</i>	<i>58</i>
<i>Figuras 20(a) e 20(b). Casos de estudo da operação MPI_Send.....</i>	<i>58</i>
<i>Figura 21. Representação para o estudo de casos apresentados pelas figuras 20(a) e 20(b). ....</i>	<i>59</i>
<i>Figura 22. Esquema da movimentação dos dados da Operação MPI_Bcast(). ....</i>	<i>62</i>
<i>Figura 23(a). Estudo da operação de comunicação MPI_Bcast(). ....</i>	<i>62</i>

<i>Figura 23(b). Estudo da operação de comunicação MPI_Bcast().</i> .....	63
<i>Figura 24. Representação em DP*Graph do estudo da operação MPI_Bcast().</i> .....	63
<i>Figura 25. Ilustração dos passos de execução do comando Broadcast dentro de um switch.</i> .....	66
<i>Figura 26. Esquema da movimentação dos dados da Operação MPI_Scatter().</i> .....	68
<i>Figura 27(a). Caso 1 de estudo da operação de comunicação MPI_Scatter.</i> .....	68
<i>Figura 27(b). Caso 2 de estudo da operação de comunicação MPI_Scatter.</i> .....	69
<i>Figura 28. Representação em DP*Graph do estudo da operação MPI_Scatter().</i> ....	69
<i>Figura 29. Esquema da movimentação dos dados da Operação MPI_Gather().</i> .....	72
<i>Figura 30(a). Caso 1 de estudo da operação de comunicação MPI_Gather.</i> .....	73
<i>Figura 30(b). Caso 2 de estudo da operação de comunicação MPI_Gather.</i> .....	73
<i>Figura 31. Representação em DP*Graph para o estudo da operação MPI_Gather().</i> .....	74
<i>Figura 32. Representação do Procedimento RANK do programa de avaliação IS.</i> .	85

## Lista de Gráficos

<i>Gráfico 1. Gráfico de tempo de execução da operação MPI_Send().</i>	<i>91</i>
<i>Gráfico 2. Gráfico do resultado da Operação de Comunicação Send.</i>	<i>92</i>
<i>Gráfico 3. Gráfico com tempos de execução da operação Broadcast.</i>	<i>93</i>
<i>Gráfico 4. Gráfico de tempos da operação Broadcast.</i>	<i>95</i>
<i>Gráfico 5. Gráficos com tempos de operação do comando MPI_Bcast.</i>	<i>96</i>
<i>Gráfico 6. Gráfico com tempos de operação do comando MPI_Scatter.</i>	<i>97</i>
<i>Gráfico 7. Gráfico com tempos de operação e de predição do comando MPI_Scatter.</i> .....	<i>98</i>
<i>Gráfico 8. Gráfico com tempos de operação e de predição do Procedimento RANK/IS/NAS.</i>	<i>100</i>

## Lista de Tabelas

<i>Tabela 1. Resultados Experimentais e de Predição da Operação Send.....</i>	<i>92</i>
<i>Tabela 2. Resultados Experimentais e de Predição da Operação MPI_Bcast.....</i>	<i>95</i>



## Lista de Abreviaturas e Siglas

- **Programa de Benchmark**: refere-se aos programas de aplicações desenvolvidos para avaliar desempenho de máquinas paralelas.
- **Cluster of Workstations, Network of Workstations, NOW** (aglomerado de estações de trabalho): refere-se ao conjunto de nós de processamento de um sistema distribuído.
- **CSMA/CD** (*carrier sense multiple access - collision detect*): protocolo de transmissão da *Ethernet*, mais especificamente, protocolo de acesso ao meio.
- **ILP** (*Instruction-Level Parallelism*): Paralelismo ao Nível de Instrução.
- **LAM** (*Local Area Multicomputer*): uma das versões mais conhecidas de implementação do MPI. (<http://www.lam-mpi.org>).
- **MAXT** (*maximum execution time*): tempo máximo de execução.
- **MPI** (Message Passing Interface): Interface de Passagem por Mensagens.
- **MPICH**: uma versão bastante conhecida de implementação do MPI, desenvolvida pelo Laboratório Nacional de Argonne (Illinois, EUA). ([www.anl.org/mpich](http://www.anl.org/mpich)).
- **Switch**: chaveador utilizado na conexão de PCs, com o propósito de formar uma rede.
- **T-graph /TG** (Timing Graph): Grafo de tempo.
- **Tempo de Latência** (Latency Time): tempo que leva a requisição de memória para atravessar a rede de interconexão. Multiprocessadores com um número grande de processadores tendem a ter complexas redes de interconexão e tempos de latência altos.

## Resumo

Processamento distribuído tem sido utilizado amplamente para melhorar o desempenho de aplicações com alta demanda computacional. Diferentes arquiteturas e topologias distribuídas têm sido pesquisadas e utilizadas para prover o alto desempenho, proporcionando assim o recurso necessário para a exploração do paralelismo presente nas aplicações. A facilidade para construir sistemas computacionais de alto desempenho a partir de estações de trabalho interligadas através de redes de alta velocidade, aliada ao custo relativamente baixo e ao crescente avanço da tecnologia de circuitos integrados, possibilita a montagem de redes de computadores de baixo custo para a execução de aplicações paralelas.

Devido a este fato, diversos sistemas de software para redes de estações têm sido desenvolvidos, visando a integração dos componentes distribuídos para a agregação das suas capacidades de processamento. No entanto, o processo de desenvolvimento de aplicações é complexo e difícil, dado que são necessários identificar o paralelismo existente nestas aplicações, e providenciar as comunicações necessárias.

Neste trabalho, é apresentada uma proposta de metodologia de análise e predição de desempenho de programas paralelos, implementados com interface de passagem de mensagem (MPI), em ambientes de redes de estações de trabalho. É definida neste trabalho uma extensão da classe de grafos de tempo  $T-graph$ , denominado  $T-graph^*$ , que representa, em alto nível, os programas paralelos instrumentados com MPI no nível de grafos. Com a construção de um grafo nesta classe, é possível conhecer o fluxo da execução do programa, do ponto de vista algorítmico. Ainda, é definida uma outra classe de grafos, denominada  $DP^*Graph$ , que representa os programas paralelos com alto grau de detalhes, como mostrar de forma clara pontos de ocorrência de comunicação entre os nós de processamento do sistema computacional. Em paralelo com recursos e técnicas de modelagem analítica, são definidas estratégias para a avaliação de desempenho dos sistemas computacionais envolvidos. Uma vez obtidas as representações em grafos do programa paralelo e junto com as modelagens já refinadas e definidas, é possível efetuar avaliações necessárias e obter assim predições de desempenho, baseadas em dados experimentais obtidos previamente.

Finalmente, os resultados experimentais obtidos mostram a viabilidade da metodologia definida nesta proposta, tanto a sua utilização e quanto à coerência das estratégias aplicadas neste trabalho.

## Abstract

Distributed processing has been widely used to improve the performance of applications that highly demand computational power. Different distributed architectures and topologies have been used in a search for high performance, providing further the necessary resource for the parallelism exploitation present in the applications.

The ease to build high performance computer systems, by interconnecting workstations using a high speed network, together with relatively low cost and IC technology advances, it's possible to assembly a low cost computer network for the execution of parallel applications.

Due to this fact, several applications and software systems for network of workstations have been developed, aiming the integration of distributed components for the aggregation of their processing power. Unfortunately, the process of application developing is complex and difficult, given that it is necessary identify the existing parallelism in these applications, and provide the communication needed. The control of multiple processes and their interactions are the main reasons for such complexity.

It is shown, in this work, a methodology proposal for the performance analysis and prediction of parallel programs, implemented with message passing interface (MPI) in a network of workstations environment. We define, still in this work, an extension for *T-graph* (timing graphs), named *T-graph\**, a newer class of graphs from which we can represent parallel programs with MPI functions by using timing graphs. Together with resources and analytical modeling techniques, strategies are defined for the performance evaluation of computer systems involved. Once obtained the graph representation of a parallel program, in parallel with defined and refined models designed, it is possible to proceed with necessary evaluations and from this, performance prediction data, based on the experimental data obtained previously.

Finally, experimental results obtained show the viability of the methodology proposed in this research, coherent strategies applied in this work and also, correct utilization of the techniques.

# Capítulo 1

## Introdução

Embora os avanços em tecnologia tenham contribuído com o desenvolvimento de máquinas cada vez mais rápidas e velozes, existem ainda perspectivas para o desenvolvimento de máquinas ainda mais rápidas. Na computação científica, a meta de executar aplicações de problemas "*Grand Challenge*" dentro de um limite de tempo mais razoável ainda não foi atingida. Diversas direções têm sido alvos de pesquisa, com o propósito de aumentar o poder de computação. Alguns exemplos são os processadores vetoriais, sistemas de processamento paralelo maciço (MPP) e redes de estações de trabalho (NOW) [ANDE95].

Processadores vetoriais utilizam registradores vetoriais, unidades funcionais *super pipelined* e módulos de memória altamente *interleaved* para prover alto desempenho. Estas máquinas utilizam processadores especiais e arquitetura do sistema sob medida e assim, têm um custo alto no seu desenvolvimento e uma vida útil relativamente curta. Os computadores vetoriais apresentam arquiteturas especiais que, em sua maioria, utilizam múltiplos processadores vetoriais. Devido à complexidade da arquitetura e dos componentes utilizados, o custo desse desempenho é alto.

Por outro lado, cada nó de um sistema MPP utiliza componentes já disponíveis no mercado e o sistema de computação é então a união de um número destes nós, como se fossem blocos de montar, utilizando tais componentes. Alguns dos sistemas desenvolvidos utilizando esta técnica são o sistema Intel ASCI Red e o SGI/Cray T3E. Por exemplo, o Intel ASCI Red é construído a partir de placas-mãe com dois processadores Pentium Pro. Ainda, nos sistemas MPP atuais, a única parte feita sob medida na arquitetura deste sistema é a interconexão entre os nós. Finalmente, os sistemas MPP têm demonstrado uma melhor relação custo-benefício para supercomputação comparado aos computadores vetoriais [CHEN98, STRO98].

Redes de estações de trabalho (NOW) têm vantagens sobre os sistemas MPP e os computadores vetoriais, uma vez que podem utilizar PCs / estações de trabalho conectados via uma rede de alta velocidade, atuando como um sistema distribuído de larga escala, fornecendo uma interface para programação paralela e distribuída aos usuários.

A vantagem de se utilizar sistemas NOW deve-se principalmente ao fato de que estações de trabalho de alto desempenho com microprocessadores estão disponíveis a um custo relativamente baixo. Atualmente, utilizar um sistema de estações de trabalho é um modo mais econômico para obter alto desempenho do que o uso de um sistema MPP [THEB99].

No projeto de redes de estações de trabalho, o sistema de comunicação entre os nós é extremamente importante, podendo influenciar, aumentando ou reduzindo, significativamente o desempenho. Antes do advento das redes de alta velocidade, a comunicação apresentava alta latência e pequena largura de banda, e assim, somente aquelas aplicações com paralelismo de granularidade grossa eram convenientes para computação baseada em redes. Com o desenvolvimento das redes de alta velocidade, como *ATM*, *Gigabit Ethernet* e *Myrinet* [BODE95], conseguiu-se um aumento na largura de banda das redes, atingindo a ordem de grandeza próxima à taxa apresentada entre CPU e memória. Além disso, o uso de sistemas de chaveamento permitiu a construção de NOWs com grau elevado de escalabilidade, tolerante a falhas e uma melhor relação custo/benefício.

O desenvolvimento de aplicações e sistemas paralelos de alto desempenho é um processo em constante evolução. Pode-se iniciar com modelos ou simulações,

seguido de implementação de um programa. O código é então modelado e modificado, à medida que se avança nas etapas do experimento, com o propósito de monitorar seu desempenho. Em cada uma das etapas, a pergunta-chave é: “como e quanto alterou o desempenho?” Esta pergunta surge no momento em que é proposta uma comparação entre diferentes implementações de modelos, como:

- ❑ Considerar versões de uma implementação que utilizam algoritmos diferentes, diferentes na implementação ou versão de bibliotecas de comunicação, bibliotecas numéricas ou linguagem;
- ❑ Estudos do comportamento do código variando o número ou o tipo de processadores, o tipo de rede de interconexão, o tipo de processo, o conjunto de dados de entrada ou a carga de trabalho, ou os algoritmos de escalonamento;
- ❑ Executar programas de avaliação de desempenho (programas de *benchmark*, tais como LINPACK, NAS e SPLASH-2) ou teste por regressão estatística.

A proliferação de arquiteturas multiprocessadores e do processamento paralelo e distribuído, tem contribuído para gerar um maior interesse na análise de desempenho de combinações de arquiteturas e algoritmos. As razões que contribuíram para este crescente interesse são basicamente: (1) descobrir como otimizar algoritmos para arquiteturas já existentes e (2) desenvolver e pesquisar novas arquiteturas cujas características proporcionem a execução de algoritmos com alto desempenho.

Efetuar previsão de desempenho e gerar resultado confiável constituem uma etapa fundamental para um rápido desenvolvimento de protótipos de sistemas multiprocessadores. Durante a etapa de desenvolvimento, ter dados provenientes de uma previsão de boa qualidade reduziria o número de etapas de desenvolvimento e pesquisa e, conseqüentemente, reduziria o tempo total de desenvolvimento. Concluída a construção deste protótipo, o modelo de desempenho desenvolvido pode ser utilizado e ajustado ao comportamento do sistema, preparando e gerando novos experimentos a partir da construção deste, encerrando assim o ciclo das práticas experimentais [VRSA88].

## 1.1 Objetivos do Trabalho

O principal objetivo deste trabalho é apresentar uma proposta de metodologia de análise e predição de desempenho de programas paralelos em redes de estações de trabalho, explorando recursos e técnicas da modelagem analítica e avaliações experimentais de desempenho.

Esta metodologia permitirá efetuar a análise de desempenho de uma aplicação com processamento paralelo sobre um sistema NOW. Uma aplicação é descrita, aqui neste trabalho, através de uma linguagem de alto nível, a linguagem C, e utilizando a interface de mensagens MPI.

A metodologia inclui uma etapa preliminar onde são executados programas que têm a finalidade de coletar dados relacionados aos tempos gastos pelas comunicações de dados entre nós, providas pela interface MPI, através de funções de comunicação. Estes dados refletem as características do sistema de interconexão do sistema NOW para o qual se pretende efetuar as predições de desempenho. A seguir, têm-se uma etapa onde se instrumenta o programa da aplicação, visando a coleta de tempos de execução de trechos entre duas comunicações. Um grafo é construído, na representação denominada de baixo nível, utilizando uma classe de grafos proposta, chamada *DP\*Graph*. Este grafo tem como principal característica representar trechos de código de um programa paralelo, correspondentes a cada um dos nós de processamento do sistema computacional envolvido na execução deste programa, considerando elementos seqüenciais, comunicação e pontos de sincronização. Numa etapa posterior, é possível construir um grafo de tempo para programas paralelos, denominado T-graph\*, que é uma representação de alto nível, considerando principalmente uma representação algorítmica deste programa paralelo. Percorrendo um grafo *DP\*Graph*, e utilizando os tempos correspondentes de execução coletados para os trechos presentes no ponto e os dados referentes ao custo das comunicações, calculam-se os parâmetros necessários para o cálculo que permitirá efetuar análises e predição de desempenho.

Neste trabalho, esta predição se refere à variação do número de nós do sistema de estações de trabalho e à variação do tamanho do problema, para

aplicações em que isto se aplica. Em algumas aplicações, o tamanho do problema fica restrito à quantidade de memória disponível no sistema em que são executados.

Pretende-se apresentar a metodologia acompanhada de um estudo de caso, onde os trechos entre comunicações seguem um comportamento polinomial, quanto ao tamanho do problema, e inversamente proporcional quanto ao número de nós utilizados. Os resultados obtidos são apresentados posteriormente.

A automatização da metodologia através de uma ferramenta é relevante, porém, não está inclusa neste trabalho e deverá ser tratada em sua continuidade.

## 1.2 Motivação

Processamento paralelo e distribuído tem sido utilizado para solucionar aplicações com alta demanda computacional. Diferentes arquiteturas e topologias paralelas têm sido pesquisadas e utilizadas para prover o alto desempenho, proporcionando o recurso necessário para a exploração do paralelismo presente nas aplicações [BAIL95, DONG00, LOVE93, MPBE01, NETP01, XIAO95, WOOS95].

A facilidade para construir sistemas paralelos a partir de estações de trabalho interligadas através de redes, adicionada ao custo baixo e à crescente capacidade de processamento de cada um dos nós, através do avanço da tecnologia dos componentes, tem resultado em sistemas paralelos de baixo custo para a execução de aplicações paralelas de alto desempenho.

Devido a este fato, diversos sistemas de software para redes de estações têm sido desenvolvidos, visando a integração dos componentes distribuídos para a agregação das suas capacidades de processamento. No entanto, o processo de desenvolvimento de aplicações é complexo e difícil, dado que é necessário visualizar, isolar o paralelismo existente nestas aplicações, e providenciar as comunicações necessárias. Assim, é possível identificar interações desfavoráveis com a arquitetura e, entre as tarefas executadas paralelamente, obter um entendimento detalhado da execução de um programa [BOYL88, DONG98, FOST95].



Na busca de soluções eficientes para as aplicações, torna-se necessária a análise de desempenho, incluindo sua predição e permitindo visualizar o seu comportamento em sistemas com maior número de nós e maior quantidade de memória.

Uma pergunta comum surge com frequência: se ao executar uma aplicação com oito nós obtém-se resultado  $X$ , qual seria o resultado se esta aplicação fosse executada com dezesseis nós de processamento? O resultado ideal seria que o tempo de execução fosse  $X/2$ . No entanto, devido a problemas de *overhead*, contenção na rede de interconexão, entre outros problemas, o resultado será diferente de  $X/2$ . Mas, quanto realmente será? É neste ponto que as técnicas de predição tornam-se relevantes.

### 1.3 Organização da Tese

Esta tese está organizada em seis capítulos. No capítulo 2, apresentam-se e discutem-se modelos de computação paralela e distribuída, como os de fluxo de controle, organização de memória e de programação paralela. São apresentados tópicos relacionados à Interface de Passagem por mensagens (MPI), e ainda, é introduzido *T-graph*, uma classe de grafos que apresenta estrutura concisa e definida para representação de programas seqüenciais.

Já no capítulo 3, são apresentados modelos aplicados à análise e predição de desempenho. Ainda, são apresentados diversos trabalhos relacionados e da área de pesquisa da metodologia proposta.

No capítulo 4 é apresentada a metodologia para análise e predição de desempenho proposta, as modelagens de aplicações e, ainda, são introduzidos os grafos *DP\*Graph* e *T-graph\**, classes de grafos para representação de programas paralelos instrumentados com interface de passagem por mensagens MPI. Modelagens de operações de comunicação coletiva e ponto a ponto, baseadas em comandos de comunicação MPI, são mostradas neste capítulo. E, também, o programa de benchmark IS, do conjunto de programas NAS/NASA, é apresentado; são feitas as modelagens e posteriormente, os grafos de representação deste programa são construídos.

O capítulo 5 apresenta técnicas utilizadas na implementação da metodologia para análise de trechos de código do programa paralelo. São mostradas ainda algumas das estratégias utilizadas para análise de desempenho.

Finalmente, no capítulo 6 estão presentes as conclusões e propostas de trabalhos futuros para esta linha de pesquisa.

# Capítulo 2

## Considerações Iniciais

A utilização de ambientes de computação paralela é destinada basicamente para a busca de maiores índices de desempenho na solução de problemas complexos. Estes ambientes se caracterizam pela utilização de várias unidades de processamento para a resolução de um único problema, mediante a divisão de tarefas ou de atividades computacionais entre os processadores disponíveis no sistema paralelo, onde o programa será executado.

### 2.1 Computação Paralela e Distribuída

A figura 1 mostra uma representação, proposta por Pancake [PANC96], para a identificação dos sistemas computacionais paralelos, que é mais completa do que a taxonomia de Flynn [HOCK91] apresentada anteriormente, que classifica os computadores paralelos em sistemas de memória compartilhada, denominados *multiprocessadores*, e sistemas de memória distribuída, denominados *multicomputadores*.

A taxonomia apresentada por C.M. Pancake se baseia em identificar e separar os ambientes, de acordo com três modelos de classificação: de controle, de memória e de programação.

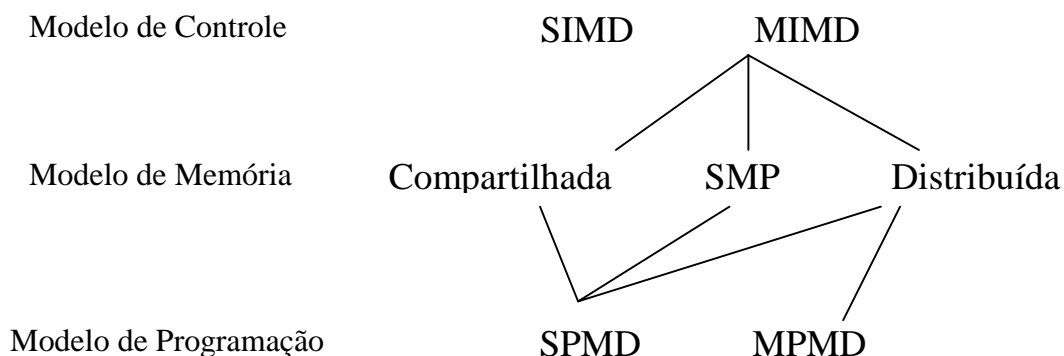


Figura 1. Classificação de Ambientes Paralelos.

### 2.1.1 Modelos de Fluxo de Controle

Este modelo representa a estrutura do sistema computacional, no que se refere à possibilidade de execução simultânea de fluxos de controle (seqüências diferentes de instruções) nos processadores.

Na classificação proposta por *Flynn*, o termo SIMD (*single-instruction, multiple-data*) significa que os sistemas computacionais executam simultaneamente as mesmas instruções de uma única seqüência sobre um conjunto de dados diferentes em cada processador. Esta categoria de sistemas computacionais, quando associada ao modelo de organização da memória, deriva outras classes de sistemas computacionais.

Ainda, de acordo com esta classificação proposta, os sistemas computacionais são pertencentes à classe MIMD (*multiple-instruction, multiple-data*) quando vários processadores ou nós de processamento cooperam para a solução de um único problema, podendo cada um dos nós de processamento ter diferentes seqüências de instruções em execução simultânea, sobre diferentes dados.

Vale observar que, quando os sistemas computacionais desta categoria são associados à classificação de modelos de organização de memória, derivam-se outras classes de sistemas computacionais.

## 2.1.2 Modelos de Organização de Memória

Segundo esta taxonomia, o sistema computacional é classificado de acordo com as possibilidades de acesso a uma dada posição de memória pelos diversos processadores do sistema.

Em um sistema com memória compartilhada (*shared memory*), todos os processadores podem acessar uma mesma posição de memória global, enquanto em sistemas com memória distribuída (*distributed memory*), cada processador possui uma memória local com acesso privativo, não havendo assim memória comum a mais de um nó de processamento. A cooperação no processamento pelos diversos processadores se dá através de uma infra-estrutura de comunicação por passagem de mensagens.

Os sistemas do tipo SMP (*Symmetric Multiprocessor*) contêm memória compartilhada por um pequeno grupo de processadores (geralmente de quatro a oito), cujos acessos a uma dada posição de memória por quaisquer dos processadores consomem a mesma quantidade de tempo. Alguns sistemas são construídos conectando-se múltiplos SMP's, cada um efetuando acessos a seu próprio conjunto de memória.

Há estudos recentes sobre sistemas computacionais que implementam o modelo de memória compartilhada sobre uma arquitetura real de memória distribuída. Tais sistemas são denominados DSM - *Distributed Shared Memory*. Para o programador e o usuário, o sistema, apesar de possuir memória distribuída, pode ser visto como se tivesse memória compartilhada.

## 2.1.3 Modelos de Programação Paralela

Esta classificação refere-se às restrições existentes no número de programas que participam da execução paralela. Os dois modelos desta classificação são:

1. Modelo Único-Programa, Múltiplos-Dados (SPMD): o processamento será executado nos diversos processadores obedecendo à mesma seqüência de instruções. Assim, se o modelo de controle for SIMD, serão processadas as mesmas instruções em cada processador simultaneamente;

se o modelo de controle for MIMD, os processadores terão cópias iguais do mesmo código objeto, mas cada processador poderá estar executando instruções diferentes, num dado momento, inclusive em segmentos diferentes do programa, dependendo das condições de desvio no fluxo de controle.

2. Modelo Múltiplos-Programas, Múltiplos-Dados (MPMD): cada processador receberá uma seqüência diferente de instruções para executar sobre dados diferentes, devendo implementar a cooperação entre seus processos através de mecanismos específicos de controle, dependendo da implementação do modelo de memória. Estes mecanismos são os principais fatores de geração de erros de construção nos programas paralelos, cuja detecção se busca no processo de depuração.

## 2.2 Interface de Passagem por Mensagens MPI

Passagem de mensagens é um paradigma de programação amplamente utilizado em computadores paralelos, especialmente Computadores Paralelos Escaláveis (*Scalable Parallel Computers*) com memória distribuída, e Aglomerados de Estações de Trabalho (NOW). Embora existam diversas variações, o conceito básico é a comunicação entre processos através de passagem de mensagens.

Nestes últimos dez anos, progressos têm sido feitos na busca de transformar diversas aplicações importantes para este paradigma. Visando a portabilidade das aplicações, em um trabalho cooperativo entre universidades e fabricantes, foi especificada a interface de passagem de mensagens MPI (*Message Passing Interface*), com o intuito de torná-la um padrão de comunicação por passagem de mensagens. O padrão MPI define a interface do usuário e a funcionalidade para uma faixa bem ampla de funções de passagem de mensagens. Ainda, versões de implementação do MPI estão disponíveis para um amplo conjunto de sistemas paralelos, que vão de Computadores Paralelos Escaláveis a Aglomerados de Estações de Trabalho. Referências da interface MPI podem ser encontrados em [GROP98,

LAUR97, MOOR01, MPI196, MPI298, MPI301, NEVI96, SCOT95, SNIR96, TABE99].

A interface de passagem de mensagens MPI oferece um padrão flexível, eficiente, portátil e prático. Os principais objetivos considerados foram:

- Projetar uma interface de programação de aplicações;
- Permitir uma comunicação eficiente;
- Permitir implementações que podem ser utilizadas num ambiente heterogêneo;
- Permitir o uso de funções em C e Fortran77 na interface;
- Prover uma interface de comunicação confiável e estável.

A arquitetura de passagem de mensagens é mostrada na figura 2.

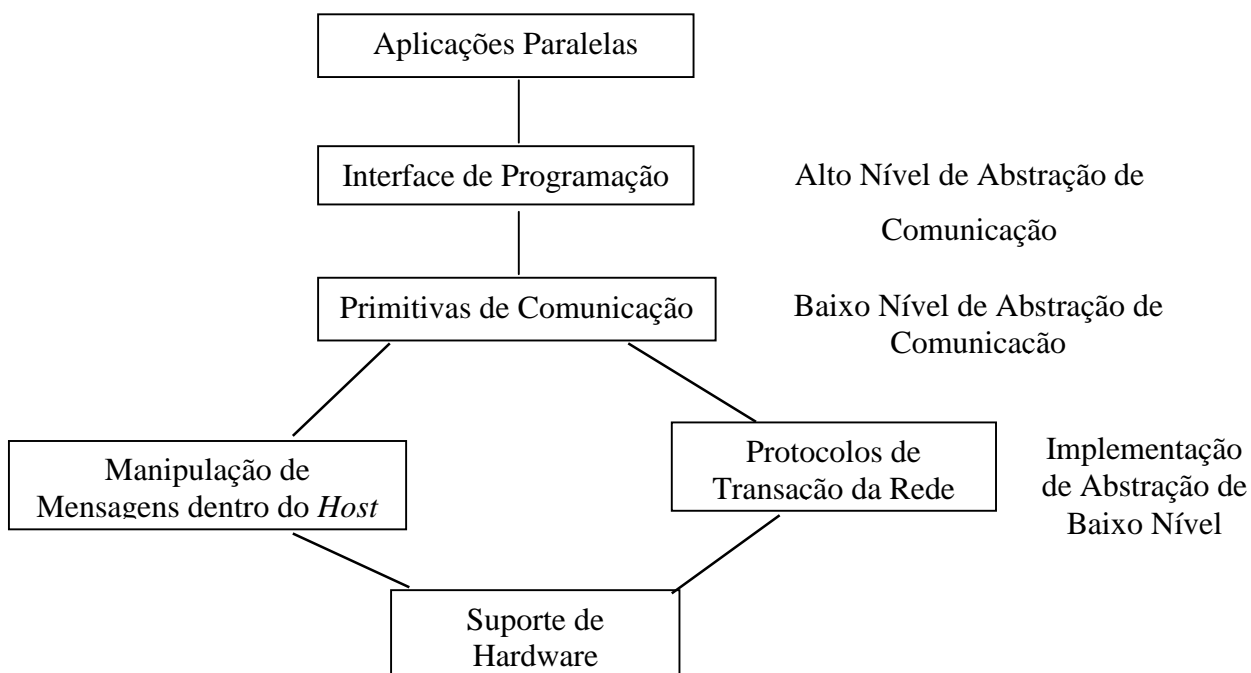


Figura 2. Arquitetura de um Sistema por Passagem de Mensagens.

## 2.2.1 Implementação do MPI – versão LAM

LAM é um ambiente de programação e sistema de desenvolvimento para um sistema multicomputador com passagem de mensagens, constituído por uma rede de estações de trabalho independentes. Dependendo das características dos nós de processamento, este sistema NOW pode ser **homogêneo**, onde todas as estações de trabalho são iguais, ou **heterogêneo**, onde pelo menos uma delas é diferente das demais. LAM oferece um suporte de ferramentas de depuração e monitoramento.

Desenvolvida pela equipe de *Ohio Supercomputer Center (The Ohio State University)*, LAM é uma das mais utilizadas entre as diversas implementações da padronização definida pela organização MPI Fórum.

Detalhes da estrutura do MPI LAM, como modularidade do micro-kernel, *daemon* de roteamento e de *buffer*, funções e sintaxes, podem ser encontradas em [BURN94, BURN95, BRUC97, GROP96].

## 2.2.2 Tipos de Dados de Mensagens

Na comunicação entre processos, quando executados programas paralelos em sistemas homogêneos ou heterogêneos, os dados contidos nas mensagens devem ser acompanhadas da informação sobre os seus respectivos tipos, pois somente assim, a estação de trabalho receptora poderá efetuar a conversão dos dados recebidos, quando necessário.

Diversos tipos pré-definidos no MPI são representados na forma MPI\_tipo-de-dado, cobrindo assim os tipos básicos de dados em grande parte das arquiteturas existentes, que são: MPI\_CHAR, MPI\_SHORT, MPI\_INT, MPI\_LONG, MPI\_UNSIGNED\_CHAR, MPI\_UNSIGNED\_SHORT, MPI\_UNSIGNED, MPI\_UNSIGNED\_LONG, MPI\_FLOAT, MPI\_DOUBLE, MPI\_LONG\_FLOAT e MPI\_BYTE. Maiores detalhes sobre a especificação de cada tipo, ver em [MPI196, MPI298, MPI301].



### 2.2.3 Operações de Comunicação Bloqueantes Ponto a Ponto

O termo bloqueante em MPI significa que a rotina não retorna ao processamento até que o *buffer* de dados associado possa ser reutilizado. Em uma mensagem ponto-a-ponto a mensagem é enviada por um processo e é recebida por um outro processo.

São oferecidos quatro modos de transmissão, que diferem entre si no que se refere à sincronização e modos de transferência de dados:

- `MPI_SEND()` (*standard*): este modo atende as necessidades de quase todas as aplicações desenvolvidas para sistemas distribuídos com o uso da interface MPI. O comando termina quando o sistema consegue colocar no *buffer* a mensagem ou quando a mensagem é recebida. O formato deste comando é:

```
MPI_Send (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);
```

- `MPI_Bsend()` (*buffered*): o comando termina quando a mensagem é colocada completamente no *buffer* alocado e reservado à aplicação ou quando a mensagem é recebida. O formato deste comando é similar àquele apresentado acima.

```
MPI_Bsend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);
```

- `MPI_Ssend()` (*synchronous*): este comando só se completa quando a mensagem é recebida.

```
MPI_Ssend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);
```

- `MPI_Rsend()` (*ready*): este comando não iniciará enquanto o outro processo com comando `MPI_RECV()` não tiver iniciado. Uma vez que o processo atinja o ponto onde consta o comando `MPI_RECV()`, o comando `MPI_Rsend()` se completará imediatamente.

```
MPI_Rsend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);
```

Quanto à recepção dos quatro modos de transmissão apresentadas acima, há somente um tipo para estes quatro modos de envio de mensagens, que é o comando `MPI_RECV()`. Os parâmetros para este comando são:

```
MPI_Recv (void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm,
MPI_Status *status);
```

## 2.2.4 Operações de Comunicação Não-Bloqueantes Ponto a Ponto

Nas operações de comunicação não-bloqueantes, um trecho de código continua a sua execução, imediatamente após ter completado a execução da operação não-bloqueante especificada.

Apesar de apresentar esta vantagem em relação às comunicações bloqueantes, não há garantia que uma aplicação utilize, de forma segura, os dados provenientes do *buffer* de mensagens, depois do retorno de uma rotina não-bloqueante. É necessário um tratamento apropriado na aplicação para obter esta garantia.

As funções oferecidas são muito similares àquelas encontradas na seção anterior, para as operações de comunicação bloqueantes, e são: `MPI_Isend()` (*standard*), `MPI_IbSEND()` (*buffered*), `MPI_ISSend()` (*synchronous*) e `MPI_IrSEND()` (*ready*), enquanto o comando para recebimento é `MPI_Irecv()`.

## 2.2.5 Operações de Comunicação Coletiva

Comunicação coletiva significa que todos os nós de processamento contribuem com informação para obter um resultado que pode ser transmitido para um ou todos os nós de processamento. Esta informação pode ser composta por dados ou sinais de controle. Em geral, neste trabalho, processo, processador ou nó de processamento (ou nó) referem-se ao mesmo conceito e serão usados indistintamente neste trabalho.

Operações coletivas podem ser classificadas em operações para controle de processos, para movimento de dados ou para cálculo global. A figura 3 mostra exemplos de operações coletivas entre N processos. Para cada operação, o lado esquerdo da figura apresenta os processos antes da operação, enquanto o lado direito mostra os processos depois da operação.

### 2.2.5.1 Operações para Movimentação de Dados

Entre as operações que realizam movimento de dados estão `broadcast`, `gather`, `scatter`, `alltogether` e `shift`.

A operação `broadcast` ocorre quando um processo envia a mesma mensagem para todos os membros do grupo. A figura 3 (a) ilustra o funcionamento desta operação. Esta operação é necessária para distribuir código e dados de um processador *host* para um conjunto de processadores. Em sistemas de memória compartilhada e memória compartilhada distribuída, esta operação pode ser usada para notificação de eventos como, por exemplo, a liberação de um *lock* para um conjunto de processadores em espera. Uma operação `broadcast` se transforma numa operação `multicast`, se apenas alguns dos membros do grupo participam na operação. No paradigma de memória compartilhada distribuída, que usa protocolos de coerência de cache baseada em diretório, `multicast` é uma operação fundamental para produzir invalidação e atualização na memória cache. Em um sistema paralelo, várias operações `multicast` podem ocorrer simultaneamente.

A operação `scatter` ocorre quando um processo envia uma mensagem diferente para cada membro do grupo. Numa operação `scatter`, um dado é distribuído por um membro do grupo (fonte) para os remanescentes deste grupo. A diferença com a operação `broadcast` é que os dados enviados pela fonte são diferentes para cada membro. A figura 3 (b) mostra o tal funcionamento.

A operação `gather` ocorre quando um processo recebe uma mensagem de cada um dos membros do grupo. Uma operação `gather` é definida como a coleta de dados de um conjunto de processos para somente um membro, em que cada processo contribui com um dado novo. A figura 3 (c) mostra esta operação. A introdução destas operações básicas é de grande importância, uma vez que podem ser estendidas e combinadas para formar operações mais complexas, como por exemplo, `broadcast todos-para-todos` (também denominado de "concatenação" ou `allgather`), onde todos os processos enviam uma mensagem para todos os membros do grupo. A figura 3 (d) mostra esta operação.

Na operação `alltogether` (troca completa), todos os membros do grupo enviam dados diferentes para os outros processos do grupo. Denominada também de "troca completa todos-para-todos", "scatter/gather" ou "todos-a-todos" (`all-to-all`). A ocorrência desta operação é a execução de uma seqüência intercalada de passos `gather` e `scatter`. A figura 3 (e) mostra esta operação.

Finalmente, na operação `shift`, cada processo envia uma mensagem para o processo seguinte e recebe uma mensagem do processo anterior, dado que os processos são enumerados pelo nó de processamento atribuído como `root`.

### 2.2.5.2 Operações para Controle de Processos

A função destas operações é de troca de sinais de controle, não de dados. A operação coletiva usada para controle de processos é a sincronização de barreira. **Sincronização de barreira** é uma operação coletiva de controle de fluxo. Esta operação define um ponto lógico no fluxo de controle de um algoritmo, no qual todos os membros do grupo devem chegar antes que seja permitido a qualquer dos processos continuar a execução.

### 2.2.5.3 Operações para Cálculo Global

As operações de comunicação coletiva, que realizam cálculo global incluem as operações de "redução" e "scan". Na operação `redução`, cada processo gera e envia para um determinado processo um operando, e sobre estes operandos, é realizada uma operação associativa e comutativa (como soma, máximo, operações lógicas bit-a-bit, etc.), e o resultado pode ser distribuído para um ou todos os processos envolvidos na operação. Quando são envolvidos todos os processos, esta operação é também denominada de "redução N/N", `allreduce` ou `gossiping`. A figura 3 (f) mostra esta operação.

A operação `scan` é uma classe de operação `reduce`, onde o membro  $i$  somente recebe os resultados de redução dos dados associados aos membros de 0 até

*i.* Esta operação é muito utilizada em sistemas de memória distribuída no contexto de processamento de imagens e aplicações de visão.

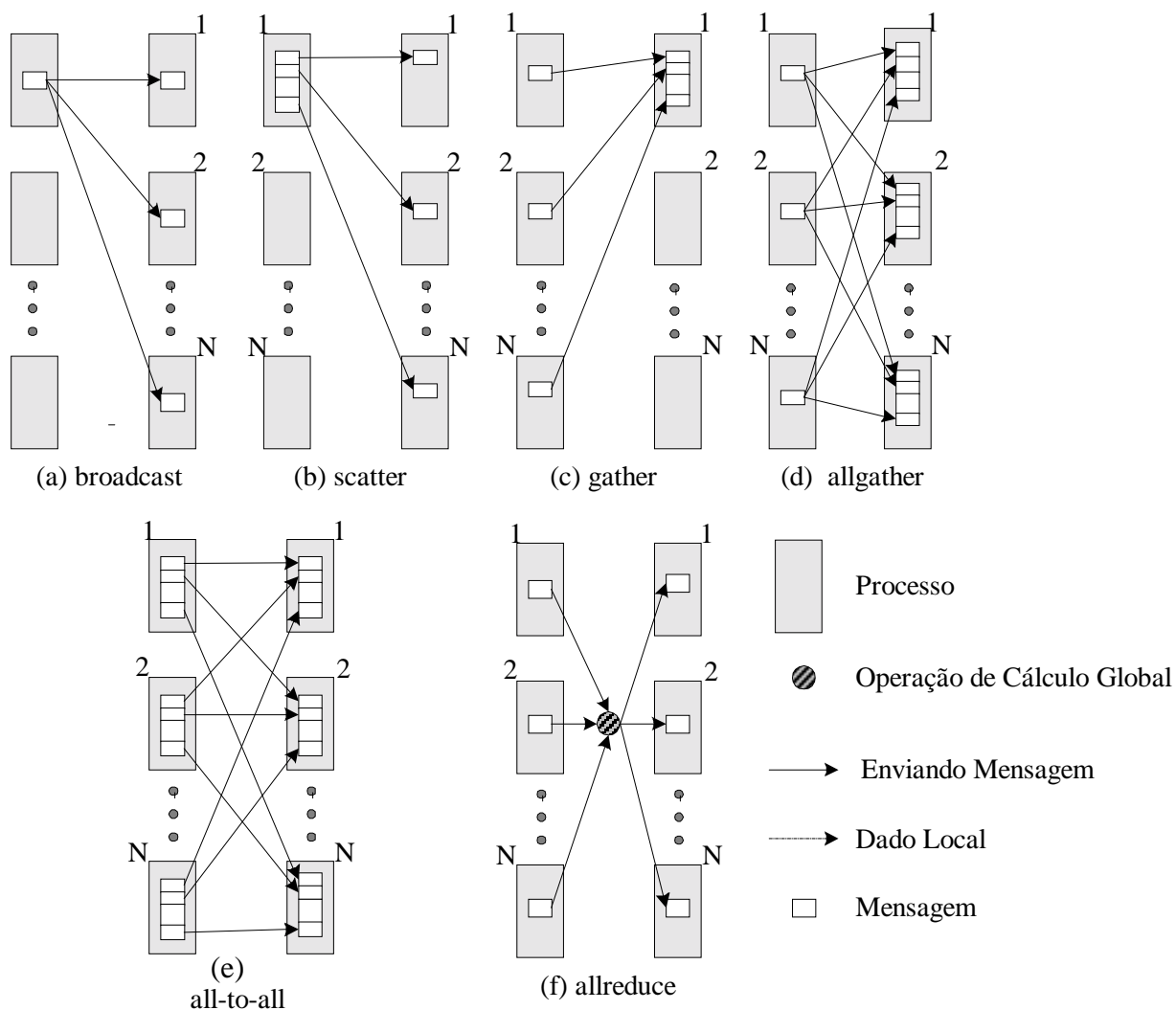


Figura 3. Algumas operações coletivas.

## 2.3 Grafo de Tempo (Timing Graph / *T-graph*)

Nesta seção, é apresentado um método pelo qual é possível descrever passos da execução paralela de um programa paralelo ou fragmentos de código paralelo. A partir desta descrição, é possível calcular o tempo de execução máximo deste programa paralelo / trecho de código. Diversos estudos e propostas têm sido feitos

utilizando esta técnica, como os apresentados por Kliegerman e Stoyenko [KLIE88], Mok e seu grupo [MOKA89].

Código de programas são representados por *T-graph*, similares a grafos de fluxo. A estrutura apresentada por um *T-graph* reflete fielmente a estrutura de um programa. Este grafo é gerado a partir de análises no código do programa-fonte e de limites do tempo de execução, que são computados após uma análise do grafo gerado no estágio anterior.

O tempo de execução de um programa é determinado basicamente por 2 fatores: o comportamento de um programa, que depende da estrutura do programa e do contexto da aplicação, e as características do hardware [MEND93]. Um programa/ trecho de código, nesta técnica, tem um ponto de início e um ponto de término.

As arestas representam trechos de código, e têm peso atribuído a elas (por exemplo, tempo de execução deste trecho de código), o que é representado pelo comprimento desta aresta. Os nós do grafo representam pontos no código do programa onde o fluxo de controle do programa divide (*split*) ou junta (*join*). A figura 4 apresenta exemplos de grafos de tempo, que ilustram algumas construções de linguagens de alto nível.



*Figura 4. Notação de T-graph.*

A execução de um programa paralelo é determinada por dois fatores: o comportamento do programa, que depende da estrutura do programa e do contexto da aplicação, assim como das características do sistema de computação envolvido [ADVE93]. A lista a seguir apresenta algumas considerações adotadas neste trabalho.

- Um programa / trecho de código de um programa tem um ponto de início e um ponto de término. O ponto de término sempre difere do ponto de início. Todas as execuções iniciam ou "entram" no código num ponto de

início e terminam ou deixam o código do programa no ponto de término. Deste modo, um trecho de código com um ponto de início e um ponto de término pode ser construído com um número arbitrário de pontos de início e de pontos de término.

- Toda execução pode ser descrita como um conjunto de execuções, que podem ser simultâneas, de vários trechos de código (por exemplo, instruções, sentenças, blocos, entre outros), nas quais os tempos de execução são conhecidos.
- O trecho de código é examinado e são conhecidas as seqüências de execução deste código, por exemplo, onde acontece um loop, um desvio. Ou seja, é conhecida a estrutura estática deste código.
- Para cada trecho de código, é conhecido o número máximo de repetições.

### 2.3.1 Representação de Programas Utilizando *T-graph*

Nesta seção será discutida a representação que será utilizada para a descrição dos problemas a serem resolvidos. Esta representação deve poder descrever propriedades estáticas e dinâmicas do código de uma aplicação em estudo. A estrutura estática de um programa/trechos de código é representada por grafos orientados. Os grafos são anotados com restrições, que caracterizam se um determinado caminho de execução pode ser praticável ou não.

#### 2.3.1.1 Representação da Estrutura Estática de Um Programa

Um programa ou um trecho de código é representado por um *T-graph*, um grafo de tempo. As arestas do *T-graph* representam instruções inclusas nos trechos de código e são atribuídos pesos a estas arestas, que representam tempos de execução. Quanto aos nós de um grafo, estes representam pontos do código, onde no fluxo de controle do programa pode ocorrer uma situação de divisão (*split*) ou de junção (*join*).

Dependendo da linguagem de descrição de código utilizado ou da linguagem de programação, as arestas de um *T-graph* podem representar instruções de máquina, seqüência de instruções/comando do código de uma linguagem de alto nível, etc. A figura 5 mostra como algumas construções típicas de linguagem de alto nível podem ser descritas em notações para *T-graph*.

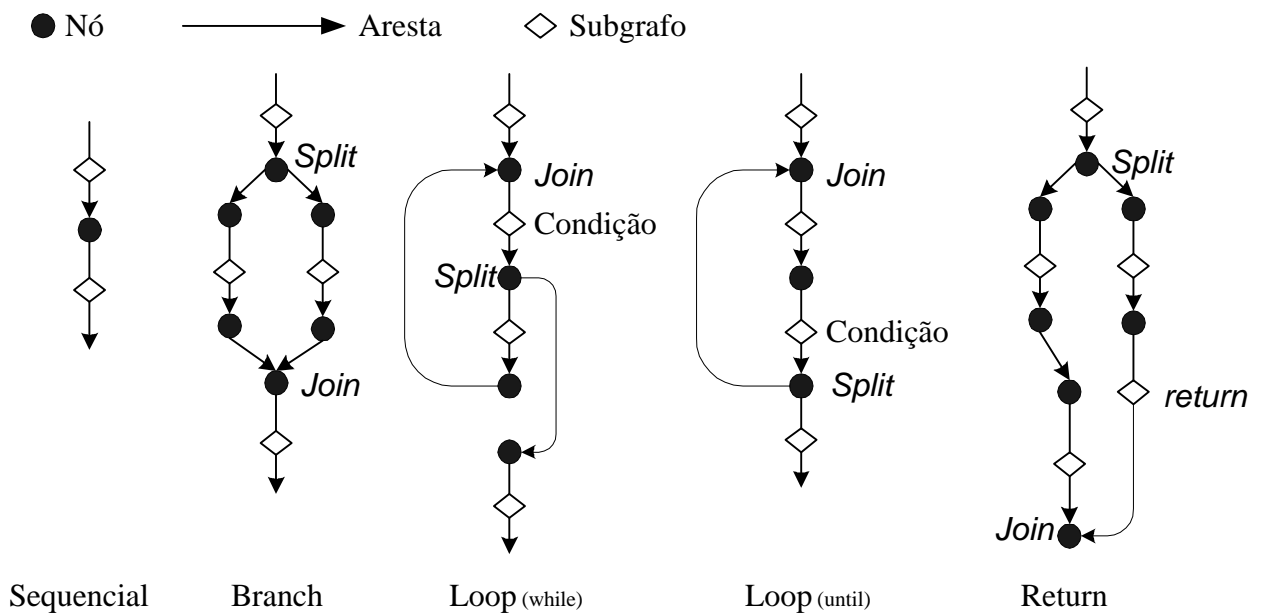


Figura 5. Representação em *T-graph* de uma construção típica de linguagem de programação.

Formalmente, um *T-graph* é um grafo orientado e conectado  $G=(V,E)$  com vértices (nós)  $V = \{v_i \mid 1 \leq i \leq |V|\}$  e arestas (arcos)  $E = \{e_i \mid 1 \leq i \leq |E|\}$ , onde cada aresta  $e_i$  pode também ser escrita como um par ordenado  $(v_j, v_k)$ . Um *T-graph* tem as seguintes propriedades:

1.  $G$  tem exatamente um nó  $s$ , no qual não há nenhuma outra aresta chegando a este nó (nó de início).
2.  $G$  tem exatamente um nó  $t$ , com nenhuma aresta saindo a partir dela (nó de término).
3. Para cada aresta  $e_i$  existe, no mínimo, uma seqüência de arestas com um ponto de início  $s$  e um ponto de final  $t$  que contenha  $e_i$  (não existe código não acessível).



4. Toda aresta tem um peso, que é representado por um valor inteiro (não-negativo)  $t_i = \tau(e_i)$  (tempo de execução).

Os itens 1 a 4 definem a estrutura de um *T-graph*. A interpretação da semântica pode ser dada por: os itens 1 e 2 auxiliam na caracterização dos caminhos de execução, nos quais serão computados os tempos de execução. Num *T-graph*, todos os caminhos iniciam no vértice  $s$  e terminam no vértice  $t$ . O item 3 garantirá que cada uma das partes do programa será parte de pelo menos um caminho do programa, com início no vértice  $s$  e término no vértice  $t$ . Finalmente, é atribuído, no item 4, o tempo de execução a cada aresta (ou arco) [KARY98, PUSC89, PUSC97].

### 2.3.2. Caminhos de Execução e Tempo de Execução

Na seção anterior, foi introduzida a representação de trechos de código utilizando *T-graph*. Nesta seção, serão definidos os termos **caminho de execução** (*execution path*) e **tempo de execução** (*execution time*).

A execução de um trecho de código é caracterizada pela execução de uma seqüência de ações descritas por aquele código. A execução é iniciada na primeira sentença (instrução, etc.), segue conforme o fluxo de controle como definido pelas construções de linguagem utilizadas, e termina na última sentença. Para cada execução deste tipo, existe no *T-graph*, uma seqüência correspondente de arestas, de  $s$  a  $t$ .

**Definição.** Uma seqüência de arestas  $P_1 = (e_{i1}, e_{i2}, \dots, e_{im})$  em um *T-graph*  $G$  com  $e_{i1} = (s, v_j)$  e  $e_{im} = (v_k, t)$  é denominado **caminho de execução**, ou simplesmente, **caminho**.

Todo caminho de execução consiste de um número finito de arestas. Cada aresta tem um peso atribuído pelo seu tempo de execução. Deste modo, pode ser atribuído a cada caminho um tempo de execução – a soma dos tempos de execução das suas arestas.

**Definição.** O tempo de execução  $\tau$  de um caminho  $P_i = (e_{i1}, e_{i2}, \dots, e_{im})$  é a soma dos tempos de execução das suas arestas, isto é,

$$\tau(P_i) = \sum \tau(e_{ij}), j = 1, 2, \dots, m.$$

Os tempos de execução das arestas de um caminho  $P_i$  podem ser somados em qualquer ordem. Cada um dos caminhos  $i$  pode, portanto, contabilizar o número de ocorrências de cada aresta  $e_j \in E$  em  $P_i$ , resultando em  $f_i(e_j)$ , ou  $f_{i,j}$ . O tempo de execução de  $P_i$  poderá então ser escrita da seguinte forma:

$$\tau(P_i) = \sum f_i(e_j) \tau(e_j) = \sum f_{i,j} t_j, \quad j = 1, \dots, |E|,$$

Onde  $|E|$  é o número total de arestas.

Para cada aplicação, o número de caminhos de execução diferentes é finito. A partir do momento em que é possível calcular o tempo de execução em cada um destes caminhos, é então possível calcular o máximo destes tempos de execução, o qual será o tempo máximo de execução (MAXT - *Maximum Execution Time*).

**Definição.** O tempo máximo de execução (MAXT - *Maximum Execution Time*) de um conjunto de caminhos  $\pi$  num  $T$ -graph  $G$  é

$$\text{Maxt}(\pi) = \text{Max}(\tau(P_i)), P_i \in \pi = \text{Max} \sum f_{i,j} t_j, P_i \in \pi, j = 1 \dots |E|$$

### 2.3.3. Caracterização de Caminhos de Execução

Ao enumerar todos os caminhos de execução de um trecho de código, o processo de enumeração não utiliza as informações contidas nas representações do  $T$ -graph, apesar do fato de que um  $T$ -graph gerado possa conter informações sobre prováveis ordens das arestas dos possíveis caminhos de execução deste código.

Nesta seção é apresentada uma técnica que utiliza a informação contida no  $T$ -graph como base para descrever os possíveis caminhos para o cálculo do MAXT. A

notação de *T-graph* é estendida com a introdução de restrição de capacidade (*capacity constraints*) que restringe o fluxo nas suas arestas. Assim, a computação do MAXT de respectivos trechos de código é dada por custo máximo de circulação nos *T-graph* estendidos.

### 2.3.3.1 Circulação

Será introduzido o conceito de circulação, como apresentado na teoria dos grafos. Dado um grafo orientado e conectado  $G$ , uma função  $f: E \rightarrow \mathfrak{R}$  é denominada **circulação** se esta conserva o fluxo em cada um dos nós:

$$\forall v \in V \quad \sum_{e=(v_j, v)} f(e) = \sum_{e=(v, v_k)} f(e)$$

As restrições de capacidade  $b: E \rightarrow \mathfrak{R}$  e  $c: E \rightarrow \mathfrak{R}$  restringem os valores de  $f$  para todas as arestas. A circulação é denominada **legal** se

$$\forall e \in E: b(e) \leq f(e) \leq c(e).$$

Finalmente, seja  $\gamma: E \rightarrow \mathfrak{R}$  uma função custo em  $G$ . O custo  $\gamma(f)$  de uma circulação  $f$  é definido como:

$$\gamma(f) := \sum_{e \in E} \gamma(e) f(e)$$

### 2.3.3.2 T-graph e Circulações

O objetivo é computar MAXT de um trecho de código como uma circulação inteira de custo máximo no *T-graph*  $G$ . Para isso, *T-graph* é mapeado sob uma descrição de uma circulação. Os seguintes passos são executados:

Um nó anterior  $e_{|E|+1} = (t, s)$  é adicionado ao *T-graph*  $G$ , produzindo um *T-graph* estendido  $G'$  com  $V' = V$  e  $E' = E \cup \{e_{|E|+1}\}$ . O *T-graph* estendido permite circulações que têm um fluxo saindo de  $s$  e entrando em  $t$ .

Para cada aresta  $e_i$  as restrições de capacidade (*capacity constraints*)  $b(e_i)$  e  $c(e_i)$  são definidos como:

$$b(e_i) := \left\{ \begin{array}{l} 1, \text{ se } e_i = (t,s) \text{ (nó anterior)} \\ 0, \text{ caso contrário} \end{array} \right\}$$

$$c(e_i) := \left\{ \begin{array}{l} 1, \text{ se } e_i = (t,s) \\ f_{\max}(e_i), \text{ caso contrário} \end{array} \right\}$$

O termo  $f_{\max}(e_i)$  representa o número máximo de execuções de uma aresta  $e_i$  no caminho  $\pi$ , isto é,  $f_{\max}(e_i) = \max_{P_j \in \pi} f_j(e_i)$ .

Os custos das arestas no *T-graph* estendido são:

$$\gamma(e_i) := \left\{ \begin{array}{l} 0, \text{ se } e_i = (t,s) \\ \tau(e_i), \text{ caso contrário} \end{array} \right\}$$

Estas regras de transformação dão a descrição formal de como um *T-graph* e um conjunto de caminhos que descrevem um trecho de código podem ser caracterizados por uma circulação e restrições de capacidade (*capacity constraint*). Vale observar as seguintes propriedades:

- Para cada caminho de execução  $P_i = ((s, v_{i1}), (v_{i1}, v_{i2}), \dots, (v_{im}, t))$  em  $G$ , a seqüência de arestas  $P' = ((s, v_{i1}), \dots, (v_{im}, t), (t, s))$  em  $G'$  induz uma circulação completa (*integer circulation*) em  $G'$ : é definido que toda ocorrência de uma aresta e numa seqüência fechada é adicionado 1 para o fluxo nesta aresta.
- Para toda aresta  $e_i$  que é membro de um caminho de execução, ocorre  $c(e_i) \geq 1$ . Somente se a aresta que não é parte de nenhum caminho (código “morto”),  $c(e_i) = 0$ .

## 2.4 Comentário

Foi visto, neste capítulo, a concepção de modelos de sistemas distribuídos, uma rápida apresentação de operações de comunicação ponto-a-ponto e coletiva do MPI, e, finalmente, alguns elementos e propriedades do *T-graph*.

A introdução do *T-graph* e a apresentação do caminho de execução são de grande importância, pois, a partir destes tópicos, foi possível estruturar técnicas para o cálculo de tempo máximo de execução de um programa, através da sua representação em grafos. Alguns outros trabalhos relacionados com o tema podem ser encontrados em [BALA91, CAIN00].

## Capítulo 3

# Análise e Predição de Desempenho

Análise de desempenho vem se tornando cada vez mais importante na medida em que as opções de uso de sistemas paralelos aumentam e pela alta sensibilidade do desempenho com relação a pequenas variações em um programa. Neste sentido, é fundamental a compreensão não apenas do desempenho obtido, mas das causas que levam a um certo resultado.

A compreensão mais detalhada do desempenho de um certo sistema é importante para uma melhor identificação destes problemas, para melhorar a paralelização e para a construção de modelos capazes de prever o desempenho em outras situações (maior número de processadores, tamanho do problema, outra arquitetura) [HOCK91, JAIN91, KANT92, KARY98, LIKU01, LIKU99A, REED87].

Algumas metas que se pretende atingir com o estudo de desempenho de um sistema paralelo são:

- Capacidade de previsão do desempenho em diversas situações. Isto é, mesmo que exista um programa executando de forma eficiente em um certo equipamento, deseja-se prever seu comportamento quando alguns parâmetros são alterados, efetuando-se desta forma perturbações.

- Verificar o uso eficiente do sistema, com indicações de gargalos e otimizações. Muitas vezes, deseja-se verificar a interação entre uma dada implementação e o sistema computacional, pois existe um conjunto de parâmetros que pode influenciar a execução do programa.
- Estudar a viabilidade de implementação de programas complexos, isto é, a construção de um programa paralelo eficiente implica em um investimento de tempo e equipamento, e a possibilidade de uma análise prévia capaz de estimar a qualidade do resultado final pode ajudar na decisão de alocação de recursos [CROV94, KARA99].

### 3.1 Introdução

Escrever programas paralelos eficientes é uma tarefa difícil, pois o tempo de execução de um programa paralelo é dependente de um conjunto de fatores, de forma complexa. Para permitir que programadores de programas paralelos descubram rapidamente a melhor estrutura para o programa paralelo a ser desenvolvido, é muito importante que este entenda o conjunto de fatores que podem afetar o desempenho. A análise a ser mostrada aqui tem a finalidade de mostrar alguns fatores associados à programação paralela quanto à eficiência.

Os fatores que afetam o desempenho podem ser classificados em dois grupos, que são externos e internos. Fatores externos são aspectos do ambiente de tempo de execução do programa não especificados no código fonte do programa. Nesta pesquisa são considerados 5 fatores externos, que são: (1) número de processadores utilizados para a execução do programa, (2) tamanho do conjunto de dados de entrada, (3) estrutura do conjunto de dados de entrada, (4) definição do problema e (5) a máquina utilizada para a execução do programa. A definição do problema refere-se ao conjunto de programas que podem resolver diversas variações de um problema.

Os fatores internos são métodos e técnicas utilizados para a paralelização da aplicação. Estes métodos e técnicas correspondem a mudanças na estrutura do programa. Alguns exemplos de fatores internos são: (1) tipo de paralelismo utilizado, que pode ser paralelismo no nível de tarefas, ou paralelismo no nível de dados, (2) a

escolha de qual trecho de código paralelizar, por exemplo, quais os *loops* de um programa paralelizar, e (3) a escolha de métodos de sincronização, por exemplo, *locks* ou barreiras, não-bloqueante ou bloqueante. Diversos outros fatores internos não foram listados aqui.

Cada um dos fatores externos listados acima pode afetar na escolha da melhor forma de paralelizar uma aplicação, isto é, a escolha de determinados valores para os fatores externos pode determinar uma melhor escolha dos fatores internos.

## 3.2 Técnicas de Avaliação

O estudo e a avaliação de um sistema computacional são baseados na descrição da arquitetura e na definição dos programas aplicativos que são executados neste sistema. Em sistemas multiprocessadores, a avaliação tem sido realizada com modelagens, construção de protótipos e técnicas de simulação [HARR93, LIKU99B, LIYS95, REED93]. Segue nesta seção algumas abordagens mais utilizadas.

### 3.2.1 Modelagem Analítica

Modelos analíticos fazem uso de um conjunto de equações e fórmulas para descrever o funcionamento da máquina e da aplicação de um modo abstrato. Um modelo deste tipo é em geral de baixo custo do ponto de vista computacional e pode possuir uma grande modularidade, permitindo seu transporte para outras arquiteturas e situações.

Em contrapartida, a modelagem de sistemas muito complexos pode requerer a utilização e o tratamento de um número muito grande de parâmetros e funções, tornando o modelo intratável. Deste modo, grandes modificações devem ser feitas prejudicando assim a precisão do resultado obtido, como acontece com problemas de comportamentos dinâmicos, como alocação de processadores durante a execução, algoritmos adaptativos, etc.

Portanto, a construção de um modelo analítico deve levar em conta dois pontos: o nível de detalhe a ser modelado (e conseqüentemente sua precisão) e a tratabilidade do modelo.



### 3.2.2 Modelagem Estatística

As modelagens pertencentes a esta abordagem ocorrem quando as funções dos modelos analíticos são substituídas por distribuições que fornecem o comportamento esperado dos diversos parâmetros. Desta forma, o comportamento exato não precisa ser conhecido, perde-se precisão, no entanto.

A principal vantagem de se utilizar esta modelagem é a flexibilidade ganha. Por outro lado, estes modelos fornecem um comportamento assintótico e muitas das ferramentas são custosas. Por exemplo, Redes de *Petri*, embora sejam úteis para a verificação de estabilidade e outras propriedades, são inviáveis para a análise do tempo de execução de um sistema complexo [STRO97].

### 3.2.3 Modelos Empíricos

Estes modelos baseiam-se em medições de trechos de programas que representam as operações mais comuns como, por exemplo, *broadcasting*, sincronização de barreiras, chamadas remotas de procedimento, criação de canais de comunicação virtual, etc. Desta forma, anota-se diretamente os parâmetros necessários, evitando a construção de modelos que descrevam os níveis mais básicos da aplicação [BALA91].

O maior problema deste método é a dependência no modo de construção da aplicação, isto é, o tipo da máquina, as bibliotecas e o compilador utilizado. Uma variação da classe de modelos empíricos é o modelo dinâmico, que usa informações retiradas de uma execução do programa para completar o modelo. Neste caso, o modelo se presta mais à análise de desempenho do que à predição.

### 3.2.4 Simulação

Num simulador, tem-se um controle maior dos diversos parâmetros de execução, e assim, é mais fácil obter informações para construir e validar modelos de desempenho.

São possíveis as construções de um simulador específico, a emulação de uma arquitetura em outra, a utilização de traços de execução [MEND93], ou ainda, a

utilização de informações fornecidas pelo sistema de compilação (combinada com modelos analíticos).

O alto custo e a necessidade de um conhecimento detalhado do equipamento a ser simulado são os maiores problemas com esta abordagem [NAHU96, NICO96].

### 3.2.5 Modelos Híbridos

Para tentar minimizar as desvantagens de cada um dos métodos, pode-se construir um modelo híbrido com a combinação de diversas abordagens.

É apresentado, em [FAHR93, GUBI95A, GUBI96, GUNT98], uma ferramenta de predição e análise de desempenho que faz uso de três abordagens diferentes: parâmetros fundamentais são obtidos a partir de uma execução simplificada do programa, e modelos analíticos são acoplados a uma simulação de modo a obter o modelo final.

Uma segunda estratégia, mais geral, é utilizada onde o programa é instrumentado e uma série de execuções simples são feitas. O resultado das diversas medições é comparado com modelos analíticos que representam os comportamentos mais freqüentes, através de regressões.

## 3.3 Trabalhos Relacionados

O trabalho da pesquisa proposta pode ser descrito mais adequadamente como uma ferramenta que utiliza TG (Grafo de Tempo no nível de Programa) e modelagem híbrida, com o propósito de fornecer predição de desempenho para analistas e programadores em computação paralela.

É possível caracterizar trabalhos relacionados em 3 áreas, que são:

- **Ferramentas de medição de desempenho e técnicas para programas paralelos.** Ferramentas de medição de desempenho usualmente oferecem a descrição de execução de um programa paralelo num formato que tenta explicitar razões do mau desempenho e fornece ainda sugestões para

melhorar a aplicação. Grande parte destas ferramentas não fornece predição de desempenho.

- **Técnicas de análise de desempenho para programas paralelos.** Análise de programas paralelos é um estudo de sensibilidade de desempenho da aplicação como uma função do número de processadores e do tamanho do problema. Tais análises são geralmente assintóticas e, portanto, não são adequados para predizer o desempenho de aplicações específicas em máquinas específicas, sendo que normalmente é necessário muito esforço do programador.
- **Técnicas de predição de desempenho para programas paralelos.** Predição de desempenho tenta oferecer predições de tempo de execução para aplicações específicas em máquinas específicas, geralmente com o suporte de ferramentas. As técnicas podem variar dependendo do tipo de dado de entrada utilizado. Técnicas estáticas utilizam código-fonte (pseudocódigo) como principal entrada. Técnicas dinâmicas dependem principalmente de medidas de desempenho, utilizando estas para predizer outras medidas de desempenho.

O grau de interesse no assunto discutido neste trabalho é crescente e o número de trabalhos e de grupos de pesquisa é cada vez maior, tanto no nível teórico quanto no estudo de implementações específicas e de métodos semi-automáticos de geração de modelos.

São descritos nas seções seguintes, trabalhos recentes cujos temas são relacionados com o trabalho aqui apresentado.

### 3.3.1 Trabalho Desenvolvido por P. Puschner e A. Schedl

O conhecimento do tempo de execução de programa é de grande importância para o desenvolvimento e a verificação de software. Desta forma, há uma necessidade de métodos e ferramentas para a predição do comportamento de tempo, tanto de trechos de códigos de programa quanto de um programa inteiro [HITC82, KAPL95, KAPL97, PUSC89, PUSC97, SHAW89].

Assim, P. Puschner e A. Schedl apresentam uma técnica inovadora para análise de tempos de execução de programas paralelos. A computação de MAXTs (Tempos Máximos de Computação) é mapeada em um problema de grafos, que é uma generalização da computação de custo máximo de circulação num grafo orientado. Programas são representados por *T-graph*, grafos de tempo, que são similares a grafos de fluxo. Estes grafos refletem a estrutura e o comportamento temporal do código fonte. Para computar MAXTs, *T-graph* são buscados para determinados caminhos de execução que correspondem a um custo máximo de circulação. Assim, o problema da busca é transformado em um problema de programação linear. Finalmente, a solução deste problema de programação linear trará o resultado MAXT, tempo máximo de computação deste programa paralelo.

São três diferenciais que este trabalho apresenta, que são: esta ferramenta utiliza uma notação concisa para caracterizar a estrutura estática do programa e prováveis caminhos para a execução. Outro diferencial é que a notação apresentada permite uma descrição de possíveis caminhos através do código do programa que caracteriza o comportamento do código, o suficiente para computar o tempo máximo de execução exato do programa. Computar utilizando programação linear não gera somente tempos máximo de execução, mas também produz informações detalhadas sobre o tempo de execução e o número de execuções de cada construção do programa no seu pior caso. Conhecer estes dados são importantes para uma análise mais detalhada dos tempos de um programa paralelo [PARK91, PARK92].

### 3.3.2 Trabalho desenvolvido por M. Gubitoso

Na medida em que as opções de uso de sistemas paralelos aumentam e pela alta sensibilidade do desempenho com relação a pequenas variações em um programa, a área de análise e de predição vem se tornando cada vez mais importante. Deste modo, é fundamental a compreensão não apenas do desempenho obtido, mas das causas que levam a um certo resultado. A abordagem analítica, por ser baseada em uma descrição detalhada do funcionamento do programa, permite um melhor entendimento do desempenho.

O trabalho apresentado pelo autor procura explorar este tipo de abordagem na análise teórica do comportamento de sistemas paralelos e na predição de desempenho de um sistema de memória compartilhada virtual. Os casos teóricos estudados são o da alocação de processadores para laços do tipo DOALL independente e do tempo total de execução em aplicações com decomposição de domínio com comunicação assíncrona. Para o sistema de memória compartilhada virtual é desenvolvido um método para a construção de modelos analíticos de desempenho que se mostrou bastante preciso, dentro das aplicações escolhidas e analisadas [GUBI95A, GUBI95B, GUBI96].

### 3.3.3 Trabalho Desenvolvido por D. F. Vrsalovic, D. P. Siewiorek, Z. Z. Segall e E. F. Gehringer

Foi desenvolvido um modelo para predição do desempenho de sistemas multiprocessadores utilizando algoritmos iterativos, elaborados pelos pesquisadores das universidades Carnegie-Mellon e North Carolina State.

Cada iteração deste algoritmo consiste de uma certa parcela de acessos a dados globais e outra parcela de processamento local. De forma global, estas iterações podem ser síncronas ou assíncronas, enquanto os processadores podem estar expostos ou não a tempo de espera, dependendo da distribuição atribuída entre o tempo de acesso e o tempo de processamento a cada um dos processadores do sistema computacional.

O efeito da velocidade do processador, memória e a rede de interconexão no desempenho podem ser estudados. A modelagem ilustra inclusive o impacto significativo no desempenho da decomposição de um algoritmo em processos paralelos. As predições feitas pelo modelo são calibradas com medidas experimentais.

O modelo desenvolvido por D.F.Vrsalovic [VRSA88] mostra diferenças no desempenho entre algoritmos síncronos e assíncronos. É contabilizada a diferença entre os efeitos de diferentes processadores e da velocidade da memória global. Inclusive, podem ser considerados os diferentes modos nos quais os algoritmos são

decompostos para o processamento paralelo com a divisão da computação em  $n$  processos.

Basicamente, o modelo desenvolvido assume que um sistema multiprocessador é composto de  $N$  processadores, onde cada um destes tem sua memória privada para código e dados locais. Ainda, cada processador tem acesso, via uma rede de interconexão, recurso comum do sistema, à memória global. O tempo de acesso à memória global é independente da localização deste. Acessos a recursos comuns são garantidos pela estrutura FIFO, ao invés de utilizar um esquema de prioridades. Deste modo, o tempo de espera poderá ser modelado como uma função linear de  $N$ . O modelo assume que o balanceamento de carga paralelo é infinitamente possível de ser decomposto. Cada processo executa um conjunto de iterações idênticas. Cada iteração é completada em um ciclo de processamento, e o mesmo ciclo é utilizado para efetuar o acesso à memória global (o acesso à memória local em cada um dos nós de processamento é “descontado” do tempo de processamento, ao invés de modelá-lo separadamente).

Seja:

$t_p \equiv$  tempo de processamento utilizado para executar uma iteração;

$t_a \equiv$  tempo de acesso global utilizado para executar uma iteração;

$t_w \equiv$  tempo de espera devido à contenção a recursos globais.

O modelo utiliza o conceito de poder de processamento real (efetivo), que é definido como o número de processadores no sistema multiplicado pelo uso médio de cada processador. O significado do poder de processamento real (efetivo)  $E$  pode ser dada como a taxa de utilização média do sistema computacional multiprocessador e este pode ser expresso por:

$$E \equiv \sum_N \frac{(t_a + t_p)}{(t_a + t_p + t_w)} \quad (1)$$

Foi-se assumido neste trabalho que a carga é balanceada, isto é,  $t_a$  e  $t_p$  são iguais em todos os processadores. No caso da carga de trabalho ser síncrona, todos os processadores devem terminar a iteração corrente antes que qualquer processador possa iniciar uma nova iteração, e ainda, todos os processadores tentam acessar a memória ao mesmo tempo. O tempo para completar a tarefa para uma iteração será determinado pelo processador na qual o acesso completou por último. Este processo espera por  $t_w = (N-1)t_a$  unidades de tempo. Dado que todos os outros processadores devem aguardar que o último processador termine, todos os outros processadores também têm  $t_w = (N-1)t_a$ . Se a carga está balanceada, a equação (1) é degradada na sua multiplicação por  $N$ , da qual obtém-se

$$E \equiv N * \frac{t_a + t_p}{(t_a + t_p + t_w)} = \frac{N}{1 + \frac{(N-1)t_a}{t_a + t_p}} \quad (2)$$

E, portanto,

$$E \equiv \frac{N}{1 + \frac{(N-1)}{\rho}}, \quad (3) \quad \text{onde } \rho = \frac{t_a + t_p}{t_a}$$

Se ocorrer o caso em que não há balanceamento de carga, ou seja, em cada um dos nós de processamento há um processo de tamanho diferente, e os processadores estão executando as mesmas instruções, estes em breve tornam-se “defasados”, e assim, seus tempos de acessos globais serão diferentes. Portanto, não haverá contenção se  $N-1$  processadores têm tempo para completar seus acessos enquanto o  $N$ -ésimo está processando, isto é, se  $t_p \geq (N-1)t_a$ , ou de forma equivalente, quando  $\rho \geq N$ .

Portanto, quando  $N \leq \rho$ , o tempo de iteração é dominado pelo tempo de processamento, e  $E = N$ . No entanto, quando  $N > \rho$ , ocorre contenção, e o tempo de espera para cada tarefa é  $(N-1)t_a - t_p$ . Neste caso, o tempo de iteração é dominado pelo tempo de acesso, e

$$E = \sum_N \frac{(t_a + t_p)}{t_a + t_p + (N - 1)(t_a - t_p)} \quad (4)$$

Portanto, no caso em que os nós de processamento não executam processos homogêneos,

$$E = \mathbf{min} [N, \rho] \quad (5)$$

Com o uso do modelo similar ao apresentado acima, e com posterior aperfeiçoamento na inserção de parâmetros arquiteturais como velocidade de processador e de memória, foi verificado que existe limite máximo de desempenho, apesar de valores crescentes de velocidades de processador e de memória. Ainda, com a aplicação do modelo final, é possível conhecer limites do pior caso de desempenho que se deve esperar na execução de um algoritmo com balanceamento de carga.

### 3.3.4 Trabalho Desenvolvido por A.J.C. van Gemund

É apresentada, neste trabalho desenvolvido por van Gemund, uma técnica que automaticamente gera um modelo de desempenho simbólico, durante a tradução do programa (compilação), que efetua predição de tempo de execução de um programa paralelo, uma vez definido o modelo do sistema multiprocessador alvo. A motivação desta proposta é que uma expressão simbólica (algébrica) retém informações com diagnósticos da complexa interação entre vários programas e parâmetros do sistema envolvidos (processadores, distribuição de dados, parâmetros de comunicação, etc.).

É possível efetuar pesquisas e estudos, variando os parâmetros do sistema envolvido, utilizando ferramentas / aplicações matemáticas baseadas em modelo compilado um-somente (*one-only*), como MAPLE, MATLAB ou MATHEMATICA, diferente dos tradicionais ciclos modificação-compilação-avaliação. O compilador desenvolvido tem uma interface que “experimenta” diversas técnicas de otimização dentro de um domínio definido, diferentemente dos tradicionais, que “forçam” uma



integração com um compilador específico (assume-se que a otimização durante a compilação seja considerada por último).

A proposta é baseada na utilização de análise de série (*serialization analysis*), uma extensão de baixo custo no tradicional esquema estático. O método de análise é feito em termos de um formalismo de modelagem de desempenho, denominado PAMELA (PerformAnce ModELing LAnguage). Ainda, o método de compilação simbólica combina vantagens de baixo custo da técnica de predição estática, com a fundamental confiabilidade apresentada pela análise de probabilidades.

Maiores detalhes do formalismo da linguagem PAMELA e exemplos do uso da técnica de compilação num algoritmo de fatoração LU paralela são encontrados em [GAUT00, GEMU93, GEMU95, GEMU96].

### 3.3.5 Trabalho Desenvolvido por J. Landrum, J. Hardwick e Q.F.Stout

Este trabalho discute o problema de predição do tempo, de quanto tempo levaria um determinado algoritmo para resolver um problema de um determinado tamanho sobre um sistema computacional específico.

Os modelos de predição propostos são baseados na técnica estatística de amostragem adaptativa. Para um sistema computacional específico e dados as condições do problema, identificar e efetuar modelagem de relevantes pontos de variação é uma tarefa crítica e difícil, onde a meta é gerar predições precisas. Diversas complicações surgem devidas à complexa natureza pouco compreendida das variações que ocorrem no sistema computacional, pois para pequenas quantidades de dados de entrada a predição de desempenho gerada tem qualidade comprometida, enquanto para grande quantidade de dados de entrada poderão ocorrer alterações no comportamento da memória cache, e ainda, adiciona-se o fato de que tempo é uma restrição e também a variável que está sendo estimada.

Varição pode ser introduzida por dados, pelo algoritmo, pelo hardware e o sistema de software sobre os quais está sendo executado o algoritmo, e ainda, pelos outros usuários presentes na mesma rede local. A proposta padrão de utilizar análise de notação da ordem (*order notation analysis*) para modelar crescimento gera

somente taxas assintóticas onde, sem constantes específicas, esta proposta é vaga e incompleta para ser aplicada e utilizada na prática.

A proposta do trabalho feita por J. Landrum e outros é: dado um sistema computacional específico, e o tamanho do problema, o objetivo é prever o tempo de execução para um tamanho de problema alvo e estimar a precisão da previsão. Dentre as metas deste projeto está planejada a inclusão de vários fatores de refinamento para o modelo e especificações para a amostragem.

Maiores detalhes sobre a modelagem e a técnica estatística aplicada podem ser encontrados em [LAND98].

### 3.3.6 RSIM (the Rice Simulator for ILP Multiprocessors)

RSIM é um simulador dirigido à execução, desenhado primeiramente para pesquisa e estudo de sistemas monoprocessores e multiprocessores de memória compartilhada e construída a partir de processadores comercialmente disponíveis [DURB98, PAIV97A, PAIV97B]. Ao ser comparado com outros simuladores para sistemas de memória compartilhada publicamente disponíveis, a principal vantagem do RSIM é que este tem implementado um modelo de processador que explora adequadamente paralelismo no nível de instruções (ILP), o que é mais representativo em termos de processadores atuais, e inclusive, os do futuro próximo.

Um principal diferencial deste simulador quando comparado com outros trabalhos é que este implementa o modelo de processador de forma mais completa e apresenta características como: múltiplas instruções, escalonamento dinâmico *out-of-order* (com opção para *in-order*), renomeação de registros, previsão de ramificações estática e dinâmica, *load* e *store* não bloqueantes, vários tipos de implementações de modelos de consistência de memória. Deste modo, poderá evitar uma falta de precisão quando utilizado na pesquisa e estudo do comportamento de sistemas de memória compartilhada. Quanto à modelagem da micro-arquitetura desenvolvida no RSIM, este tem as principais características dos processadores atuais. Em particular, é muito próximo a MIPS R10000.

O sistema RSIM modela as contenções em recursos do processador, memória *cache*, bancos de memória, barramento processador-memória e inclusive, a rede de interconexão. O sistema de memória definido no RSIM apresenta uma hierarquia da

memória cache de dois níveis, sendo L1 *pipelined* e *multiported*, enquanto L2 *pipelined*. O sistema multiprocessador definido é um sistema CC-NUMA de memória compartilhada com protocolo de coerência de cache baseada em diretórios, e tem apoios para protocolos de coerência MSI ou MESI, consistência seqüencial, consistência de processador e consistência por liberação.

A técnica de simulação utilizada pelo RSIM basicamente efetua interpretação dos executáveis dos aplicativos. Foi escolhida esta técnica, ao invés da execução direta, pois modelar processadores ILP utilizando execução direta é, ainda, um problema em aberto.

O sistema de simulação RSIM é desenvolvido de forma modular e implementado utilizando as linguagens C e C++, visando a portabilidade e possibilidades de extensão. As plataformas utilizadas para executar este simulador podem ser: SUN executando Solaris 2.5 ou versão acima, SGI Power Challenge executando IRIX 6.2 e Convex Exemplar executando HP-UX versão 10. A figura 6 mostra a modelagem do processador no simulador RSIM, enquanto a figura 7 mostra a arquitetura do sistema paralelo modelado e utilizado no simulador RSIM.

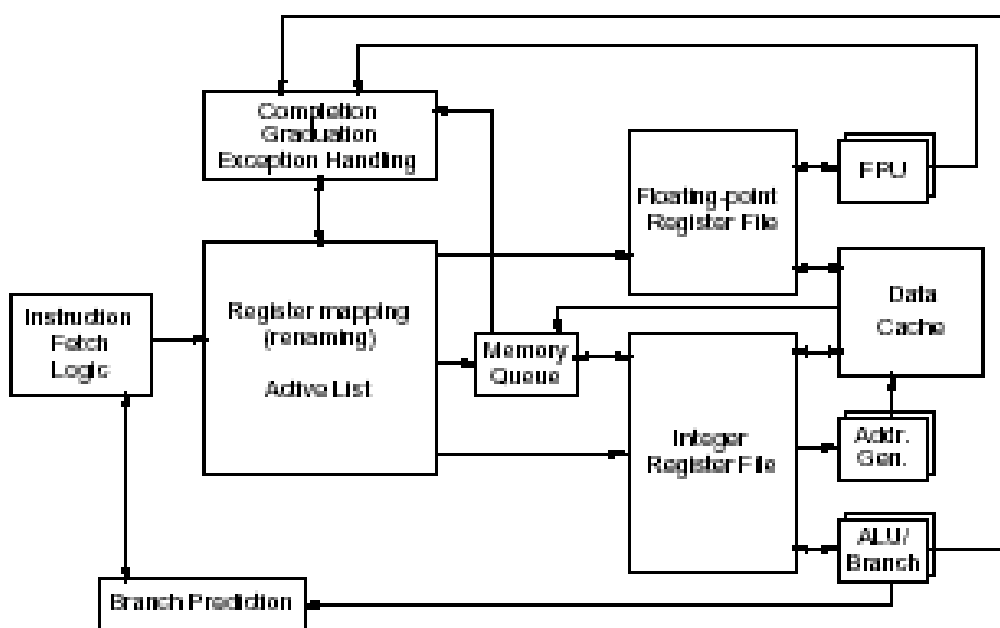
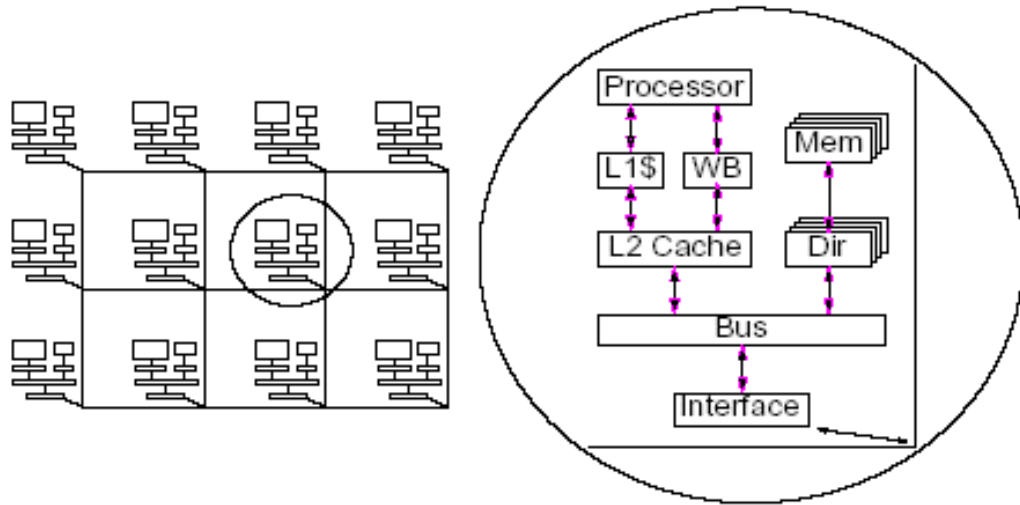


Figura 6. Modelagem de um processador no RSIM.



*Figura 7. Arquitetura do sistema paralelo.*

## Capítulo 4

# Metodologia para Análise e Predição de Desempenho

Escrever programas paralelos é uma tarefa difícil, especialmente quando se pretende descrever este algoritmo de forma eficiente. Ainda, a execução de um programa paralelo é dependente de diversos fatores, que interagem entre si de forma complexa. Para identificar e compreender qual seria a melhor estrutura para um programa paralelo, é importante entender quais são os fatores que poderiam interferir no desempenho deste programa.

Os fatores que interferem no desempenho podem ser identificados e separados em 2 classes, denominados de externos e internos. Os fatores externos são aspectos do ambiente do sistema computacional, não especificados no código do programa paralelo, que podem ser enumerados como: o número de processadores utilizados para executar o programa, o tamanho dos problemas, a estrutura do conjunto de dados de entrada (o conteúdo dos dados de entrada, por exemplo, matriz esparsa), o modo como o problema foi resolvido e o sistema computacional envolvido na execução do programa.

Quanto aos fatores internos, estes se referem aos métodos utilizados para descrever e paralelizar a aplicação. Podem ser citados como exemplos destes fatores:

(1) o tipo de paralelismo utilizado, tais como, paralelismo no nível de tarefas e paralelismo no nível de dados, (2) quantos e quais trechos de código / *loop* paralelizar e (3) métodos de sincronização / comunicação utilizados.

A proposta de efetuar predição de desempenho, utilizando a concepção de dois níveis, tem sido amplamente divulgada e utilizada, dentre as técnicas de predição existentes [ADVE93], pois utilizando-se desta técnica é possível isolar os fatores de desempenho provenientes do nível de aplicação e do nível de sistema. Entre todos os métodos de predição existentes, estes diferem entre si no modo como a modelagem de naturezas não-determinísticas do sistema reflete nos eventos de comunicação entre os processos, contenção dos recursos compartilhados e estrutura de programas [ZHAN95].

Neste trabalho, é proposta uma metodologia de análise e predição de desempenho de aplicações paralelas processadas sobre uma rede de estações de trabalho. São apresentadas estratégias para a obtenção de informações, especificamente tempos de execução de uma dada aplicação, solicitadas pela metodologia. Ainda, é construída uma extensão do conjunto de grafos de tempo, denominado *T-graph\**, com o propósito de completar a tabela de simbologias para a representação de programas paralelos com MPI em alto nível.

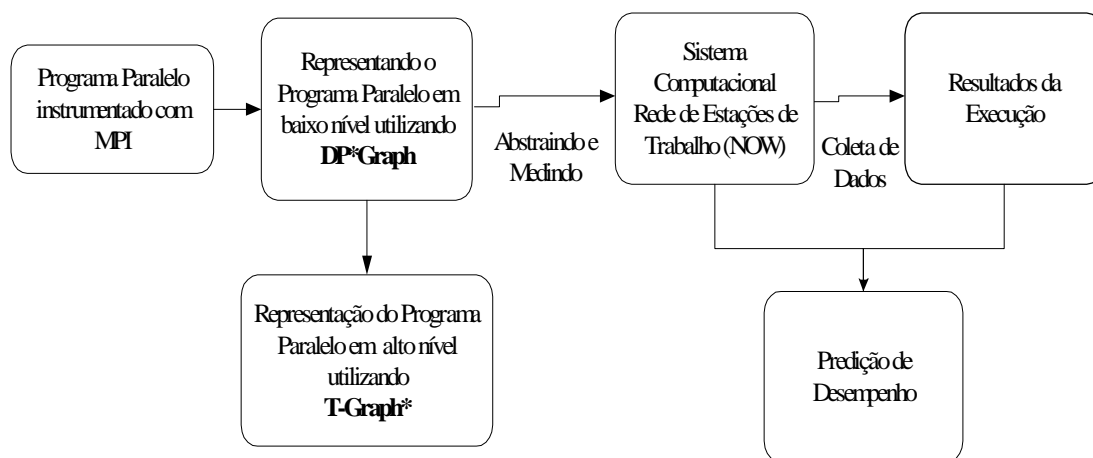
Utilizando a abordagem proposta de grafos de tempo estendidos e, em conjunto com técnicas e dados experimentais, ter-se-á, assim, instrumentos e informações necessários para efetuar estudos de análise e predição de desempenho.

A metodologia proposta neste trabalho foi concebida a partir de observações sobre trabalhos apresentados no capítulo anterior, nas quais propostas de modelagens foram introduzidas, utilizando as mais diversas técnicas matemáticas e modelos, onde os objetivos são sempre a avaliação de desempenho. Portanto, mais do que efetuar uma avaliação de desempenho, como já apresentam outros trabalhos, o objetivo é possibilitar a realização de uma predição de desempenho para um programa paralelo sobre um sistema de estações de trabalho, com características diferentes (número de nós, tamanho do problema) daquelas com as quais foram obtidos os dados.

A figura 8 mostra a estrutura completa da metodologia proposta de análise e predição de desempenho. O grafo de tempo *T-graph\** é uma forma de abstração da

aplicação num modelo de alto nível, enquanto o grafo de baixo nível *DP\*Graph* é responsável pela representação do mesmo programa paralelo de forma detalhada, correspondendo a cada passo da seqüência de execução do código deste programa. Uma vez traçado o grafo de representação de baixo nível, o programa é submetido a várias execuções, com diferentes números de processadores e tamanhos de problema. Após terminar o conjunto de testes, é possível efetuar modelagens de trechos de código com estes dados. Com os resultados obtidos da modelagem deste programa paralelo, é possível efetuar previsões de desempenho sobre ele.

Para isolar os efeitos não-determinísticos da contenção na rede, os pontos onde ocorrem comunicações dentro de um trecho de código em execução são identificados, destacados e computados isoladamente em relação às computações locais de uma tarefa. As computações locais de cada tarefa distribuída são abstraídas como um conjunto de segmentos de código, onde cada segmento abrange todas as computações locais entre dois pontos sucessivos da comunicação de uma tarefa.



*Figura 8. Metodologia de Análise e Predição Proposta Neste Trabalho.*

## 4.1 Definição de Classes de Grafos

Foi introduzida na seção 2.3 a classe de grafos *T-graph*, apresentando suas estruturas de representação, utilização e integração entre si, com o propósito de representar

trechos de código. Nesta seção, o propósito é formalizar a extensão da classe de grafos de tempo *T-graph*, definindo as estruturas já introduzidas e mostradas, inclusive a sua aplicabilidade no próximo capítulo. Foi constatada a necessidade desta extensão, para que programas paralelos implementados com interface de passagem de mensagens MPI possam ser representados de forma completa e precisa.

#### 4.1.1 Definição da Classe de Grafos *DP\*Graph*

No processamento distribuído, a comunicação entre nós de processamento é fundamental, provendo a troca de informações e dados entre eles. Na interface de passagem por mensagens MPI, estas podem ser efetuadas utilizando os comandos *send*, *receive*, *broadcast*, *reduce*, *scatter*, entre outros comandos de comunicação.

A figura 9 mostra uma proposta de elementos de grafos definidos para permitir a representação de programas paralelos em baixo nível.

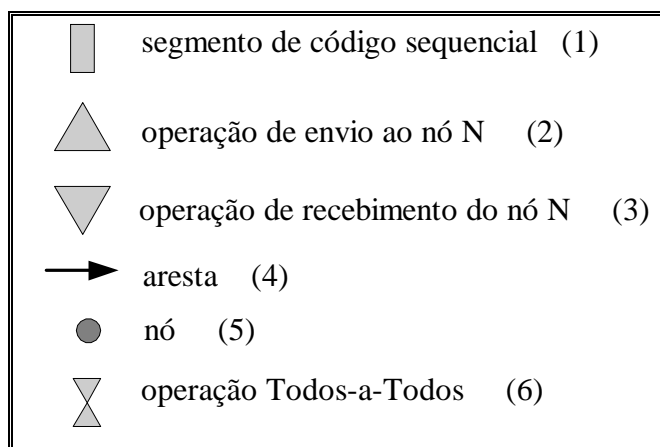


Figura 9. Elementos de Representação da classe *DP\*Graph*.

A estrutura (1) representa um trecho de código de uma tarefa, que não apresenta comando de comunicação com outro processo, ou pode ser um trecho de código entre duas comunicações feitas na tarefa.

As estruturas (2) e (3) são sempre operações “casadas”, isto é, se existir uma operação de comunicação dentro de um processo, que envia dados, existirá um outro



comando de comunicação num outro processo, que recebe estes dados enviados. Pode-se citar exemplos destas situações, como o comando *send* e suas variações (diferentes tipos de *send*, bloqueantes ou não-bloqueantes) com a operação de comunicação *receive*, *broadcast* (um nó de processamento envia, enquanto todos os outros processos recebem), entre outros diferentes tipos de operação de comunicação. A estrutura (6) representa operações de comunicação que envolvem todos os nós de processamento do sistema computacional.

Finalmente, as estruturas (4) e (5) são estruturas fundamentais para expressar o fluxo da execução do programa paralelo.

#### 4.1.2 Definição da Classe de Grafos *T-graph*\*

A classe de grafos *T-graph*\* consiste basicamente do *T-graph*, apresentado anteriormente na seção 2.3, adicionado de uma estrutura de representação para um segmento de código sequencial. Esta extensão é necessária pois a análise e predição de desempenho requisita um maior grau de detalhamento.

Deste modo, as estruturas que compõem o conjunto de representações desta classe são:

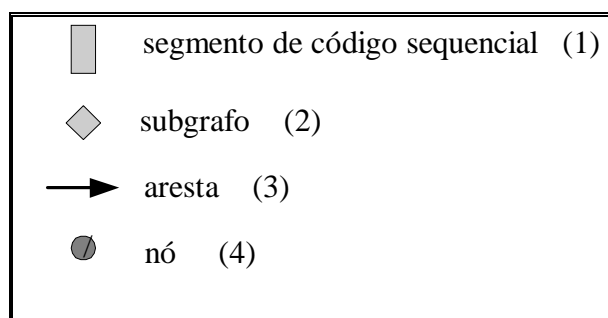





Figura 10. Elementos de Representação da Classe *T-graph*\*

## 4.2 Representação de Baixo Nível

A representação de baixo nível é construída com o uso do conjunto de estruturas oferecidas pela classe de grafos  $DP*Graph$ , apresentada na seção 4.1.1. A representação reflete com fidelidade a seqüência de execução dos códigos nos processos de cada um dos nós de processamento.

Os trechos de código seqüenciais são representados utilizando , enquanto as comunicações, independentemente de serem ponto-a-ponto, bloqueantes ou não, ou coletivas, são representadas por  (quando este nó de processamento envia mensagens) e  quando o nó de processamento recebe mensagens de um outro nó.

Segue na figura 11(a) uma listagem de um programa paralelo, instrumentado com funções da interface MPI. E, em seguida, a figura 11(b) mostra a representação deste programa listado, com a utilização da classe de grafos  $DP*Graph$ , traduzindo os trechos de código às suas representações correspondentes. A representação de baixo nível mostra a execução de trecho de código seqüencial e uma operação de comunicação ponto-a-ponto, entre dois processos, que corresponde ao de baixo nível.

```

if (rank == 0){
  for (x=0;x<BUF1024;x++){
    buf1[x] = x + 99;
  }
  MPI_send(buf1,1,0);
}

if (rank == 1){
  for (x=0;x<BUF1024;x++){
    buf2[x] = x + 3;
  }
  MPI_Recv(buf1,0,1);
}

```

Figura 11(a). Listagem de um programa paralelo-Caso1.

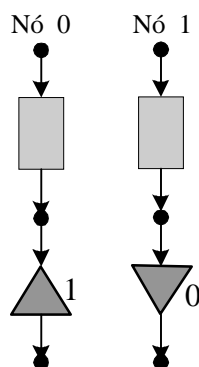


Figura 11(b). Representação do programa - Caso1 - apresentado na figura 11(a).

A figura 12(a) apresenta a listagem de um programa paralelo em que cada processo efetua um *loop* como parte da computação local, e logo depois, efetua uma operação de envio ou de recebimento ponto-a-ponto, dependendo do processo. Vale observar que a sintaxe apresentada na listagem está simplificada, para evitar dúvidas.

```

if (rank == 0){
  for (x=0;x<BUF888;x++){
    buf1[x] = x + 99;
  }
  MPI_send(buf1,1,0);
}

if (rank == 1){
  for (x=0;x<BUF888;x++){
    buf2[x] = x + 7;
  }
  MPI_Recv(buf1,0,1);
}

if (rank == 2){
  for (x=0;x<BUF999;x++){
    buf3[x] = x + 89;
  }
  MPI_Recv(buf4,3,2);
}

if (rank == 3){
  for (x=0;x<BUF999;x++){
    buf4[x] = x + 99;
  }
  MPI_Send(buf4,2,3);
}

```

Figura 12(a). Listagem de um programa paralelo-Caso2.

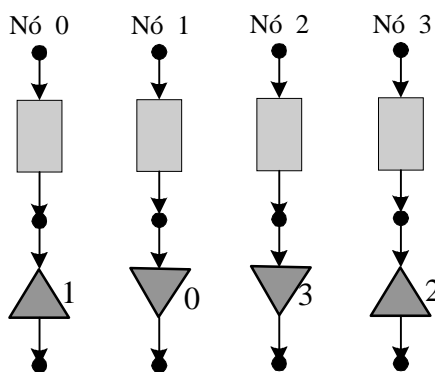


Figura 12(b). Representação do programa - Caso2 - utilizando DP\*Graph.

```

for (x=0;x<BUF1024;x++){
    teste[x] = 2*x;
}

if (rank == 0){
    for (x=0;x<BUF1024;x++){
        buf1[x] = x + 99;
    }
    MPI_Bcast(buf1,0,todos);
}
if (rank == 1){
    for (x=0;x<BUF1024;x++){
        buf2[x] = x + 3;
    }
}

MPI_Bcast(buf1, 0,1);
}
if (rank == 2){
    for (x=0;x<BUF1024;x++){
        buf3[x] = x + 15;
    }
    MPI_Bcast(buf1,0,2);
}
if (rank == 3){
    for (x=0;x<BUF1024;x++){
        buf4[x] = x + 19;
    }
    MPI_Bcast(buf1, 0,3);
}

```

Figura 13(a). Listagem de um programa paralelo-Caso3.

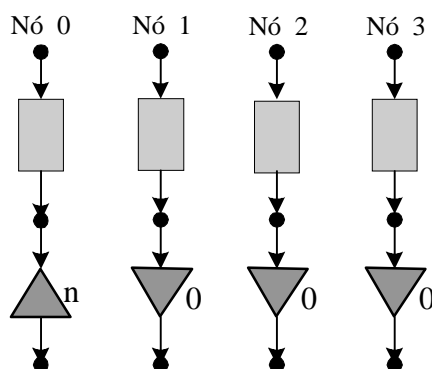


Figura 13(b). Representação do programa - Caso3 - utilizando DP\*Graph.

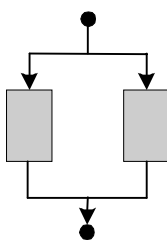
A listagem 13(a) ilustra o caso em que cada um dos processos executa um trecho de código seqüencial, e após o término da execução deste *loop*, é executada uma operação de comunicação coletiva. A figura 13(b) mostra a representação de baixo nível deste programa paralelo.

### 4.3 Representação de Alto Nível

A representação de alto nível de um programa paralelo pode ser especificada utilizando o conjunto de estruturas da classe de grafos *T\_graph*, apresentada em 4.1.2. Esta representação mostra o fluxo de execução de programas paralelos.

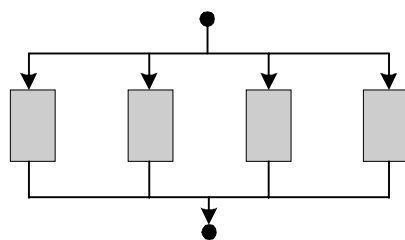
Após obter a representação no baixo nível do programa paralelo, a construção da sua representação no alto nível é quase imediata, pois a representação de baixo nível torna as ocorrências e a seqüência de execução do programa paralelo claras e explícitas.

A figura 14 mostra representações no alto nível de trechos de código de programa paralelo que contém chamadas de funções da interface MPI, apresentado na figura 11(a).



*Figura 14. Representação em Alto Nível da listagem apresentada na figura 11(a).*

A figura 15 mostra uma representação no alto nível dos programas paralelos apresentados nas figuras 12(a) e 13(a). Note-se que a representação é a mesma para os programas apresentados nas listagens 12(a) e 13(a). Isso ocorre, pois, nas representações de alto nível, o enfoque principal é algorítmico.



*Figura 15. Representação em Alto Nível das listagens apresentadas nas figuras 12(a) e 13(a).*

## 4.4 Esquema da Metodologia Proposta

A metodologia proposta contém os passos seguintes:

1. Efetua-se uma primeira análise no programa paralelo, descrevendo-o na representação no baixo nível, utilizando o conjunto de estruturas da classe de grafos *DP\*Graph*,
2. Instrumenta-se o programa paralelo apresentado, incluindo medições de tempo entre o início e o fim de cada trecho de código, delimitado por comunicações, como também para cada comunicação de dados. O programa instrumentado é executado no ambiente de rede de estações de trabalho, variando o número de nós de processamento e o tamanho do problema, obtendo assim, tempos de execução de cada um dos trechos do programa paralelo e de cada uma das comunicações de dados, nas diversas situações.
3. Modelagens são feitas a partir dos dados obtidos, em cada um dos trechos de código, considerando o comportamento de cada trecho de código entre comunicações como também as características obtidas através do uso de dados obtidos no passo 2. Neste passo são formuladas equações que calculam o tempo gasto para cada trecho, delimitado entre comunicações, incluindo a própria comunicação. Os coeficientes das equações são obtidos através da solução de um sistema de equações lineares construído a partir dos tempos obtidos experimentalmente no passo 2 e aplicados na equação do cálculo do tempo. Estes coeficientes refletem as características do sistema computacional e da rede de interconexão. O tempo de execução do trecho de código ou do programa, para o qual se deseja efetuar a predição, é a somatória dos tempos de cada trecho de código entre comunicações, incluindo-se as comunicações. Se a aplicação não incluir trechos de código que levem a equações cuja somatória que calcula o tempo total não é demasiadamente complexa, pode-se utilizar a equação resultante desta somatória para o cálculo do tempo total. Neste caso, basta instrumentar o programa, no passo 2, para medir o tempo de execução total do trecho de código ou programa, e aplicar os tempos obtidos na equação que calcula o tempo total, e obter os coeficientes desta equação.
4. É possível realizar predições de desempenho de trechos de código ou do programa paralelo inteiro, para um dos seguintes casos:

- Fixando o número de nós de processamento e variando o tamanho do problema.
- Fixando o tamanho do problema, e variando o número de nós de processamento.

Para se efetuar a predição, toma-se a equação que calcula o tempo gasto em cada trecho de código, delimitado por comunicações, e substitui-se os valores de **n**, para o primeiro caso ou **p**, para o segundo caso, e finalmente, faz-se a somatória dos tempos referentes a cada trecho de código. Ou, conforme mencionado no passo 3, uma alternativa, dependendo da aplicação, é tomar a equação que efetua o cálculo do tempo de execução total do trecho de código ou programa.

5. A partir da representação no baixo nível apresentada é possível construir um segundo grafo, cuja representação utiliza a classe de grafos *T-graph\**. Esta representação do programa paralelo em alto nível tem o objetivo de mostrar o fluxo de execução do programa paralelo, sem explicitar detalhes como as comunicações entre os nós.

## 4.5 Modelagem da Aplicação para o Cálculo dos Tempos de Execução

Em uma aplicação paralela, para se obter o tempo de execução de um trecho de código é necessário o tempo de execução de todos os processos que executam simultaneamente.

Para exemplificar, é mostrada, na figura 16, a representação de baixo nível de um programa paralelo. Observe que há cinco processos em execução concorrentemente, onde cada processo executa diferentes trechos de código. Um processo X pode estar executando a soma de **n** números, assim como um processo Y pode estar somando 2 matrizes quadradas de ordem **n**. Baseado nisso, é fácil notar que os tempos de execução de processos X e Y são diferentes, dado que a complexidade do algoritmo de cada processo é diferente. O tempo de execução  $T_{\max}$

deste programa paralelo, que contém 5 processos em execução simultânea, é o tempo do processo que tem o maior valor, no caso,  $T_1$ .

$$T_{\max} = \max \{T_1, T_2, T_3, T_4, T_5\} = T_1$$

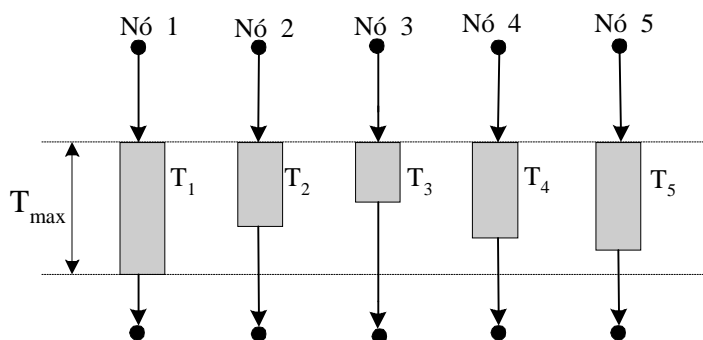


Figura 16. Tempo máximo de execução de um programa paralelo com 5 processos.

#### 4.5.1 Chaveador (*Switch*)

Chaveador, ou *Switch*, é a parte fundamental na interconexão física de sistemas computacionais, sejam PCs ou estações de trabalho. Estes chaveadores tornam possível que diversos usuários enviem ou recebam informações, simultaneamente através da rede, sem que isso prejudique a taxa de transferência dos dados, como ocorre com *hubs*. Similar a roteadores que permitem que diferentes redes de interconexão comuniquem entre si, chaveadores permitem que diferentes nós (tipicamente um PC ou estação de trabalho) de uma rede se comuniquem, de forma eficiente.

A principal diferença entre o *hub* e o *switch* é que todos os nós conectados a um *hub* compartilham a mesma largura de banda entre estes nós, enquanto um nó conectado a um *switch* tem a banda integral, como se estivesse isolado. Por exemplo, se 10 nós estão conectados entre si através de um *hub* numa rede de 10Mbps, então a taxa máxima de cada nó é uma parcela dos 10Mbps, quando outros nós conectados ao *hub* quiserem enviar mensagem também. Mas, se os nós estivessem conectados a



um *switch*, cada nó pode possivelmente atingir a taxa máxima de 10Mbps [CISC00B, CISC00C].

Com o avanço da tecnologia, as principais vantagens do *switch* apresentadas sobre *hub* são:

- ❑ Escalabilidade: numa rede local onde a interconexão é feita com o uso de um *hub*, a largura de banda é limitada e compartilhada entre os nós conectados, o que torna ainda mais difícil o crescimento da rede, sem sacrificar o desempenho.
- ❑ Latência: latência é basicamente o tempo necessário para que um pacote de informação chegue até o nó destino, a partir de um nó que o enviou. Numa rede baseada em conexões via *hub*, os nós aguardam uma oportunidade de transmitir seus pacotes na rede, para evitar colisões. Ainda, a latência aumenta à medida que um maior número de nós de processamento são conectados a esta rede.
- ❑ Colisões: *Ethernet* utiliza um processo denominado CSMA/CD para comunicar-se na rede de interconexão. Sob CSMA/CD, um nó não enviará um pacote de mensagem, a menos que a rede esteja livre para tráfego. Se dois nós enviam mensagens ao mesmo tempo, ocorre uma colisão, e os pacotes de mensagens são perdidos. Ambos os nós aguardam por um tempo aleatório e retransmitem os pacotes.

#### 4.5.1.1 Arquitetura do Chaveador (*Switch*)

É mostrada na figura 17 a arquitetura de um chaveador moderno, pertencente a projetos de otimização da rede de interconexão Ethernet a baixo custo, atendendo assim às tecnologias *Ethernet* e *Fast Ethernet*. Um exemplo de chaveador baseada nesta arquitetura é a linha Catalyst, fabricada pela Cisco Systems [CISC00A, CISC97A].

O chaveador utilizado no ambiente de testes deste trabalho tem características baseadas nesta arquitetura.

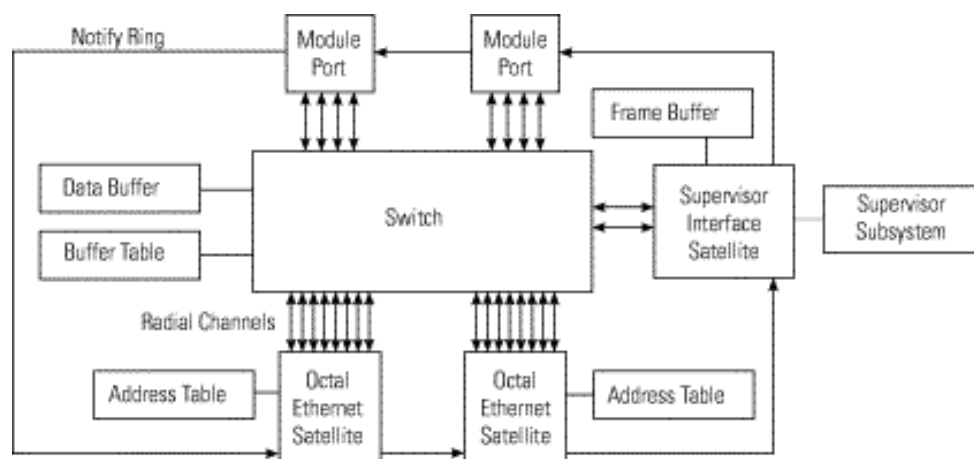


Figura 17. Arquitetura Interna de um Switch.

#### 4.5.2 Um exemplo de Cálculo do Tempo de Execução

A figura 18 mostra representações nos dois níveis, apresentados nas seções anteriores, de um programa paralelo dado. O objetivo desta seção é mostrar como o tempo global de execução de um programa paralelo é calculado.

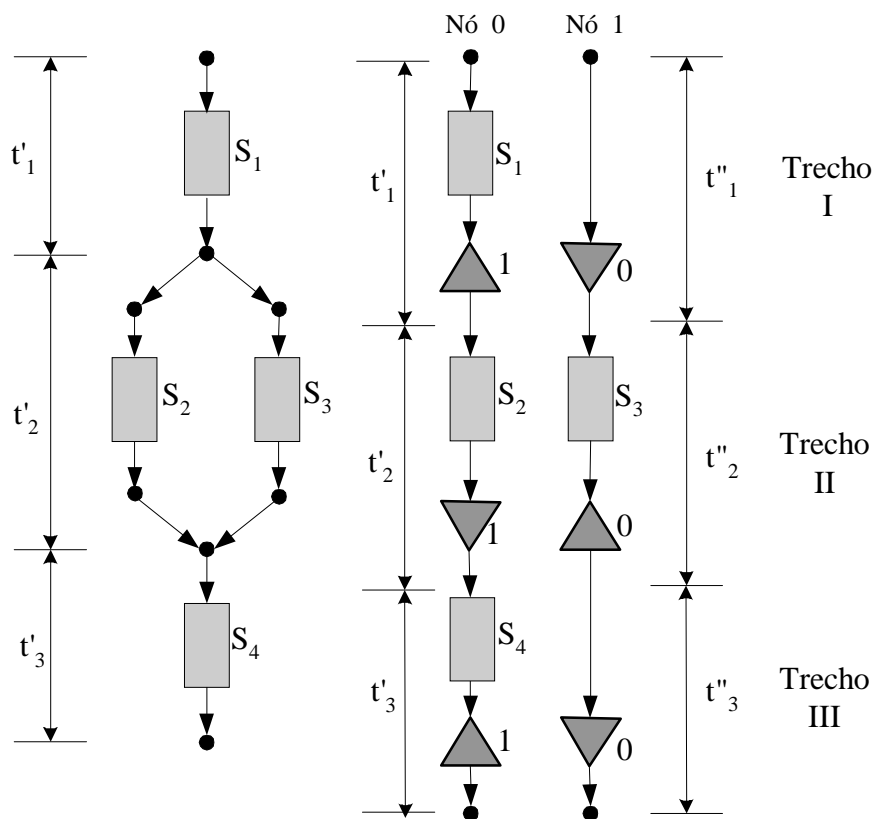


Figura 18. Cálculo do Tempo de Execução.

O tempo de execução do programa paralelo é o máximo entre os tempos de execução total em cada nó, calculado pela soma dos tempos parciais  $t'_1$ . Assim, o tempo de execução total é dado por:

$$t_{\text{execução total}} = \text{máximo}(t'_1 + t'_2 + t'_3, t''_1 + t''_2 + t''_3)$$

Define-se a seguinte simbologia. Seja:

- $t^{(n)}_{S1}$  = tempo de execução do trecho de código **S1**, calculando **n** dados.
- $t^{(n)}_e$  = tempo de envio dos **n** dados à rede de interconexão.
- $t^{(n)}_t$  = tempo de transferência dos **n** dados na rede de interconexão, de um nó ao outro.
- $t^{(n)}_r$  = tempo de recepção dos **n** dados da rede de interconexão.

O tempo de execução do trecho I no nó de processamento 0 é dado por:

$$t'_1 = t^{(n)}_{S1} + t^{(n)}_e$$

Para obter o tempo de execução do trecho  $t'_2$ , é preciso analisar os tempos de execução  $t^{(n)}_{S2}$  e  $t^{(n)}_{S3}$ .

Se  $t^{(n)}_{S2} + t^{(n)}_r + t^{(n)}_t + t^{(n)}_e < t^{(n)}_{S3}$  então  $t'_2 = t^{(n)}_{S3} + t^{(n)}_r + t^{(n)}_t + t^{(n)}_e$ , pois a tarefa S2 já terminou e o nó de processamento 0 aguarda o término da execução da tarefa S3, e então, é adicionado ao tempo  $t'_2$  os tempos de envio, transferência e recepção dos dados a serem enviados pelo nó de processamento 1.

Se  $t^{(n)}_{S2} > t^{(n)}_{S3}$  então o tempo de execução  $t'_2$  é dado por:  $t'_2 = t^{(n)}_{S2} + t^{(n)}_r$ , pois quando o nó de processamento 0 termina a tarefa S2 e solicita dados para o nó de processamento 1, estes dados já se encontram no seu *buffer* de recepção, sendo necessário adicionar ao tempo  $t'_2$  apenas o tempo de recepção  $t_r$ .

O tempo  $t'_3$  é dado pela execução da tarefa S4 pelo nó de processamento 0. Assim, o tempo de execução  $t'_3$  é dado por  $t'_3 = t^{(n)}_{S4}$ .

Portanto, o tempo de execução total do programa representado pela figura 18 é dado por:

$t_{\text{execução total}} = t'_1 + t'_2 + t'_3$ , dependendo do tempo de execução das tarefas  $t^{(n)}_{s2}$  e  $t^{(n)}_{s3}$ .

O tempo de execução total do programa representado pela figura 17 é dado por:

$$t_{\text{execução total}} = \max(t'_1 + t'_2 + t'_3, t''_1 + t''_2 + t''_3).$$

A situação exemplificada aqui é bastante comum na programação paralela.

Serão feitas, nas subseções que seguem, análises de comandos de comunicação da interface de passagem de mensagens MPI, estruturas fundamentais num programa paralelo para sistemas distribuídos.

Comunicação entre nós de processamento é fundamental em sistemas de estações de trabalho, e o tempo gasto na comunicação é um fator que deve ser levado em consideração, quando se pretende efetuar estudos e análises de programas paralelos. Desta forma, são feitos alguns estudos destas operações de comunicação.

As nomenclaturas apresentadas abaixo serão necessárias para os estudos de análises das próximas subseções. Como já relacionado anteriormente, seja:

- $t^{(n)}_e$  = tempo de envio dos **n** dados à rede de interconexão.
- $t^{(n)}_t$  = tempo de transferência dos **n** dados na rede de interconexão, entre um nó ao outro.
- $t^{(n)}_r$  = tempo de recepção dos **n** dados da rede de interconexão.

Visualizando a representação na figura 19, e identificando com a numeração acima fornecida, tem-se:

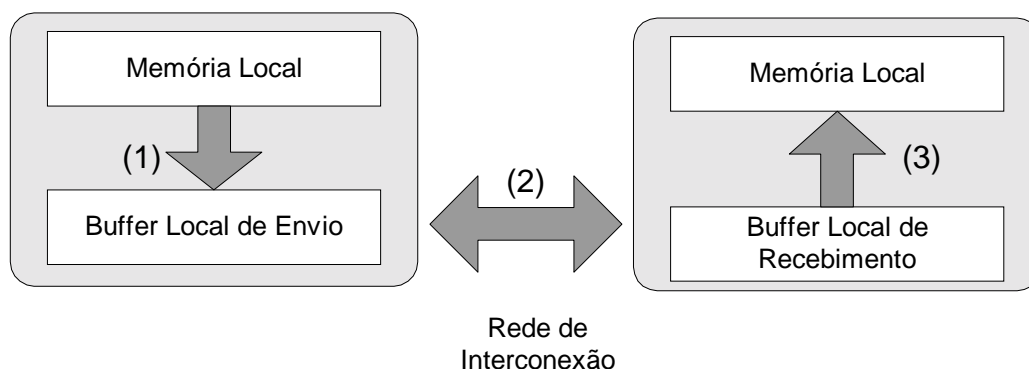
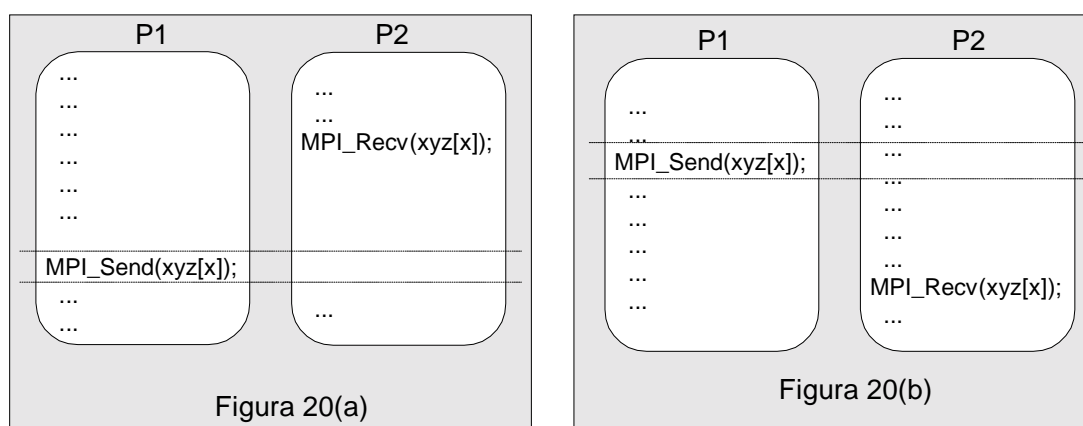


Figura 19. Explicação da nomenclatura fornecida.

### 4.5.3 Análise da Operação de Comunicação *Send/Receive*

Nesta subseção, serão feitos estudos das operações de comunicação `MPI_Send`/`MPI_Recv`, funções de transferência de dados muito utilizadas em implementações de programas paralelos.

Basicamente, o estudo é estruturado e dirigido em 2 casos:



Figuras 20(a) e 20(b). Casos de estudo da operação `MPI_Send`.

A figura 20(a) apresenta o caso em que o processo P2 espera pela chegada dos dados do processo P1, enquanto a figura 20(b) mostra que a operação de comunicação *send* do processo P1 já enviou os dados e no instante em que P2 os

solicitar para dar a continuidade na execução do trecho de código, estes já se encontram no *buffer* de recepção de P2.

No caso apresentado pela figura 20(a), e observando pelo esquema apresentado na figura 21,  $T_1 > T'_1$ , ou seja, tem-se que a operação `MPI_Recv()` no processo  $P_2$  está aguardando os dados a serem enviados pelo processo  $P_1$ . Assim, o tempo de execução é dado por:

$$T_{\text{execução}} = T'_1 + T_c + T'_2 = T'_1 + T_e^{(n)} + T_t^{(n)} + T_r^{(n)} + T'_2.$$

No caso apresentado pela figura 20(b),  $T_1 < T'_1$ , ou seja, tem-se que ao atingir a operação `MPI_Recv()` no processo  $P_2$ , o dado já está disponível no *buffer de recepção de P2*, dando continuidade na execução do trecho de código. Assim, o tempo de execução máximo para este programa paralelo é dado por:

$$T_{\text{execução}} = T_1 + T_c + T_2 = T_1 + T_e^{(n)} + T_2.$$

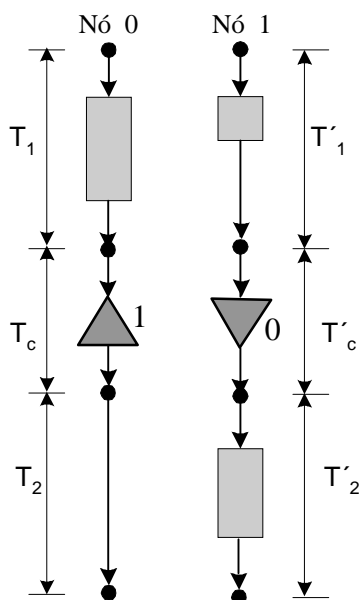


Figura 21. Representação para o estudo de casos apresentados pelas figuras 20(a) e 20(b).

O estudo de outros modos de operação de envio, como *MPI\_Bsend*, *MPI\_Ssend* e *MPI\_Rsend*, apresentados no capítulo 2, pode ser realizado de forma similar. O mesmo ocorre com as operações de comunicação não-bloqueantes, como *MPI\_Isend*, *MPI\_Ibsend*, *MPI\_Issend* e *MPI\_Irsend*.

#### 4.5.3.1 Análises em Função de Tamanho de Mensagem e Número de Nós de Processamento

No caso da figura 20(a) estudado, e considerando o tamanho de mensagem  $n$ , em um ambiente com conexão através de um chaveador (*switch*), conforme apresentado anteriormente, e seja:

$k_1$  constante que caracteriza a latência da rede de comunicação,

$k_2$  constante que caracteriza largura de banda da rede de comunicação,

$k_3$  e  $k_4$ , constantes.

$$t_t^{(n)} = k_1 + k_2 n.$$

$t_e^{(n)} = k_3 \times n$ , pois  $t_e$  inclui apenas uma transferência de envio, da memória local para *buffer* de envio.

$t_r^{(n)} = k_4 \times n$ , pois  $t_r$  inclui apenas uma transferência de recebimento, do *buffer* de recepção para a memória local.

Desta forma, da fórmula do tempo de execução acima obtida para o caso (a), tem-se no nó de processamento P2:

$$\begin{aligned} T'_{\text{execução}} &= t'_1 + t_e^{(n)} + t_t^{(n)} + t_r^{(n)} + t'_2. = \\ &= t'_1 + t'_2 + k_1 + k_2 n + k_3 n + k_4 n = \\ &= t'_1 + t'_2 + (k_2 + k_3 + k_4) n + k_1 \end{aligned}$$

A equação acima pode ser reescrita da seguinte forma:

$$T'_{\text{execução}} = t'_1 + t'_2 + kn.$$

No nó de processamento P1, o tempo é dado por:

$$T_{\text{execução}} = t_1 + t_2 + t_e^{(n)} = t_1 + t_2 + k_3 n.$$

Analisando agora o caso (b), o tempo no nó de processamento P1 é

$$T_{\text{execução}} = t_1 + t_2 + t_e^{(n)} = t_1 + t_2 + k_3 n ,$$

Enquanto no nó de processamento P2, o tempo é dado por:

$$T'_{\text{execução}} = t'_1 + t'_2 + t_r^{(n)} = t'_1 + t'_2 + k_4 n .$$

#### 4.5.4 Análise da Operação de Comunicação *Broadcast*

A função `MPI_Bcast()` é uma operação de comunicação coletiva em que um processo *root* envia uma mesma mensagem a todos os outros nós do grupo, pertencentes ao sistema computacional de estações de trabalho. A figura 21 mostra a movimentação dos dados, quando esta função de comunicação é executada, num sistema com quatro nós de processamento.



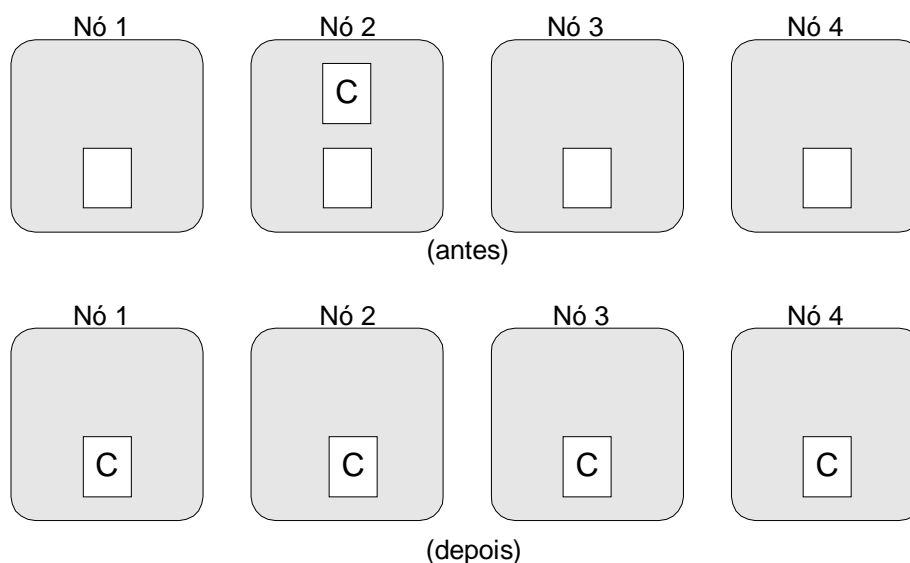


Figura 22. Esquema da movimentação dos dados da Operação `MPI_Bcast()`.

Nos códigos de programas paralelos que incluem esta operação da interface de passagem de mensagens MPI, podem ocorrer casos que serão analisados a seguir, sendo exemplificados pelas figuras 23(a) e 23(b).

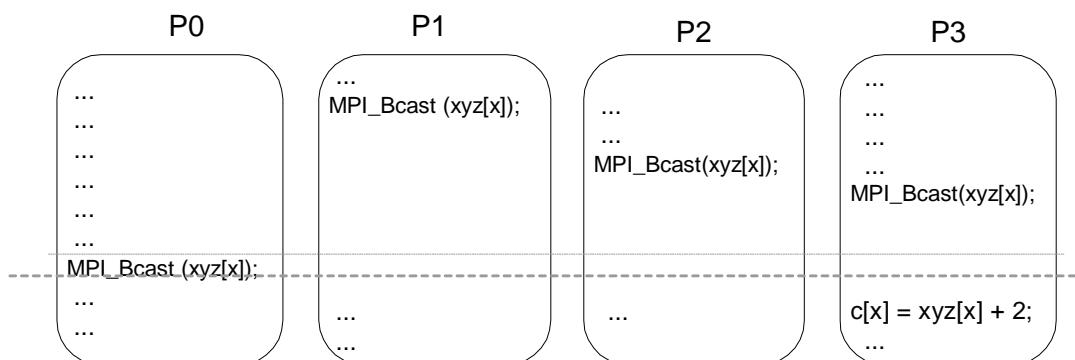


Figura 23(a). Estudo da operação de comunicação `MPI_Bcast()`.

A figura 23(a) ilustra o caso em que os nós P1, P2 e P3 já aguardam os dados relativos à operação de comunicação `MPI_Bcast()`, a ser executada pelo nó P0. Neste caso, a execução dos códigos nos nós P1, P2 e P3 fica na espera, aguardando o nó P0 executar a operação `MPI_Bcast()`. Após o envio dos dados para os nós P1, P2 e P3, a computação nestes nós continuará somente após estes dados serem transferidos para suas memórias locais.

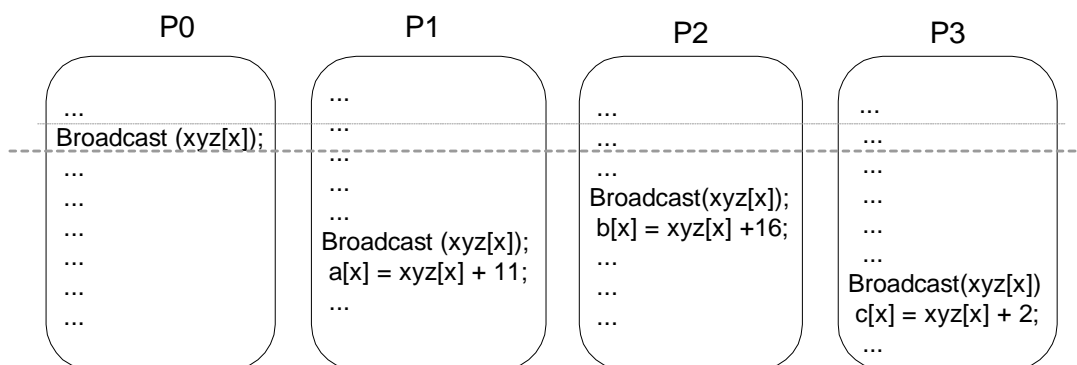


Figura 23(b). Estudo da operação de comunicação *MPI\_Bcast()*.

Na figura 23(b), é ilustrado o caso em que os nós P1, P2 e P3 já têm os dados enviados pelo nó P0 na operação *MPI\_Bcast()*, e que ao ser atingida a linha de código correspondente a esta operação, em cada um dos nós, não haverá espera para a continuidade do processamento, sendo feita imediatamente uma simples transferência dos dados, do *buffer* local de recepção para a memória local.

Na figura 24, encontra-se a representação em *DP\*Graph* dos trechos de código apresentados das figuras 23(a) e 23(b).

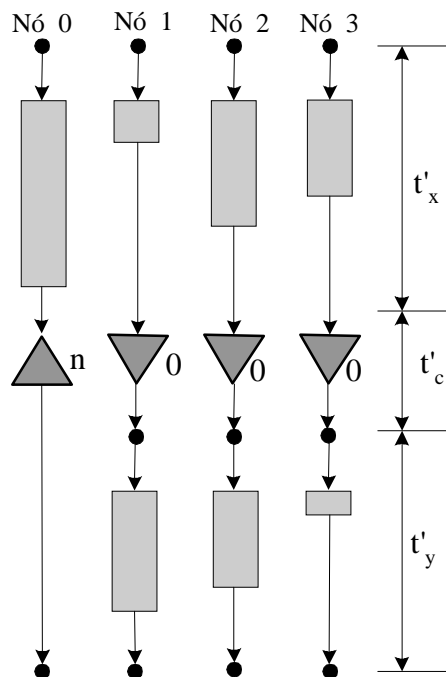


Figura 24. Representação em *DP\*Graph* do estudo da operação *MPI\_Bcast()*.

Sejam os nós de processamento P1, P2 e P3 receptores, enquanto o nó de processamento P0 é o nó envidor de dados. No primeiro caso, mostrado pela figura 23(a), os nós de processamento P1, P2 e P3 aguardam o dado enviado pelo nó P0. Desta forma, neste caso, tem-se:

$$t'_x(z) < t'_x(P0) + t^{(n)}_e + t^{(n)}_t, \quad z = 1, 2, 3$$

O tempo de execução de trecho de código em cada nó de processamento é dado por:

$$t_{\text{execução}}(z) = t'_x(P0) + t^{(n)}_e + t^{(n)}_t + t^{(n)}_r + t'_y(z), \quad z = 1, 2, 3$$

$$t_{\text{execução}}(P0) = t'_x(P0) + t^{(n)}_e + t'_y(P0)$$

Quanto ao segundo caso, é analisada a situação em que o dado a ser utilizado na execução do trecho de código já se encontra no seu *buffer* local de recepção, isto é:

$$t'_x(z) > t'_x(P0) + t^{(n)}_e + t^{(n)}_t, \quad z = 1, 2, 3$$

Deste modo, os tempos de execução do trecho de código nos nós de processamento são calculados por:

$$t_{\text{execução}}(z) = t'_x(z) + t^{(n)}_r + t'_y(z), \quad z = 1, 2, 3$$

#### 4.5.4.1 Análises em Função de Tamanho de Mensagem e Número de Nós de Processamento

Numa operação de comunicação broadcast, uma mensagem com  $n$  dados é enviada a todos os nós de processamento do ambiente. Esta mensagem é colocada num *buffer* interno do chaveador (*switch*), e transferida seqüencialmente, dentro do

chaveador, para a porta de recepção conectada a cada nó de processamento. Assim, tem-se que:

$k_2$ : fator que caracteriza a latência da rede de comunicação,

$k_3$ : fator que caracteriza a transferência do *buffer* de recepção do nó de processamento enviado ao *buffer* de dados do chaveador,

$k_4$ : fator que caracteriza a largura de banda da rede de comunicação,

$k_5$ : fator que caracteriza a transferência da mensagem para o *buffer* de recepção de cada um dos nós, a partir do chaveador (*switch*),

$k_6$ : fator que caracteriza a transferência interna da mensagem no chaveador, dos *buffer* de dados para as portas de recepção dos ( $\#NO-1$ ) nós de recebimento,

$k_7$ : fator que caracteriza o tratamento inicial para transmitir os dados

$k_8$ : fator que caracteriza a transferência dos dados do *buffer* de dados a porta de recepção do nó de processamento no chaveador.

Assim,

$$t_t^{(n)}(\text{rank}) = k_2 + k_3n + k_4n + k_6(\text{rank})n + k_7 + k_8 n$$

$$t_t^{(n)}(\text{rank}) = (k_2 + k_7) + (k_3+k_4+k_8)n + k_6 (\text{rank})n$$

Seja:

$$k'_1 = k_2 + k_7 \text{ e}$$

$$k'_2 = k_3 + k_4 + k_8$$

Tem-se:

$$t_t^{(n)}(\text{rank}) = k'_1 + k'_2 n + k_6(\text{rank})n$$

Para o pior caso, tem-se:

$$t_t^{(n)}(\text{rank}) = k'_1 + k'_2 n + k_6(\#\text{NO}-1)n$$

onde: rank = 1, 2, ..., #NO-1.

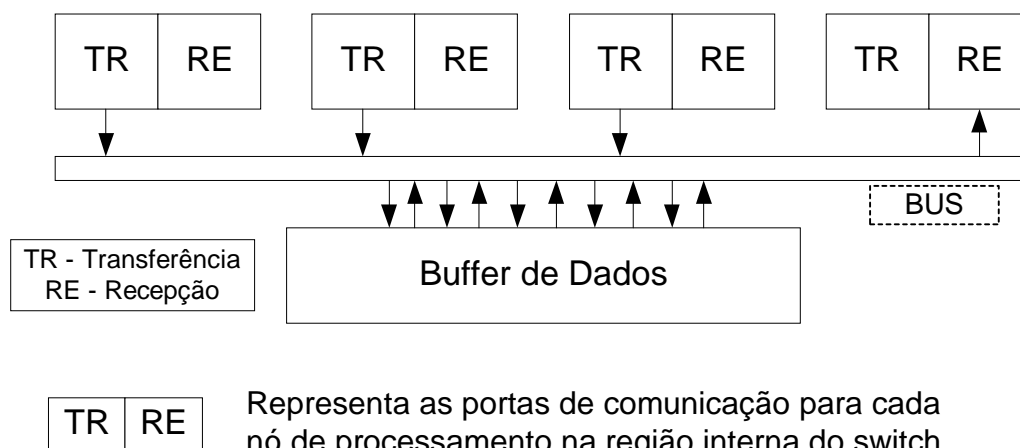


Figura 25. Ilustração dos passos de execução do comando Broadcast dentro de um switch.

Desta forma, da fórmula do tempo de execução para o caso (a), tem-se no nó de processamento P2:

$$\begin{aligned} t_{\text{execução}}(z) &= t'_x(\text{P0}) + t_e^{(n)} + t_t^{(n)} + t_r^{(n)} + t'_y(z), \quad z = 1, 2, 3 \\ &= t'_x(\text{P0}) + t'_y(z) + k_3 n + k_4 n + [k_1 + k_5(\#\text{NO}-1)n + k_2 n] \\ &= t'_x(\text{P0}) + t'_y(z) + [k_3 + k_4 + k_2 + k_5(\#\text{NO}-1)]n + k_1 \end{aligned}$$

A expressão anterior pode ser dada por:

$$t_{\text{execução}}(z) = t'_x(\text{P0}) + t'_y(z) + kn + k_1, \text{ para } z = 1, 2, 3$$

Para o nó de processamento P0, a expressão é dada por:

$$t_{\text{execução}}(\text{P0}) = t'_x(\text{P0}) + t^{(n)}_e + t'_y(\text{P0}) = t'_x(\text{P0}) + t'_y(\text{P0}) + k_3n.$$

Analisando o caso (b), o tempo no nó de processamento P1, P2 e P3 é

$$t_{\text{execução}}(z) = t'_x(z) + t^{(n)}_r + t'_y(z), \quad z = 1, 2, 3 \Rightarrow$$

$$t_{\text{execução}}(z) = t'_x(z) + t'_y(z) + k_4n, \quad z = 1, 2, 3$$

Enquanto o tempo de execução no nó de processamento P0 é dado por:

$$t_{\text{execução}}(\text{P0}) = t'_x(\text{P0}) + t^{(n)}_e + t'_y(\text{P0}) \Rightarrow$$

$$t_{\text{execução}}(\text{P0}) = t'_x(\text{P0}) + t'_y(\text{P0}) + k_3n$$

#### 4.5.5 Análise da Operação de Comunicação *Scatter*

O comando `MPI_Scatter()` é uma operação de comunicação coletiva do tipo um-para-todos, onde diferentes dados são enviados para cada um dos outros nós de processamento do ambiente computacional, definidos na variável `MPI_Comm`. Deste modo, o nó *root* distribui a todos os outros nós a mesma quantidade de dados, que são partes do conjunto de dados enviados pelo nó *root*. A figura 26 mostra o funcionamento desta operação, num sistema de estações de trabalho com quatro nós de processamento.

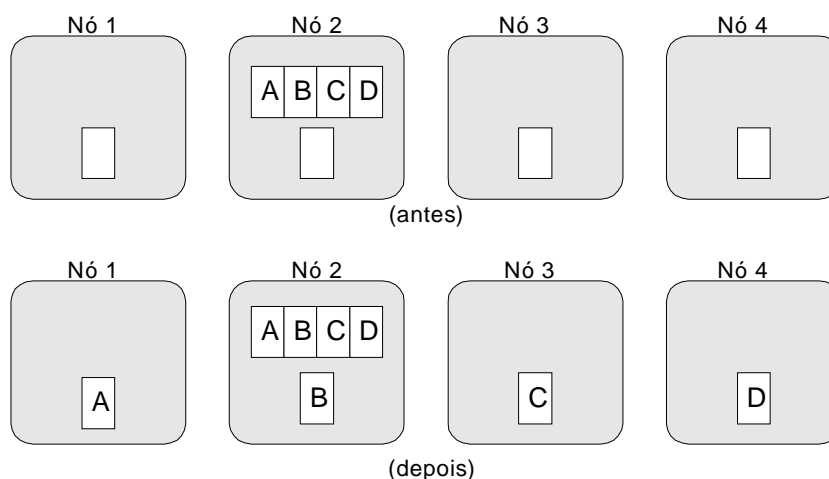


Figura 26. Esquema da movimentação dos dados da Operação `MPI_Scatter()`.

Dois casos típicos que acontecem em programas paralelos são analisados a seguir e ilustrados pelas figuras 27(a) e 27(b).

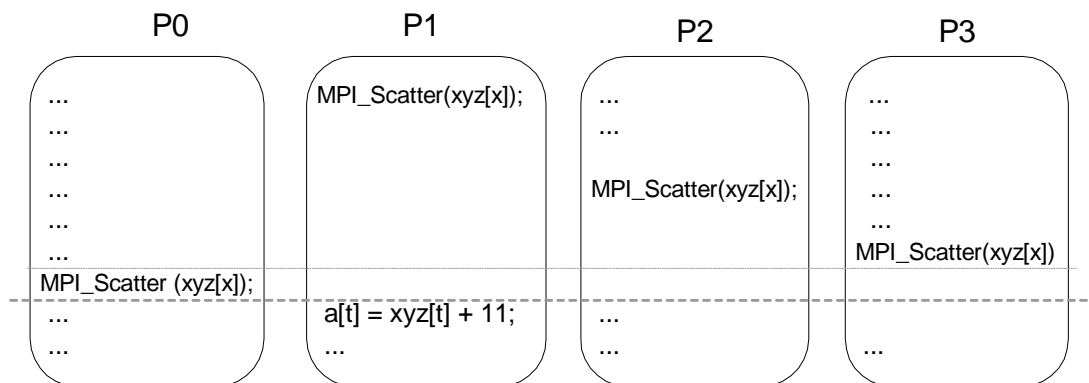


Figura 27(a). Caso 1 de estudo da operação de comunicação `MPI_Scatter`.

A figura 27(a) ilustra o caso em que os nós P1, P2 e P3 já aguardam os dados relativos a operação `MPI_Scatter()`, a ser executado no nó P0. Neste caso, a execução dos códigos nos nós P1, P2 e P3 fica na espera, aguardando o nó P0 executar a operação `MPI_Scatter()`.

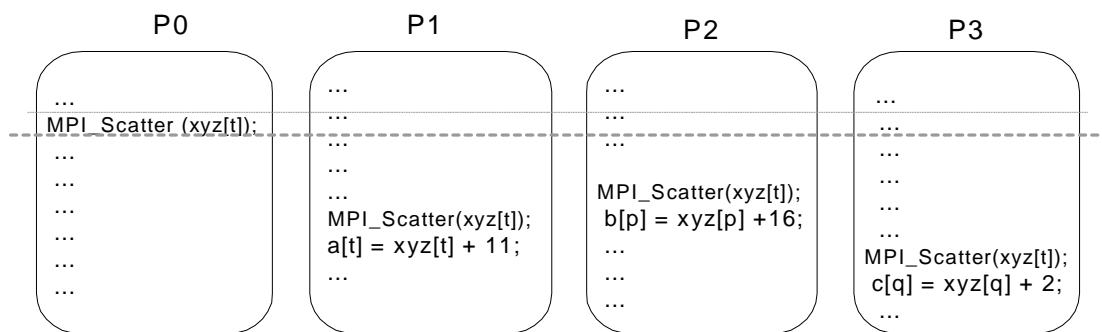


Figura 27(b). Caso 2 de estudo da operação de comunicação MPI\_Scatter.

A figura 27(b) ilustra o caso em que os nós P1, P2 e P3 já têm os dados relativos à distribuição feita pelo nó P0 na operação MPI\_Scatter() ao atingir a linha de código correspondente à operação MPI\_Scatter(), não havendo espera para a continuidade do processamento. Neste caso, é feita somente uma transferência de dados, do *buffer* de recepção de cada nó para sua memória local.

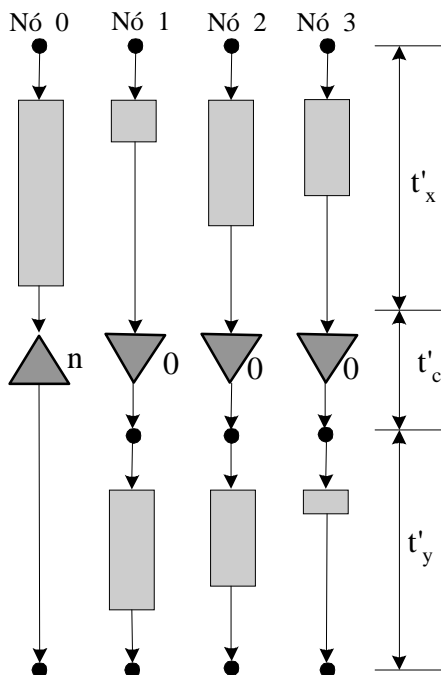


Figura 28. Representação em DP\*Graph do estudo da operação MPI\_Scatter().



Sejam os nós de processamento P1, P2 e P3 nós receptores, enquanto o nó de processamento P0 é o nó envidador de dados, ou seja, o nó *root*. No primeiro caso, mostrado anteriormente pela figura 27(a), os nós de processamento P1, P2 e P3 aguardam o dado enviado pelo nó P0.

Seja #NO o número total de nós de processamento no sistema de estações de trabalho, envolvido na execução do programa paralelo. Desta forma, tem-se:

$$T'_x(Z) < T'_x(P0) + T^{[(\#NO-1)*n]/(\#NO)}_e + T^{[(\#NO-1)*n]/(\#NO)}_t, \quad z = P1, P2, P3$$

Tem-se o fator  $[(\#NO-1)*n]/(\#NO)$ , pois numa operação de comunicação coletiva `MPI_Scatter()`, o nó de processamento *root* mantém o correspondente  $1/(\#NO)$  avos do número total de dados (denominado de  $n$ ), e envia a cada um dos outros nós de processamento do ambiente  $1/(\#NO)$  de dados. Desta forma, dado que são  $(\#NO-1)$  nós de processamento para os quais o nó *root* enviou os dados, a quantidade total de dados enviado pelo nó *root* é o valor apresentado neste fator.

O tempo de execução do trecho de código dos nós de processamento é dado por:

$$T_{\text{execução}}(z) = T'_x(P0) + T^{[(\#NO-1)*n]/(\#NO)}_e + T^{[(\#NO-1)*n]/(\#NO)}_t + T^{(n)/(\#NO)}_r + T'_y(z), \quad z = P1, P2, P3$$

Agora, é analisado o segundo caso, em que o dado a ser utilizado na execução do trecho de código já se encontra no seu *buffer* local de recepção, isto é:

$$T'_x(P0) + T^{(n)}_e + T^{(n)}_t < T'_x(z) + T^{[(\#NO-1)*n]/(\#NO)}_e + T^{[(\#NO-1)*n]/(\#NO)}_t, \quad z = P1, P2, P3$$

Deste modo, o tempo de execução do trecho de código é:

$$T_{\text{execução}}(z) = T'_x(z) + T^{(n)/(\#NO)}_r + T'_y(z), \quad z = P1, P2, P3$$

#### 4.5.5.1 Análises com Variação no Tamanho das Mensagens e Nós de Processamento

Considerando que cada nó de processamento, exceto o nó P0, receberá do nó P0  $n/\#NO$  quantidade de dados, e que o comando de comunicação `MPI_Scatter` é composto de  $(\#NO-1)$  mensagens, cada uma enviada para um dos outros  $(\#NO-1)$  nós de processamento e que a transmissão desta operação não é bloqueante, e sejam as constantes:

$k_1$ : reflete o tempo de transferência de um dado da memória para o *buffer* de envio de dentro do nó P0.

$k_2$ : fator que reflete na latência.

$k_3$ : inverso da largura de banda.

$k_4$ : transferência dos dados na *switch* da porta de transferência de P0 para o *buffer* de dados e a seguir para a porta de recepção correspondente a cada nó.

$k_5$ : latência interna à *switch* para transferência dos dados do *buffer* de dados para a porta de recepção correspondente a cada nó.

$k_6$ : transferência dos dados em cada nó receptor do seu *buffer* de recepção para a memória.

Tem-se:

$$T_e = k_1 (\#NO-1) (n / \#NO)$$

$$T_{ti} = k_2 + k_3 (\#NO - 1) n / \#NO + k_4 (i - 1) n / \#NO + k_5 (i - 1)$$

Considerando o pior caso, onde  $i = \#NO$ , tem-se:

$$T_t = k_2 + k_3 (\#NO - 1) n / \#NO + k_4 (\#NO - 1) n / \#NO + k_5 (\#NO - 1)$$

Sintetizando, tem-se:

$$T_t = k_1' + k_2' (\#NO - 1) n / \#NO + k_3' (\#NO - 1)$$

$$T_r = k_6 n / \#NO$$

Resumindo, tem-se:

$$T_e = c_1 (\#NO - 1) n / \#NO$$

$$T_t = c_2 + c_3 (\#NO - 1) n / \#NO + c_4 (\#NO - 1)$$

$$T_r = c_5 n / \#NO$$

E, substituindo nas equações para cálculo do tempo de execução nos casos analisados anteriormente, obtém-se as equações em função de tamanho de mensagem **n** e número de processadores **p**.

#### 4.5.6 Análise da Operação de Comunicação *Gather*

O comando de comunicação `MPI_Gather()` é uma operação de comunicação coletiva do tipo todos-para-um, onde diferentes dados localizados nos diferentes nós de processamento são enviados a um nó de processamento, o nó *root* que reúne todos os dados. Em outras palavras, é uma operação inversa do comando de comunicação `MPI_Scatter()`. A figura 29 mostra o funcionamento desta operação, num sistema de estações de trabalho, com quatro nós de processamento. No caso, os nós de processamento emissores são P0, P2 e P3, enquanto o nó receptor é o P1.

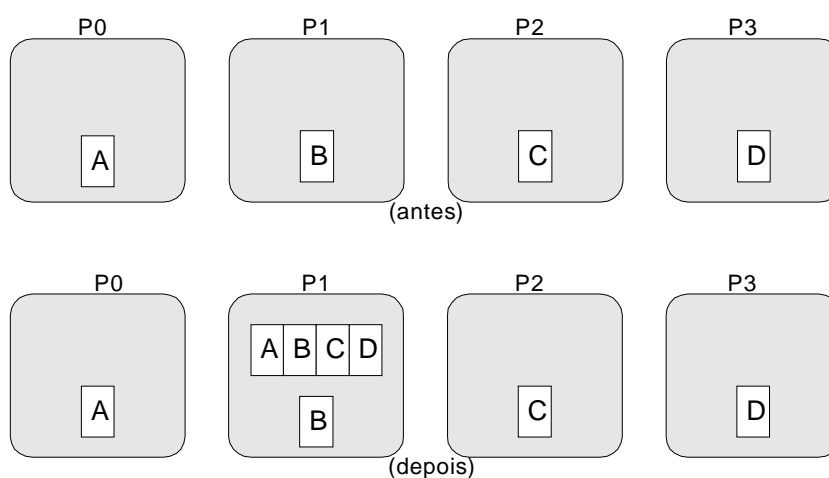


Figura 29. Esquema da movimentação dos dados da Operação `MPI_Gather()`.

Dois casos de ocorrências comuns em execução de programas paralelos são analisados a seguir, e ilustrados pelas figuras 30(a) e 30(b).

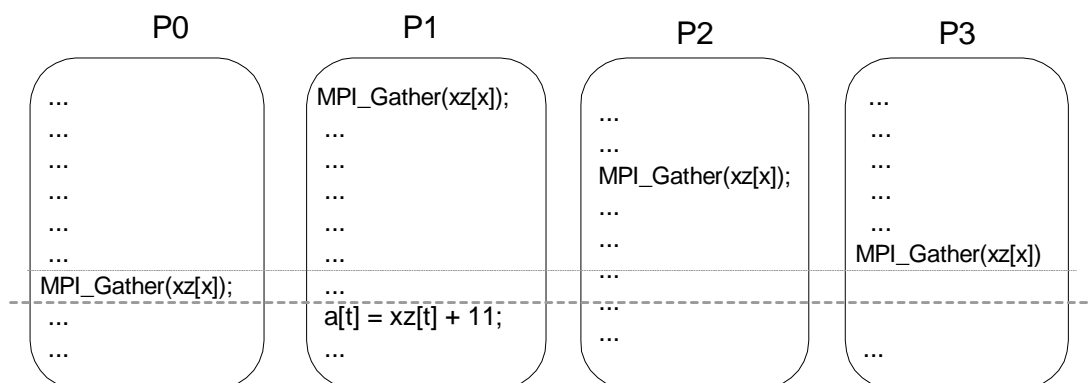


Figura 30(a). Caso 1 de estudo da operação de comunicação MPI\_Gather.

A figura 30(a) ilustra o caso em que no momento em que o nó P0 executar a função de comunicação MPI\_Gather, os nós P1, P2 e P3 já enviaram seus dados ao nó de processamento P0. No nó P0, então, os dados já presentes no *buffer* local de recepção são transferidos para sua memória local, continuando a execução do trecho de código.

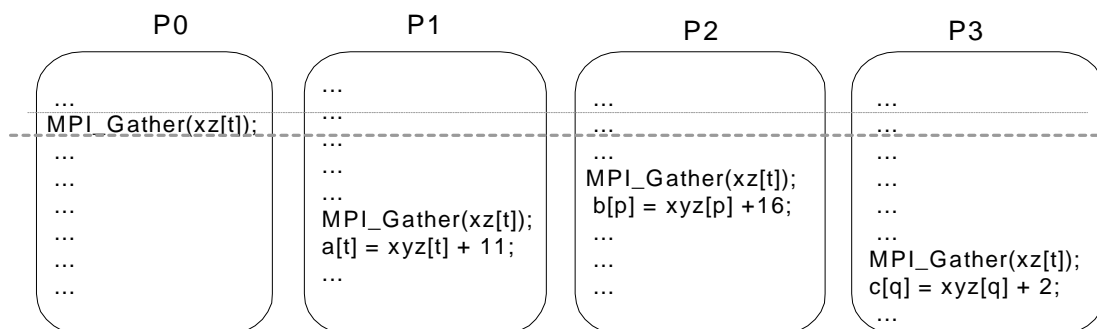


Figura 30(b). Caso 2 de estudo da operação de comunicação MPI\_Gather.

A figura 30(b) ilustra o caso em que os nós P1, P2 e P3 continuam o processamento do trecho de código; enquanto o nó de processamento P0 fica na espera e aguarda os dados provenientes dos nós de processamento P1, P2 e P3, ao

atingir a linha de código que contém a chamada da função de comunicação `MPI_Gather()`.

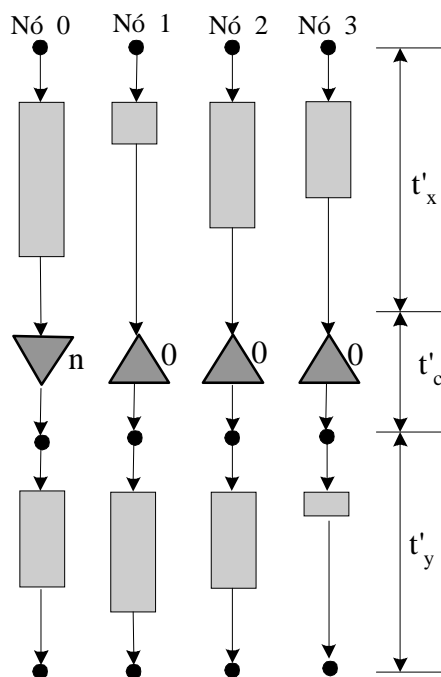


Figura 31. Representação em DP\*Graph para o estudo da operação `MPI_Gather()`.

Análises desta operação de comunicação serão feitas a seguir, auxiliada pela ilustração apresentada na figura 31. Sejam os nós de processamento P1, P2 e P3 os nós emissores, enquanto o nó de processamento P0 é o nó denominado de receptor de dados.

Seja #NO o número total de nós de processamento no sistema de estações de trabalho, envolvidos na execução do programa paralelo.

Será analisado o caso 30(a). Tem-se:

$$T'_x(P0) > T'_x(Z) + T^{(n)/(\#NO)}_e + T^{(n)/(\#NO)}_t, \quad Z = P1, P2, P3$$

Tem-se o fator  $(n)/(\#NO)$ , pois numa operação do comando `MPI_Gather`, os dados enviados por cada nó de processamento emissor corresponde a  $1/(\#NO)$  avos do total de dados  $n$ , e sendo que o nó de processamento receptor (*root*) recebe dados de  $(\#NO-1)$  nós de processamento emissores, o nó de processamento *root* recebe a

quantidade total de  $[(\#NO-1)*n]/(\#NO)$  dados. Vale notar que, ao atingir o comando *MPI\_Gather* do trecho de código no nó de processamento P0, os dados já se encontram disponíveis no seu *buffer* local de recepção.

O tempo de execução do trecho de código é dado por:

$$T_{\text{execução}} = T'_x(P0) + T^{[(\#NO-1)*n]/(\#NO)}_r + T'_y(P0).$$

Agora, é analisado o segundo caso, de acordo com a figura 30(b), em que o dado a ser utilizado na execução do trecho de código não se encontra no seu *buffer* local de recepção, isto é:

$$T'_x(P0) < T'_x(Z) + T^{n/(\#NO)}_e + T^{n/(\#NO)}_t, \quad Z = P1, P2, P3$$

Deste modo, o tempo de execução do trecho de código para cada um dos nós de processamento é:

$$T_{\text{execução}}(z) = T'_x(z) + T^{n/(\#NO)}_e + T^{n/(\#NO)}_t + T'_y(z), \quad z = P1, P2, P3$$

#### 4.5.6.1 Análises Variando Número de Mensagens e Número de Nós de Processamento

Considerando que o nó P0 recebe  $n/\#NO$  dados dos demais nós, e que o comando de comunicação *MPI\_Gather* é composto de  $(\#NO-1)$  mensagens enviadas para o nó P0, valendo observar que a transmissão desta operação não é bloqueante, e considerando que cada nó de processamento, exceto o nó P0, enviará ao nó P0  $n/\#NO$  quantidade de dados e que o comando de comunicação *MPI\_Gather* é composto de  $(\#NO-1)$  mensagens, cada uma recebida de um dos outros  $(\#NO-1)$  nós de processamento e vale observar que a transmissão desta operação não é bloqueante, e sejam as constantes:

$k_1$ : caracteriza a transferência do dado da memória para o *buffer* ou a transmissão dentro de cada nó.

$k_2$ : latência de comunicação do nó para o switch.

$k_3$ : inverso da largura de banda

$k_4$ : transferência interna de cada porta para o *buffer* de dados e deste para a porta de recepção correspondente ao nó  $P_0$ .

tem-se:

$$T_e = k_1 * n / \#NO \text{ (em cada nó enviado)}$$

Considerando o pior caso, onde todos os nós emissores enviam simultaneamente, tem-se:

$$T_t = k_2 + (k_3 * n / \#NO) + (k_4 * (\#NO-1) * n / \#NO)$$

$$T_r = k_5 * (\#NO-1) * n / \#NO$$

#### 4.5.7 Análise da Operação de Comunicação *Reduce*

O comando de comunicação `MPI_Reduce()` é uma operação de comunicação coletiva, onde cada processo envia um operando, e sobre os operandos, é realizada uma operação associativa ou comutativa (por exemplo, soma, máximo, mínimo, subtração, operações lógicas bit-a-bit, etc.), e o resultado pode ser distribuído para somente um processo (*root*) ou para todos os processos envolvidos nesta operação. Quando são envolvidos todos os processos do ambiente computacional definido, esta operação é também denominada “Redução N/N”, gossiping, ou *allreduce* [SNIR96, GROP96]

##### 4.5.7.1 Análises Variando Tamanho de Mensagens e Número de Nós de Processamento

É uma operação de comunicação coletiva, onde cada nó envia para o nó 0 um único dado. O comportamento, sob o aspecto de comunicação, é similar à operação

MPI\_Gather, com n igual a #NO, exceto que envolve uma operação sobre os dados recebidos. Tem-se então:

$$T_e = k_1$$

$$T_t = k_2 + k_3 + k_4 * (\#NO - 1)$$

$$T_t = k_2' + k_3' * \#NO$$

$$T_r = k_5 * (\#NO - 1)$$

Resumindo, tem-se:

$$T_e = c_1$$

$$T_t = c_2 + c_3 * \#NO$$

$$T_r = c_4 * (\#NO - 1)$$

#### 4.5.7.2 Caso Particular: Operação de Comunicação All Reduce

No caso em que todos os nós de processamento executam a operação All\_reduce praticamente de forma simultânea ou o nó de processamento 0 executa esta operação antes dos demais, e sejam:

$T_{c0}$ : tempo de comunicação gasto no *root*.

$T_{ci}$ : tempo de comunicação gasto nos demais nós.

$T_e + T_t + T_r$ : referentes ao reduce.

$T_{e'}$ : referente ao broadcast (n=1)

Tem-se:

$$T_{c0} = T_e + T_t + T_r + T_{e'}$$

$$T_e = c_1$$

$$T_t = c_2 + c_3 * \#NO$$

$$T_r = c_4 * (\#NO - 1)$$

$$T_{e'} = c_5$$



$$T_{c0} = c_1 + c_2 + c_3 * \#NO + c_4 * (\#NO - 1) + c_5$$

$$T_{c0} = (c_1 + c_2 - c_4 + c_5) + (c_3 + c_4) * \#NO$$

$$T_{c0} = c_1' + c_2' * \#NO$$

Resumindo:

$$T_{c0} = c_1 + c_2 * \#NO$$

Para os demais nós:

$$T_{cj} = T_e + T_f' + T_r'$$

$$T_e = c_1$$

$$T_f' = c_2 + c_3 + c_4 * (\#NO - 1)$$

$$T_r' = c_5$$

$$T_{cj} = c_1 + c_2 + c_3 + c_4 * (\#NO - 1) + c_5$$

$$T_{cj} = (c_1 + c_2 + c_3 + c_5) + c_4 * (\#NO - 1)$$

$$T_{cj} = c_1' + c_4 * (\#NO - 1)$$

Resumindo:

$$T_{cj} = c_1 + c_2 * (\#NO - 1)$$

$T_e$ : referente ao reduce.

$T_f' + T_r'$ : referente ao broadcast (n=1)

#### 4.5.8 Análise da Operação de Comunicação All-to-All

Considerando o caso em que a operação de comunicação All-to-All é executada simultaneamente em todos os nós, tem-se:

$$T_c = T_e + T'_e + T'_f + T'_r$$

$$T_e = c_1 * n$$

$$T_e = c_2 * n$$

$T_e$ : referente ao envio para todos.

$T'_e + T'_f + T'_r$ : referente ao recebimento de dados de todos os nós .

Tomando a equação que calcula o tempo  $t_i$  do broadcast e considerando que os dados podem estar sendo transferidos do *buffer* de transmissão de cada nó até a porta de transmissão do chaveador, simultaneamente, tem-se:

$$T'_t = c_3 + c_4 n + c_5 n (\#NO - 1) + c_6 (\#NO - 1)^2 n + c_7 (\#NO - 1) + c_8 n (\#NO - 1) \Rightarrow$$

$$T'_t = c_3 + c_4 n + (c_5 n + c_7 + c_8 n) (\#NO - 1) + c_6 (\#NO - 1)^2 n \Rightarrow$$

$$T'_t = c_9 n (\#NO - 1)$$

Desta forma, tem-se:

$$T_c = c_1 n + c_2 n + c_3 + c_4 n + (c_5 + c_8) n (\#NO - 1) + c_7 (\#NO - 1) + c_6 (\#NO - 1)^2 n + c_9 n (\#NO - 1)$$

$$T_c = (c_1 + c_2 + c_4) n + c_3 + (c_5 + c_8 + c_9) n (\#NO - 1) + c_7 (\#NO - 1) + c_6 (\#NO - 1)^2 n$$

Ou seja,

$$T_c = c_1 n + c_2 + c_3 n (\#no - 1) + c_4 (\#no - 1)^2 + c_6 (\#no - 1)^2 n .$$

## 4.6 Predição de Desempenho

Uma predição de desempenho é feita a partir de análises e estudos baseados nos dados experimentais obtidos, podendo ser um trecho de código, assim como um programa de uma aplicação. Após as análises feitas e estruturado o comportamento

de trechos de código em funções, como já mostrado nas seções anteriores, é possível efetuar, assim, estudos e pesquisas de predição de desempenho do comportamento de um código específico, variando o número de processadores  $p$  e o tamanho do problema  $n$ .

Desta forma, um dos objetivos deste trabalho é apresentar esta técnica que permite verificar o comportamento para uma variação de tamanho do problema  $n$  e prever o tempo de execução para um determinado tamanho do problema, utilizando um número fixo de processadores. Ou, estimar tempos de execução para um número variável de nós de processamento, para um tamanho fixo do problema.

Serão apresentados exemplos para os dois casos.

**Caso 1.** Para um tamanho de problema variável  $n$  e um número de nós fixos  $p$ . Para exemplificar, serão utilizados as equações de estudo do caso (a) da operação de comunicação *MPI\_Bcast()*. As equações provenientes deste caso (a) são:

$$t_{\text{execução}}(z) = t'_x(P0) + t'_y(z) + kn + k_1, \text{ para } z = 1, 2, 3$$

$$t_{\text{execução}}(P0) = t'_x(P0) + t^{(n)}_e + t'_y(P0) = t'_x(P0) + t'_y(P0) + k_3n.$$

Para este estudo, as equações ficam:

$$t_{\text{execução}}(z) = t'_x(P0) + t'_y(z) + kn + k_1, \text{ para } z = 1, 2, 3,$$

$$t_{\text{execução}}(P0) = t'_x(P0) + t'_y(P0) + k_3n,$$

Onde  $k$ ,  $k_1$  e  $k_3$  são constantes.

**Caso 2.** Para um tamanho de problema  $n$  fixo e número de nós  $p$  variável. Para ilustrar este caso, são utilizadas as equações de estudo do caso (a) da operação de comunicação *MPI\_Bcast()*. As equações provenientes deste caso (a) são:

$$t_{\text{execução}}(z) = t'_x(P0) + t'_y(z) + kn + k_1, \text{ para } z = 1, 2, 3$$

$$t_{\text{execução}}(P0) = t'_x(P0) + t^{(n)}_e + t'_y(P0) = t'_x(P0) + t'_y(P0) + k_3n.$$

Para este estudo, as equações ficam:

$$t_{\text{execução}}(z) = t'_x(\text{P0}) + t'_y(z) + c_1, \text{ para } z = 1, 2, 3, \quad c_1 = kn + k_1,$$

$$t_{\text{execução}}(\text{P0}) = t'_x(\text{P0}) + t'_y(\text{P0}) + c_2, \quad c_2 = k_3n$$

Onde  $c_1$  e  $c_2$  são constantes.

As técnicas de análise de desempenho, propostas neste trabalho, podem ser aplicadas a trechos de código de um programa paralelo, assim como poderá ser o programa paralelo de uma aplicação inteira. Conseqüentemente, a predição de desempenho poderá ser feita nas situações descritas para a análise de desempenho.

Como o resultado obtido é correspondente a cada um dos nós de processamento, o tempo de execução do trecho de código ou da aplicação é dado por:

$$T_{\text{aplicação}} = \max\{t_i\}, \text{ sendo } t_i = \sum_{k=\text{todos}} t_{i,k},$$

onde  $t_{i,k}$  = trecho  $k$  do nó de processamento  $i$  executado entre duas comunicações, incluindo o tempo gasto nas comunicações.

## 4.7 Estudo de Caso: Modelagem da aplicação IS (NAS)

Encontra-se no apêndice I uma relação de todos os programas pertencentes ao conjunto de programas de *benchmark* NAS, com suas breves descrições e no apêndice II, uma listagem completa do programa IS (*Integer Sort – Parallel Sort over Small Integers*), do conjunto de programas de *benchmark* NAS.

O objetivo desta aplicação é ordenar um conjunto de números em paralelo. Os números são gerados por um algoritmo de geração de números seqüencialmente e devem ser distribuídos uniformemente na memória no início.

### □ Definição

Seja uma seqüência de números,  $K_0, K_1, K_2, \dots, K_{n-1}$ . Esta seqüência é dita **ordenada** se os números estão em ordem crescente, isto é,  $K_0 \leq K_1 \leq \dots \leq K_i \leq K_{i+1} \leq K_{i+2} \leq \dots \leq K_{n-1}$ .

### □ Mapeamento em Memória

Este programa de *benchmark* basicamente tem como objetivo ordenar uma seqüência de  $N$  números. A seqüência de números, uma vez gerada por um algoritmo seqüencial, é mapeada na memória dos nós de processamento do sistema de estações de trabalho (dependendo do modelo de sistema computacional, quanto à localização de memória, o algoritmo de mapeamento é diferente). As subseqüências da seqüência completa de números, ao serem transferidas para o sistema de memória de cada um dos nós de processamento, estes não são movidas ou modificadas, com exceção quando são solicitados por procedimentos internos ao programa paralelo.

### □ Algoritmo

Num sistema de memória distribuída, com  $p$  unidades de memória distintas, cada unidade de memória é carregada inicialmente com  $N_p$  números, onde:

$$N_p = N / p, \text{ onde } N \text{ é o número total de números.}$$

Seja:

$A_i$  :  $i$ -ésimo número dentro de um sistema de memória,

$P_j$  :  $j$ -ésima unidade de memória,

Então,

$A_i \cap P_j$  significa  $i$ -ésimo número na  $j$ -ésima unidade de memória.

Quando  $N$  é dividida por  $p$ , e não ocorrer uma divisão exata, então, as unidades de memória  $\{P_j \mid j = 0, 1, 2, \dots, p-2\}$  são carregadas com  $N_p$  números cada, enquanto a unidade de memória  $P_{p-1}$  é carregada com  $N_{pp}$  números, onde:

$$N_p = \lfloor N/p + 0.5 \rfloor$$

$$N_{pp} = N - (p-1)N_p.$$

#### □ Distribuição da Seqüência de Números

Após a seqüência ter sido gerada através de um algoritmo seqüencial, é necessário distribuir os números desta seqüência nos  $p$  nós de processamento do sistema de estações de trabalho.

Supondo-se que  $N$  é divisível por  $p$ , isto é, o resto da divisão de  $N$  por  $p$  é zero. Desta forma, numa unidade de memória, este contém os números  $[A_i, A_{i+N_p-1}]$ ; nos sistemas de estações de trabalho, pode-se dizer que foram alocados em  $N_p$  nós de processamento, e que em cada um dos nós de processamento,  $A_i$  significa  $i$ -ésimo número da unidade de memória de um determinado nó de processamento.

Assim, a seqüência de números  $K_0, K_1, K_2, \dots, K_{N-1}$  é mapeada no sistema distribuído da seguinte forma:

$$P_k \cap A_{i+j} \leftarrow \text{MEM}(K_{kN_p+j}), \text{ para } j = 0, 1, 2, \dots, N_p-1 \text{ e } k = 0, 1, 2, \dots, p-1$$

onde  $\text{MEM}(K_{kN_p+j})$  refere-se ao número  $K_{kN_p+j}$  da seqüência gerada.

Ainda, se  $N$  não é divisível por  $p$ , a distribuição dada acima deve ser modificada, e adotar  $k = p-1$ , e

$$P_{p-1} \cap A_{i+j} \leftarrow \text{MEM}(K_{(p-1)N_p+j}), \text{ para } j = 0, 1, 2, \dots, N_{pp}-1.$$

## 4.7.1 Procedimento RANK da aplicação IS

Este procedimento dentro do programa de benchmark IS tem como objetivo determinar a redistribuição dos números, trocando os números entre os processos, à medida que avança a ordenação.

### 4.7.1.1 Representação do Procedimento

Segue na figura 32 uma representação em baixo nível do procedimento RANK. O procedimento foi dividido em trechos para que sejam feitas as análises destes trechos individualmente.

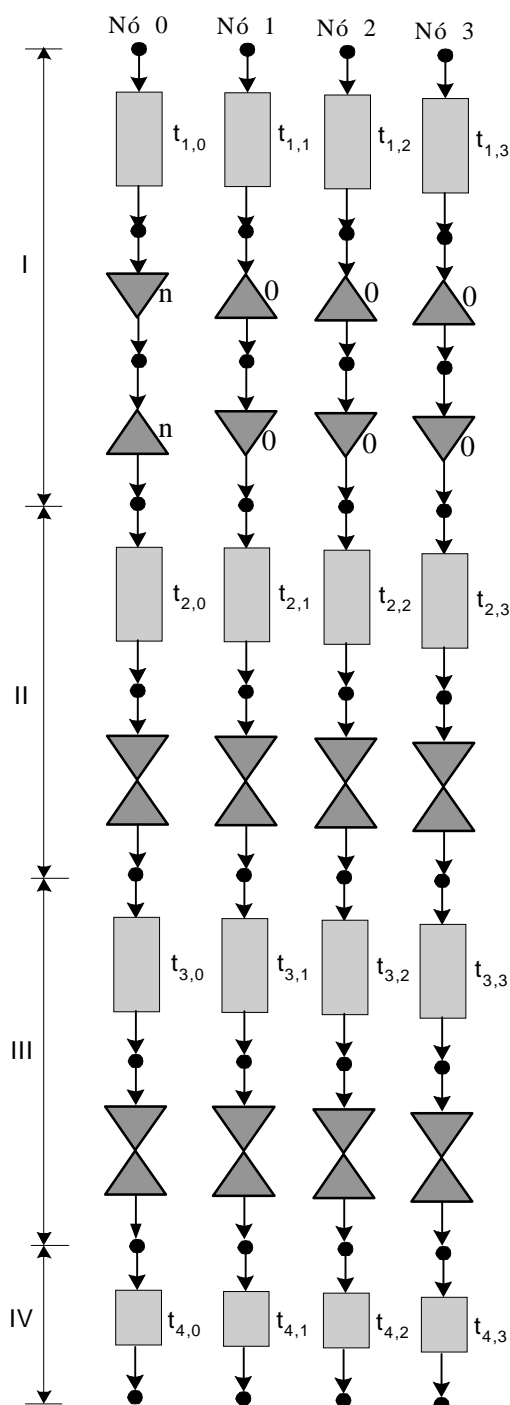


Figura 32. Representação do Procedimento RANK do programa de avaliação IS.

#### 4.7.1.2 Análises do Procedimento RANK

Supondo que todos os nós têm a mesma capacidade de processamento e como todos executam o mesmo código entre as comunicações, será considerado que os tempos



$T_{i,j}$  são iguais para todos os  $j$ . E, de fato, isto acontece, pois o ambiente de testes a ser utilizado tem os nós de processamento idênticos uns aos outros.

### Trecho I:

$T_{1,j}$  = função polinomial (TOTAL\_KEYS).

O tamanho do problema reflete em NUM\_KEYS  $\Rightarrow$  em dois loops  
 $NUM\_KEYS = TOTAL\_KEYS / \#NO$ , onde  $\#NO$  é o número de nós de processamento.

O grau do polinômio de  $T_{1,j}$  é 1  $\Rightarrow c_1 + c_2 * TOTAL\_KEYS / \#NO$

Considerando que  $T_{1,0} = T_{1,j}$ , tem-se que  $T_{1,0} = c_1 + c_2 * TOTAL\_KEYS + t_{c0}$

A execução da operação de comunicação All\_Reduce, ocorre no *root*:

$$t_{c0} = c_3 + c_4 * \#NO$$

$$T_{1,0} = c_1 + c_2 * TOTAL\_KEYS / \#NO + c_3 + c_4 * \#NO$$

$$T_{1,0} = (c_1 + c_3) + c_2 * TOTAL\_KEYS + c_4 * \#NO$$

$$T_{1,0} = c_1 + c_2 * TOTAL\_KEYS / \#NO + c_3 * \#NO$$

Enquanto nos demais nós, ocorre:

$$t_{cj} = c_3 + c_4 * (\#NO - 1)$$

$$T_{1,j} = c_1 + c_2 * TOTAL\_KEYS / \#NO + c_3 * \#NO$$

### Trecho II:

$T_{2,j}$  = constante, considerando a variação de TOTAL\_KEYS.

$$T_{2,j} = c_4$$

$$T_{II,j} = c_4 + T_c$$

Este trecho é finalizado por uma operação de comunicação *All-to-All*, com uma mensagem de tamanho igual a 1.

$$T_c = c_1' + c_2' + c_3' * (\#NO - 1) + c_4' * (\#NO - 1) + c_6' * (\#NO - 1)^2$$

$$T_c = c_1' + c_2' * (\#NO - 1) + c_3' * (\#NO - 1)^2$$

$$T_{II,j} = c_4 + c_5 + c_6 * (\#NO - 1) + c_7 * (\#NO - 1)^2$$

$$T_{II,j} = c_4 + c_5 * (\#NO - 1) + c_6 * (\#NO - 1)^2$$

**Trecho III:**

$T_{3,j}$  = constante em relação à variação de TOTAL\_KEYS

$$T_{3,j} = c_7$$

$$T_{III,j} = T_{3,j} + T_c$$

Este trecho é finalizado por uma operação de comunicação *All-to-All* numa mensagem de tamanho *Send\_Count*, que não depende de TOTAL\_KEYS.

$$T_c = c_1' \text{Send\_Count} + c_2' + c_3' \text{Send\_Count} (\#NO - 1) + c_4' (\#NO - 1) + c_6' (\#NO - 1)^2 \text{Send\_Count}$$

Considerando que *Send\_Count* é constante em relação à variação de TOTAL\_KEYS, tem-se que:

$$T_c = c_1' + c_2' (\#NO - 1) + c_3' (\#NO - 1)^2$$

$$T_{III,j} = c_7 + c_8 (\#NO - 1) + c_9 (\#NO - 1)^2$$

$$T_j = c_1 + c_2 * \text{TOTAL\_KEYS} / \#NO + c_3 * \#NO + c_4 + c_5 * (\#NO - 1) + c_6 * (\#NO - 1)^2 + c_7 + c_8 * (\#NO - 1) + c_9 * (\#NO - 1)^2 + c_{10} \Rightarrow$$

$$T_j = c_1' + c_2 * \text{TOTAL\_KEYS} / \#NO + (c_6 + c_9) * \#NO^2 + (c_5 - 2 * c_6 + c_8 - 2 * c_9) * \#NO \Rightarrow$$

$$T_j = c_1 + c_2 * \text{TOTAL\_KEYS} / \#NO + c_3 * \#NO + c_4 * \#NO^2$$

#### 4.7.1.3 Estudos de Predição de Desempenho

São apresentados, a seguir, estudos de predição de desempenho deste procedimento, em dois casos. No primeiro, o número de nós de processamento é fixo enquanto varia-se o tamanho do problema. No segundo caso, são efetuados estudos fixando o tamanho do problema, enquanto é variado o número de nós de processamento.

**Caso 1.** Considerando um número de nós de processamento #NO constante e variando o tamanho do problema TOTAL\_KEYS, tem-se:

$$T_0 = c_1 + c_2 * \text{TOTAL\_KEYS}$$

$$T_j = c_3 + c_4 * \text{TOTAL\_KEYS} \quad (j=1 \text{ a } \#NO - 1)$$

$$T = \max(T_i), \quad T_i=0 \text{ a } \#NO - 1$$

**Caso 2.** Considerando um tamanho do problema TOTAL\_KEYS fixo e variando o número de nós de processamento #NO, tem-se:

$$T_0 = c_1 + c_2 / \#NO + c_3 * \#NO + c_4 * \#NO^2$$

$$T_j = c_5 + c_6 / \#NO + c_7 * \#NO + c_8 * \#NO^2$$

$$T = \max(T_i), \quad \text{para } T_i=0 \text{ a } \#NO - 1$$

# Capítulo 5

## Implementação, Resultados e Discussão

Foi apresentada, no capítulo anterior, uma modelagem de estruturas de comunicação da interface de passagem por mensagens MPI, operações fundamentais para o processamento distribuído. Sem estas estruturas, não teria sentido falar em processamento paralelo, para sistemas computacionais com memória distribuída.

Os tempos de execução, dados necessários para as análises e predições de desempenho, são obtidos, de forma experimental, a partir de execuções de programas paralelos instrumentados com estruturas de comunicação da interface de passagem por mensagens MPI, variando o tamanho do problema e número de nós de processamento do sistema de estações de trabalho.

Conforme apresentado no capítulo 4 , aplicando-se estas equações, já com os coeficientes definidos, é possível realizar predições de desempenho considerando a variação do número de nós utilizados na execução da aplicação ou a variação do tamanho do problema.

Neste capítulo, são apresentados resultados experimentais que mostram os tempos de execução obtidos experimentalmente e os tempos de execução obtidos aplicando-se a metodologia de predição proposta, para os casos descritos no capítulo 4.

## 5.1 Ambiente de teste

A plataforma utilizada para efetuar os testes é um sistema de rede de estações de trabalho, com 16 nós de processamento, onde cada um dos nós de processamento é composto por um processador *INTEL Celeron* 433 MHz, 128 Mb de memória SDRAM (66MHz), placa de rede Fast-Ethernet (100Mbits/s) Intel Ether-Express Pro (também conhecida como Pro/100). Para realizar as medidas, a rede de interconexão foi totalmente isolada da rede externa, inexistindo assim qualquer tipo de interferência externa, como por exemplo, processos de outros usuários. Para conectar os nós de processamento entre si, utilizou-se um chaveador (*switch*) 3COM SuperStack3300 (de 24 portas), que apresenta características de funcionamento similares às aquelas apresentadas anteriormente. Os nós de processamento executam *Linux RedHat* versão 6.2, *kernel* versão 2.2.16 e MPI versão LAM 6.4.

## 5.2 Resultados com Operação de Comunicação *Send*

São apresentadas, nesta seção, análises de resultados obtidos experimentalmente, e a partir das análises, são aplicadas as funções da modelagem geral desenvolvidas no capítulo anterior. Feito isso, é obtida então uma função que expressa o comportamento daquela operação de comunicação MPI, para aquele tamanho de mensagem.

Estudos são estruturados e elaborados com o propósito de coletar dados para efetuar as avaliações desta operação de comunicação. Uma vez que todos os dados experimentais obtidos são organizados e analisados, são aplicados às equações resultantes da modelagem desta operação de comunicação apresentada no capítulo anterior, obtendo-se um sistema linear de equações. Após ter resolvido o sistema de

equações, obtém-se uma equação característica desta operação de comunicação MPI para o tamanho da mensagem determinado no trecho de código executado.

É mostrado, no gráfico 1, o comportamento do desempenho da operação de comunicação `MPI_Send`, utilizando os tempos de execução obtidos experimentalmente através de testes, variando o tamanho do problema e o número de nós de processamento no ambiente de teste detalhado anteriormente.

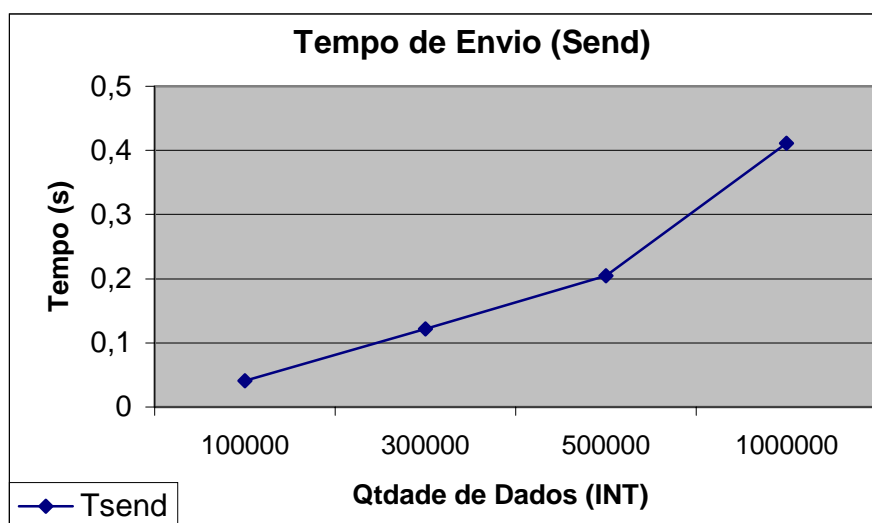


Gráfico 1. Gráfico de tempo de execução da operação `MPI_Send()`.

Aplicando-se as equações obtidas das análises da seção 4.5.3 nos resultados experimentais, obtém-se a seguinte equação que expressa o comportamento desta operação de comunicação:

$$T_{\text{execução}} = k_1 * n \Rightarrow 0,040615 = 100000 * k_1 \Rightarrow k_1 = 4,0615 \times 10^{-7}$$

Foi equacionada e feita uma avaliação nos outros dados obtidos, verificando os diversos resultados. Observou-se que, entre todos os  $k_i$  calculados, a diferença entre o menor e o maior  $k_i$  obtido foi menor a 1%. A equação característica do tempo de execução obtida para esta operação para um tamanho do problema variável  $n$ , considerando o número de nós fixo, é:

$$T_{\text{send}}(n) = 4,0615 \times 10^{-7} * n, \text{ onde } n \text{ representa a quantidade de dados.}$$

A tabela 1 mostra os dados experimentais obtidos, na coluna  $T_{\text{execução}}$ , e na segunda coluna, os dados obtidos efetuando a predição através da função obtida.

O gráfico 2 apresenta resultados da operação de comunicação MPI\_Send. A curva  $T_{\text{execução}}$  corresponde a tempos de execução obtidos experimentalmente. A segunda curva,  $T_{\text{predição}}$ , é obtida utilizando os resultados da predição, realizada aplicando-se a análise analítica desta operação, apresentada com detalhes na seção 4.5.2. A maior diferença entre o tempo de execução real medido, e o tempo de predição, é menor que 1,3%.

Qtidade de Dados (INT)	$T_{\text{execução}}$	$T_{\text{predição}}$
100000	0,040615	4,06E-02
300000	0,121759	1,22E-01
500000	0,204589	2,03E-01
1000000	0,410908	4,06E-01

Tabela 1. Resultados Experimentais e de Predição da Operação Send.

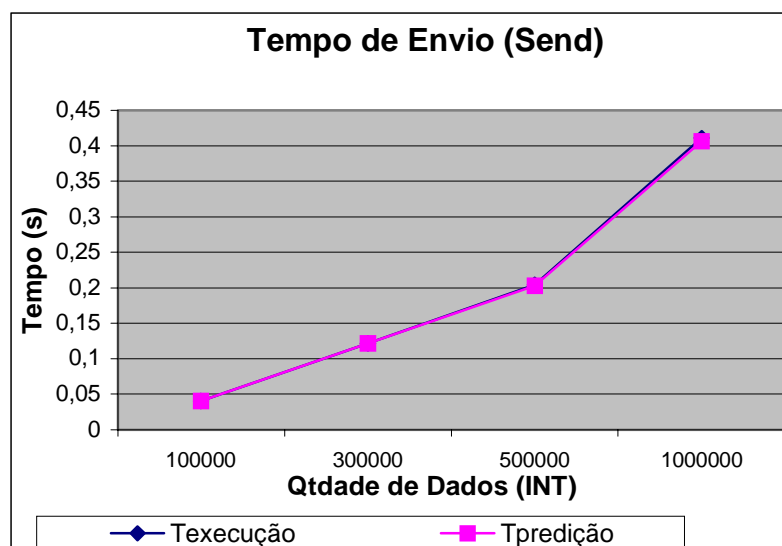


Gráfico 2. Gráfico do resultado da Operação de Comunicação Send.

### 5.3 Resultados com Operação *Broadcast*

Nesta seção, são mostradas as análises dos resultados experimentais, e após esta etapa, são aplicadas as funções da modelagem geral desenvolvidas no capítulo anterior. O resultado obtido é uma função que expressa o comportamento da operação coletiva de comunicação MPI.

Após ter coletado todos os dados experimentais, estes são organizados e analisados e, posteriormente, aplicados nas equações resultantes da modelagem desta operação de comunicação apresentada no capítulo anterior.

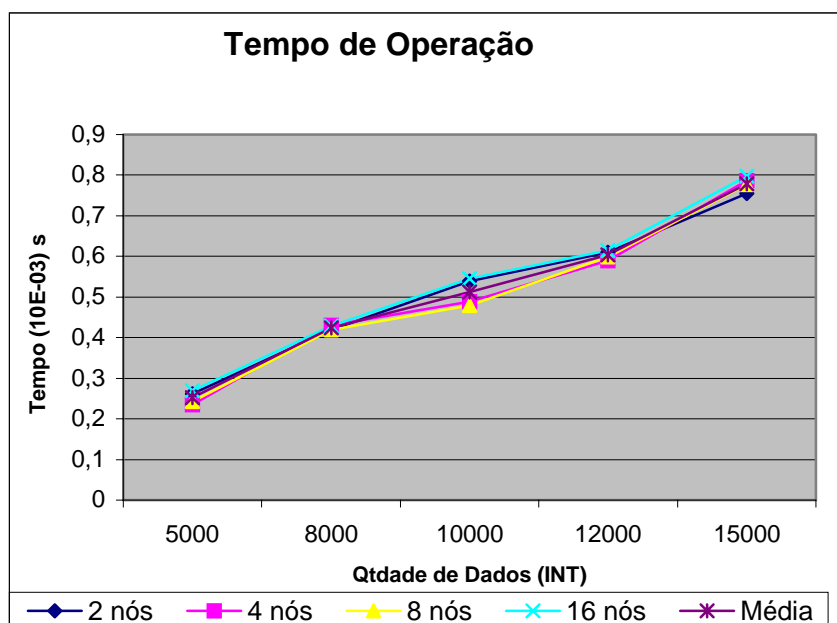


Gráfico 3. Gráfico com tempos de execução da operação *Broadcast*.

As análises utilizando resultados experimentais são divididas em duas etapas. O gráfico 3 traz tempos de execução desta operação de comunicação, com o envio de tamanhos determinados de mensagem.

Os estudos para esta operação estão divididos em dois casos. No primeiro caso, é analisado o comportamento da operação de comunicação quanto à variação do tamanho das mensagens. No segundo caso é pesquisado, fixando o tamanho das



mensagens, qual seria o impacto no tempo de execução da aplicação com a variação no número de nós de processamento.

Analisando-se o gráfico 3, pode-se notar que, variando-se o tamanho da mensagem, as curvas correspondentes aos conjuntos de nós de processamento praticamente se sobrepõem.

Conclui-se, então, que na operação de comunicação Broadcast, o envio não depende do número de nós de processamento.

Utilizando uma das equações geradas pela metodologia do capítulo anterior, na seção 4.5.4, para os casos da análise de desempenho desta operação de comunicação, tem-se:

$$t_{\text{execução}}(z) = t'_x(z) + t'_y(z) + k_4n, z = 1, 2, 3$$

Simplificando a expressão, tem-se:

$$t_{\text{execução}}(z) = k_4n$$

A equação mostra que não há dependência em relação ao número de nós utilizados, o que também foi concluído através da análise experimental.

Substituindo os valores com os dados experimentais obtidos, tem-se :

$$0,0004189 = 8000k_4 \Rightarrow k_4 = \frac{0,0004189}{8000}$$

Assim, a função fica:  $T(n) = \frac{0,0004189}{8000} * n$ , onde n representa a quantidade

de dados.

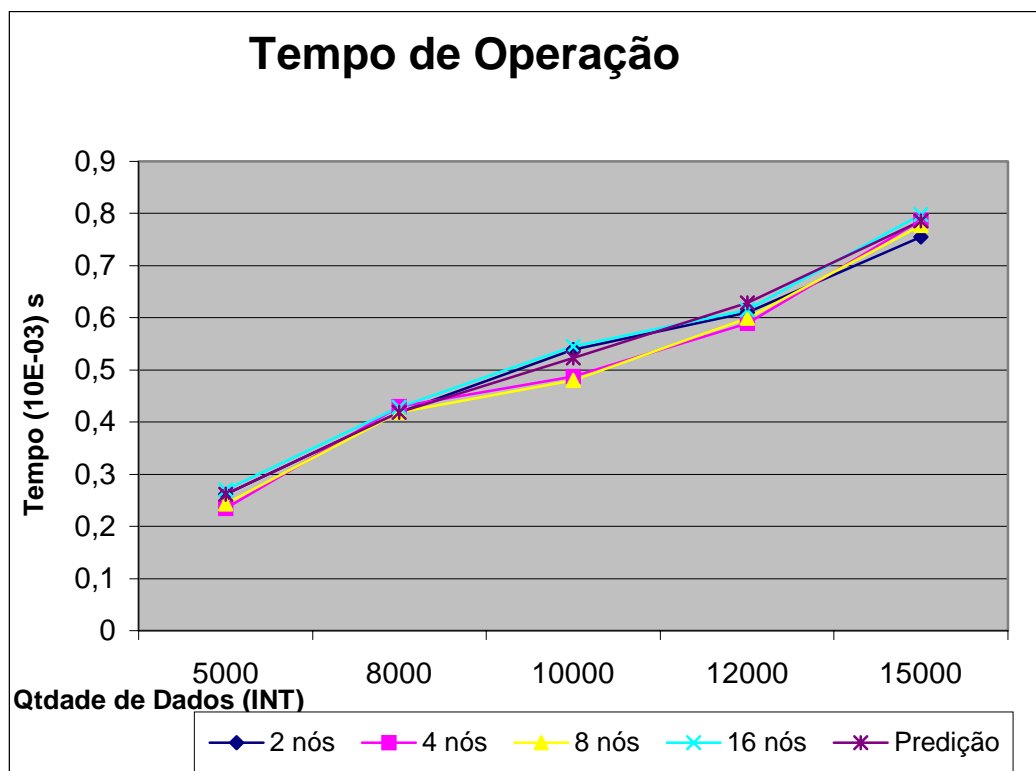


Gráfico 4. Gráfico de tempos da operação Broadcast.

O gráfico 4 apresenta resultados da operação de comunicação MPI\_Bcast. As curvas 2nós, 4nós, 8nós e 16nós correspondem a resultados de execução obtidos experimentalmente, com os respectivos números de nós de processamento. A quinta curva, Predição, é obtida utilizando os resultados da análise analítica desta operação, apresentada com detalhes na seção 4.5.4. A maior diferença entre o tempo de execução real medido, e o tempo de predição, é menor que 1,5%.

A tabela 2 apresenta os tempos de execução experimentais e de predição.

Qtdade Números (INT)	2 nós	4 nós	8 nós	16 nós	Predição
5000	0,261	0,235	0,243	0,270	0,2618
8000	0,419	0,430	0,419	0,428	0,4189
10000	0,539	0,488	0,479	0,545	0,5236
12000	0,610	0,589	0,599	0,615	0,6283
15000	0,755	0,787	0,777	0,798	0,7854

Tabela 2. Resultados Experimentais e de Predição da Operação MPI\_Bcast.

### 5.3.1 Outros Gráficos

Pelo gráfico 5, pode-se observar que, para um dado tamanho de mensagem, o incremento no número de nós de processamento envolvidos em cada conjunto de testes não alterou o tempo de operação deste comando de comunicação.

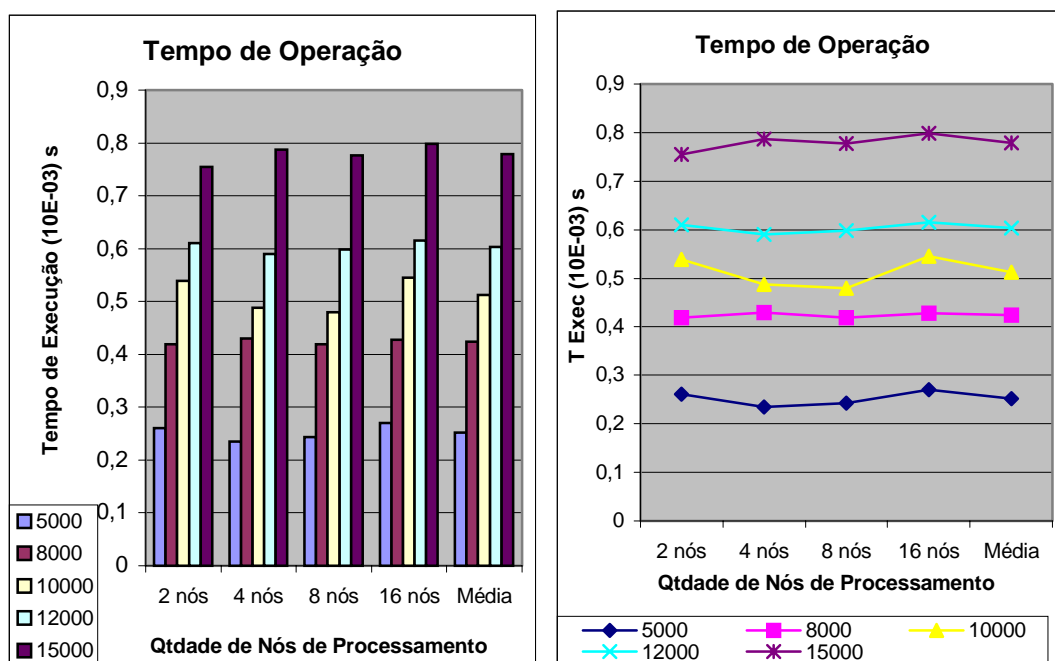


Gráfico5. Gráficos com tempos de operação do comando MPI\_Bcast.

## 5.4 Resultados com Operação de comunicação Scatter

São mostradas, nesta seção, as análises dos resultados obtidos experimentalmente, e em seguida, são aplicadas as funções da modelagem geral desenvolvidas no capítulo anterior. O resultado obtido é uma função que expressa o comportamento da operação coletiva de comunicação MPI.

As equações provenientes de estudos analíticos feitos na seção 4.5.5 do capítulo anterior são:

$$T_{\text{execução}}(z_1) = T'_x(P_0) + T^{[(\#NO-1)*n]/(\#NO)}_e + T^{[(\#NO-1)*n]/(\#NO)}_t + T^{(n)/(\#NO)}_r + T'_y(z), \quad \text{onde } z = P_1, P_2, P_3$$

$$T_{\text{execução}}(z_2) = T'_x(z) + T^{(n)/(\#NO)}_r + T'_y(z).$$

e que

$$T_e = c_1 (\#NO - 1) n / \#NO$$

$$T_t = c_2 + c_3 (\#NO - 1) n / \#NO + c_4 (\#NO - 1)$$

$$T_r = c_5 n / \#NO$$

Considerando que o trecho de código a ser avaliado executa somente a operação de comunicação, basta tomar as parcelas desta função relativas à operação de comunicação. Portanto, a função fica reduzida a

$$T_{\text{execução}}(Z_1) = T^{[(\#NO-1)*n]/(\#NO)}_e + T^{[(\#NO-1)*n]/(\#NO)}_t + T^{(n)/(\#NO)}_r,$$

onde  $z = P1, P2, P3$

$$T_{\text{execução}}(Z_2) = T^{(n)/(\#NO)}_r$$

O gráfico 6 que segue abaixo mostra os tempos de execução desta operação de comunicação, perante diferentes tamanhos de mensagens e número de nós de processamento que foi tratada em cada um dos casos, durante o processo da avaliação e execução dos experimentos.

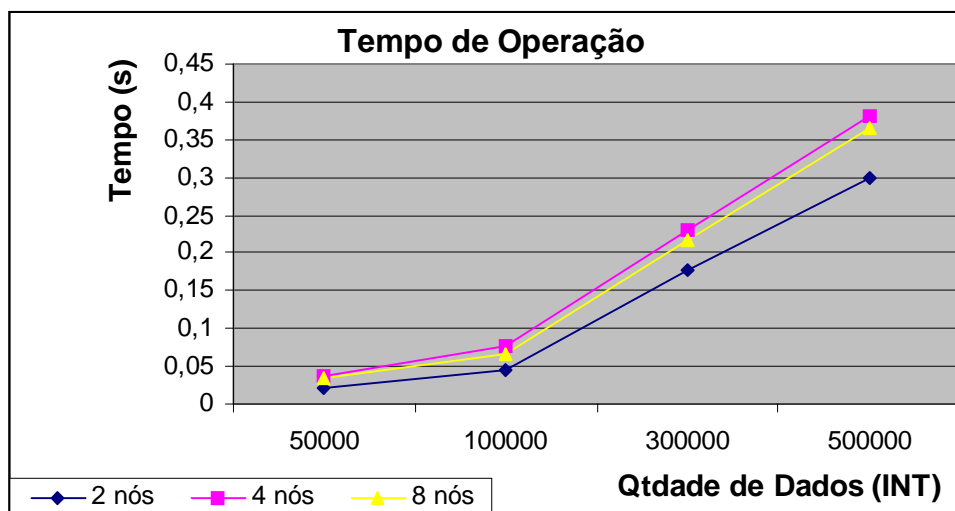


Gráfico 6. Gráfico com tempos de operação do comando `MPI_Scatter`.

Além dos tempos de execução da operação nas condições fornecidas acima, o gráfico 7 apresenta resultados de predição da operação de comunicação MPI\_Scatter. As curvas 2nós, 4nós e 8nós correspondem a resultados de execução obtidas experimentalmente, com os respectivos números de nós de processamento. Uma quarta curva denominada Predição é obtida, utilizando os resultados da análise analítica desta operação de comunicação, apresentada com detalhes na seção 4.5.5 e rerepresentadas no início desta seção. A maior diferença entre o tempo de execução real medido, e o tempo de predição, é menor que 3,9%.

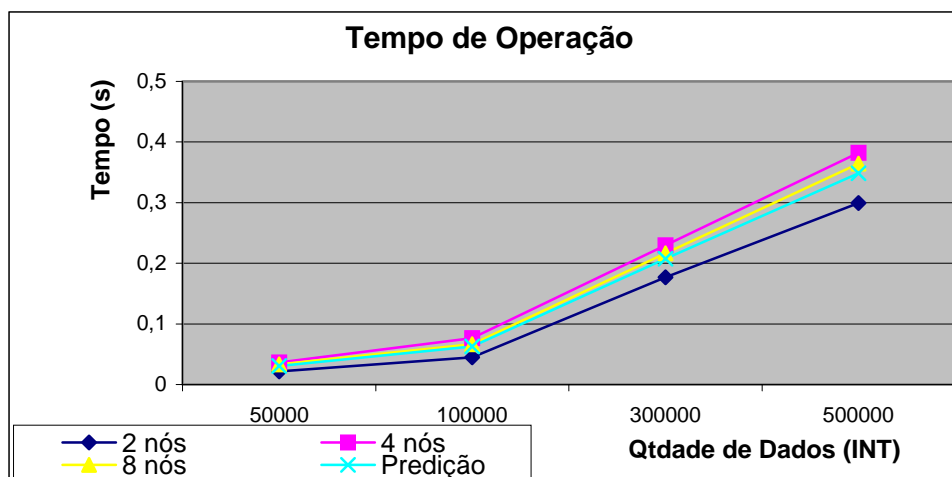


Gráfico 7. Gráfico com tempos de operação e de predição do comando MPI\_Scatter.

## 5.5 Resultados obtidos com Procedimento RANK, do programa IS/NAS

Os estudos e testes experimentais foram feitos no procedimento RANK do programa IS, e foram executados nos programas pertencentes às classes “A”, com  $64^3$  elementos, e classe “B”, com  $102^3$  elementos.

As fórmulas obtidas da modelagem deste trecho de código no capítulo anterior, são aplicadas para as análises de desempenho nesta seção.

É considerado o tamanho do problema TOTAL\_KEYS fixo e variado o número de nós de processamento #NO. Deste modo, tem-se:

$$T_i(\text{\#NO}) = c_1 + c_2 / \text{\#NO} + c_3 * \text{\#NO} + c_4 * \text{\#NO}^2$$

$$T = \max(T_i), \text{ para } T_i = 0 \text{ a } \text{\#NO} - 1$$

Os tempos de execução, obtidos experimentalmente, deste trecho de código são:

$T(2) = 6,853$	$T(4) = 4,1405$	onde $T(n)$ é o tempo de
$T(8) = 2,5375$	$T(16) = 2,2155$	execução do programa com $n$ nós

E substituindo os valores, tem-se que:

Para 2 nós de processamento,  $T(2) = c_1 + c_2 / 2 + c_3 * 2 + c_4 * 4$

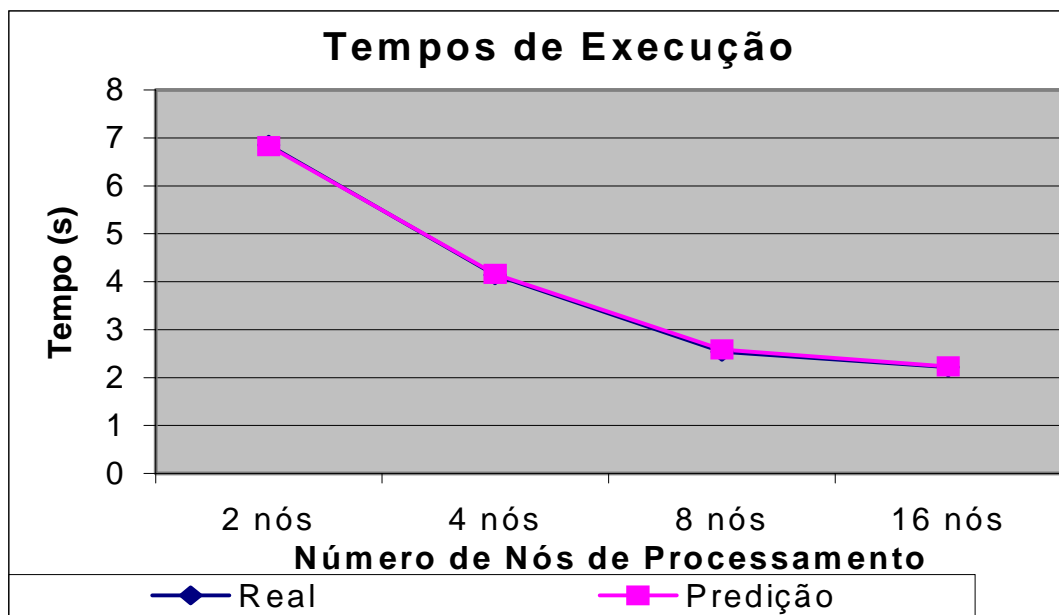
Para 4 nós de processamento,  $T(4) = c_1 + c_2 / 4 + c_3 * 4 + c_4 * 16$

Para 8 nós de processamento,  $T(8) = c_1 + c_2 / 8 + c_3 * 8 + c_4 * 64$

E finalmente, para 16 nós de processamento,  $T(16) = c_1 + c_2 / 16 + c_3 * 16 + c_4 * 256$

Após resolver este sistema de equações lineares, tem-se que a equação para este trecho de código é:

$$T(\text{\#NO}) = 2,5702 + 9,4507 / \text{\#NO} - 0,2444 * \text{\#NO} + 0,0116 * \text{\#NO}^2$$



*Gráfico 8. Gráfico com tempos de operação e de predição do Procedimento RANK/IS/NAS.*

Observando o gráfico 8, foi-se traçada duas curvas. A primeira curva, denominada Real, traz o resultado experimental deste programa submetido ao sistema computacional. A segunda curva, denominada Predição, foi obtida baseada nos resultados da modelagem feita no capítulo anterior, já trazendo consigo as características do trecho de código examinado. Comparando os resultados, a maior diferença entre os dados obtidos foi menos de 4%.

## Capítulo 6

# Conclusões e Trabalhos Futuros

Explicitar um programa paralelo instrumentado com a interface de passagem de mensagens MPI, aplicando representações das classes de grafos *DP\*Graph* e *T-graph\**, introduzidos neste trabalho, é um estágio importante quando se pretende efetuar avaliação de desempenho de programa paralelo, independente da plataforma computacional de memória distribuída utilizada. A representação de programas paralelos utilizando estas classes de grafos estendidos, que é uma representação simples e precisa no seu conteúdo e significado, é relevante para o desenvolvimento de aplicações paralelas, permitindo efetuar pesquisas e análises com grau elevado de detalhes do comportamento de trechos de código de uma aplicação ou uma aplicação completa.

Desta forma, neste trabalho, foram introduzidas duas novas classes de grafos e proposta uma metodologia para efetuar análises e predições de desempenho de programas paralelos em sistemas NOW ( ou rede de estações de trabalho).

A representação de programas paralelos em alto nível utiliza a classe de grafos denominada T-Graph\*, uma extensão do *T-graph (timing graphs)*, representando o programa paralelo no seu nível algorítmico. A representação de



baixo nível, que utiliza classe de grafos *DP\*Graph* (*Distributed Processing Graph*), mostra com elevado grau de detalhe os programas paralelos, quanto a pontos de sincronização e de comunicação no seu fluxo de execução, além de mostrar a presença ou não de elementos seqüenciais. É fácil visualizar os trechos de programas paralelos executados dentro de cada um dos nós de processamento. Enfim, é um instrumento prático para a análise de comportamento de programa paralelo, não somente no aspecto teórico, mas também para práticas experimentais.

Projetos de desenvolvimento de aplicações paralelas podem, em alguns casos, se tornar demasiadamente complexos. Realizando análises com a utilização da metodologia aqui apresentada, pontos de gargalo e de maior custo de processamento podem ser identificados, provendo assim mais informações que auxiliam na compreensão mais exata do comportamento do sistema no aspecto referente ao desempenho.

A metodologia torna possível fazer predições de desempenho de um programa paralelo para um sistema de redes de estações de trabalho com diferentes números de nós de processamento, ou de diferentes tamanhos do problema, com o uso de técnicas analíticas sobre a representação de grafos deste programa. Os resultados obtidos e apresentados no capítulo anterior mostram que a técnica de modelagem analítica proposta apresentou um mínimo de desvio com relação aos tempos obtidos experimentalmente.

## 6.1 Contribuições

De maneira geral, as contribuições deste trabalho estão relacionadas com a definição de classes de grafos para representações de programas paralelos, no nível algorítmico e no nível de execução, e as técnicas analíticas de avaliação de desempenho, que possibilitarão efetuar predições de desempenho destes programas paralelos.

Assim, as principais contribuições deste trabalho são as apresentações das propostas de:

- Estratégia de representação de programas paralelos, utilizando as classes de grafos *DP\*Graph* e *T-graph\**,

- Modelagem e análise de operações de comunicação ponto a ponto e coletivas para programas paralelos que utilizam a interface de passagem de mensagens (MPI),
- Técnica de modelagem analítica e de predição de desempenho para programas paralelos em redes de estações de trabalho, utilizando dados obtidos experimentalmente, através de execuções do programa, instrumentado para fornecer informações sobre tempos de execução,

## 6.2 Trabalhos e Projetos Futuros

É grande a ausência de ferramentas de análise de programas paralelos e, ao mesmo tempo, existe uma demanda enorme por ferramentas de estudo e pesquisa de aplicações que demandam alto desempenho e são descritas com interface de passagem de mensagens e executadas em ambientes de redes de estações de trabalho. Do ponto de vista da relação custo/benefício, atualmente, sistemas NOW são vistos como uma excelente forma de ter acesso à supercomputação.

Este trabalho não abordou uma série de aspectos que mereceriam ser tratados como futuros projetos de estudo e pesquisa.

Com o desenvolvimento desta proposta de metodologia, abre-se uma série de discussões sobre as abordagens feitas neste trabalho e assim, possibilidades de implementar novas linhas de pesquisa e de estudos. Entre elas, podem ser destacadas:

- Efetuar estudos e modelagens de operações de comunicação da interface MPI, para que a análise da biblioteca de funções se torne ainda mais completa e exata,
- Desenvolvimento de uma ferramenta, que ofereça recursos para a aplicação da metodologia proposta e descrita neste trabalho, facilitando a realização de análises e predições,
- Efetuar estudos e avaliações de desempenho em outros tipos de redes de interconexão, como Gigabit Ethernet, SCI, Myrinet e ATM,
- Efetuar o estudo de análise e predição de desempenho feito neste trabalho em redes de estações de trabalho heterogêneas,

- Estudos de análise e predição de desempenho sobre Computação em Malha (*Grid Computing*), que é basicamente uma união de recursos computacionais para a execução de uma aplicação. Detalhes e estudos sobre Computação em Malha, ver [FOST98, FOST01],
- Estudo de algoritmos de ordens diferentes de complexidade. Nos casos pesquisados aqui neste trabalho, foram analisados os casos de programas paralelos cuja complexidade puderam ser descritas utilizando polinômios.

## Referências Bibliográficas

- [ADVE93] ADVE, V.S. **Analyzing the behavior and performance of parallel programs**. Madison, 1993, Tese (doutorado) - Departamento da Ciência de Computação, Universidade de Wisconsin-Madison.
- [ANDE95] ANDERSON, T.E. et al. A case for NOW (network of workstations). **IEEE micro**, v.15, n.1, p.54-64, fevereiro, 1995.
- [BAIL95] BAILEY, D. et al. **The NAS Parallel Benchmarks 2.0**. NASA Ames Research Center, 1995. (Technical Report NAS-95-020)
- [BALA91] BALASUNDARAM, V. et al. Static performance estimator to guide data partitioning decisions. In: Third ACM SIGPLAN Symposium on Principles and Practice of parallel programming, 1991. **Anais**.
- [BODE95] BODEN, N. J. et al. Myrinet - a gigabit-per-second local-area network. **IEEE Micro**, v.15, n.1, p.29-36, 1995.
- [BOYL88] BOYLE, R. et al. **Portable Parallel Programs**. Addison Wesley, 1988.
- [BRUC97] BRUCK, J. et al. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. **The Journal of Parallel and Distributed Computing**, v.40, n.5, p.19-34, 1997.

- [BURN94] BURNS, G.; DAOUD, R.; VAIGL, J. LAM: an open cluster environment for MPI. Artigo em evento. In: Supercomputing Symposium'94, Toronto, Canada, 1994. **Anais**.
- [BURN95] BURNS, G., DAOUD, R. Robust MPI message delivery through guaranteed resources. Artigo em evento. In: MPI developers conference, Universidade de Notre Dame, Notre Dame, Indiana, 1995. **Anais**.
- [CAIN00] CAIN, H.W.; MILLER, B.P.; WYLIE, B.J. A callgraph-based search strategy for automated performance diagnosis. In: Euro-Par 2000, Munich, Alemanha, 2000. **Anais**.
- [CHEN98] CHENG, C.T. **Architectural support for network-based parallel computing**. Los Angeles, 1998. Tese (Doutorado). Departamento de EE-Systems, Universidade de Southern California, CA, EUA.
- [CISC97a] CISCO white paper. Catalyst 2900 XL *Switch* Architecture. Cisco Systems Inc. ([www.cisco.com](http://www.cisco.com)), 1997.
- [CISC00a] CISCO technology brief overview. *Switch* Clustering Overview. Cisco Systems Inc. ([www.cisco.com](http://www.cisco.com)), 2000.
- [CISC00b] CISCO design implementation guide. LAN design guide for the midmarket. Cisco Systems Inc. ([www.cisco.com](http://www.cisco.com)), 2000.
- [CISC00c] CISCO technology brief overview. Cisco midmarket Catalyst LAN *switching* solution. Cisco Systems Inc. ([www.cisco.com](http://www.cisco.com)), 2000.
- [CROV94] CROVELLA, M.E. **Performance Prediction and Tuning of Parallel Programs**. Rochester, 1994. Tese (Doutorado) – Departamento de Ciência da Computação, Universidade de Rochester, EUA.
- [DONG98] DONGARRA, J.; DUFF, I.; SORENSEN, D.C.; VAN DER VORST, H. Numerical linear algebra for high-performance computers. **SIAM**, 1998.
- [DONG00] DONGARRA, J.J. **Performance of various computers using standard linear equations software**. Knoxville, TN, Department of Computer Science, University of Tennessee, 2000.(Technical report CS-89-85, CS, 2000)

- [DURB98] DURBHAKULA, M.; PAI, V.S.; ADVE, S.V. **Improving the accuracy vs. speed tradeoff for simulating shared-memory multiprocessors with ILP processors**. Depto de ciência da computação, Universidade de Rice, 1998. (Technical Report ECE 9802)
- [FAHR93] FAHRINGER, T. **Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers**. Vienna, 1993. Tese (Doutorado) - Tehnischen Universität Wien.
- [FOST95] FOSTER, I.T. **Designing and building parallel programs**, Addison-Wesley, 1995.
- [FOST98] FOSTER, I.T.; KESSELMAN, C. The Globus Project: a Status Report. In: IPSP / SPDP'98 Heterogeneous Computing Workshop, p.4-18, 1998. **Anais**.
- [FOST01] FOSTER, I.T.; KESSELMAN, C.; TUECKE, S. The Anatomy of the Grid – Enabling Scalable Virtual Organizations. **International Journal of Supercomputer Applications**, 2001.
- [GAUT00] GAUTAMA, H.; GEMUND, A.J.C. van Static Performance Prediction of Data Dependent Programs. Artigo em evento. In: Second International ACM Workshop on Software and Performance (WOSP'00), Ottawa, Canada, p.216-226, 2000. **Anais**.
- [GEMU93] GEMUND, A.J.C. van Performance prediction of parallel processing systems: The PAMELA methodology. In: ACM International Conference on Supercomputing (ICS'93), 7<sup>th</sup>, Tokyo, p. 318-327, 1993. **Anais**.
- [GEMU95] GEMUND A.J.C. van Compile-time performance prediction of parallel systems. In: Computer Performance Evaluation: Modeling Techniques and Tools (Tools'95), Heidelberg, p.299-313, 1995. **LNCS 977**.
- [GEMU96] GEMUND A.J.C. van **Performance modeling of parallel systems.**, Delft, 1996. Tese (Doutorado) - Delft University of Technology, Delft University Press, ISBN 90-407-1326-X.
- [GROP96] GROPP, W. et al **A high-performance, portable implementation of the MPI message passing interface standard**. Chicago, 1996. Argonne National

Laboratory, University of Chicago, 1996.  
(<http://www.mcs.anl.gov/mpi/mpich>).

- [GROP98] GROPP, W. et al **MPI The complete reference** - The MPI extensions, v.2. MIT Press, 1998.
- [GUBI95A] GUBITOSO, M.D.; CORNSEN, J. **Performance considerations in a virtual shared memory system**. Berlin, Alemanha, 1995. Plena Project, 1995. (Technical Report).
- [GUBI95B] GUBITOSO, M.D.; CORNSEN, J. **Performance considerations in VOTE for PEACE**. São Paulo, 1995. IME-USP. (Relatório Técnico)
- [GUBI96] GUBITOSO, M.D. **Modelos analíticos de desempenho para sistemas de memória compartilhada virtual**. São Paulo, 1996. Tese (Doutorado) - Departamento de ciência da computação, Instituto de Matemática e Estatística – Universidade de São Paulo.
- [GUNT98] GUNTHER, N.J. **The practical performance analyst - performance-by-design techniques for distributed systems**. McGraw-Hill Series on Computer Communications, 1998.
- [HARR93] HARRISON, P.G.; PATEL, N.M. **Performance modeling of communication networks and computer architectures**. Addison-Wesley, 1993.
- [HITC82] HITCHCOCK, R.B. Timing verification and the timing analysis program. In: Design Automation Conference, 19<sup>th</sup>, p.594-604, 1982. **Anais**.
- [HOCK91] HOCKNEY, R. Performance parameters and benchmarking of supercomputers. **Parallel Computing**, v.17, p. 11-30, 1991.
- [JAIN91] JAIN, J. **The art of computer systems performance analysis**. Wiley Professional Computing, Wiley, 1991.
- [KANT92] KANT, K. **Introduction to computer system performance evaluation**. Mc Graw-Hill, 1992.

- [KAPL95] KAPLAN, D.J.; STEVENS, R.S. **Processing graph method 2.0 semantics**. US Naval Research Laboratory, 1995. (Relatório Interno). [www.ait.nrl.navy.mil/pgmt/](http://www.ait.nrl.navy.mil/pgmt/)
- [KAPL97] KAPLAN, D.J. An introduction to the Processing Graph Method. Artigo em evento. In: IEEE International Conference on Engineering of Computer Based Systems, California, EUA, 1997. **Anais**.
- [KARA99] KARAVANIC, K.L. **Experiment management support for parallel performance tuning**. Madison, EUA, 1999. Tese (Doutorado) - Departamento de ciência da computação, Universidade em Wisconsin-Madison, EUA.
- [KARY98] KARYPIS, G.; KUMAR, V. **Analysis of multilevel graph partitioning**. Minneapolis, EUA, 1998. Universidade de Minnesota. (Technical report 95-037)
- [KLEI93] KLEINROCK, L. On the modeling and analysis of computer networks. **IEEE**, v.81, n.8, p. 1179-1191, 1993.
- [KLIE88] KLIEGERMAN, E.; STOYENKO, A Real-Time Euclid: a language for reliable real-time systems. **IEEE Transactions on Software Engineering**, SE-1(9): 941-949, 1988.
- [LAND98] LANDRUM, J.; HARDWICK, J.; STOUT, Q.F. Predicting algorithm performance. **Computing Science and Statistics** 30, p. 309-314, 1998.
- [LAUR97] LAURIA, M.; CHIEN, A. MPI-FM: high performance MPI on workstation clusters. **The Journal of Parallel and Distributed Computing**, v.40, n.5, p.431-452, 1997.
- [LIKU01] LI, K.C.; SATO, L.M.; GAUDIOT, J.-L. A Tool for Performance Analysis and Prediction of Parallel Computing on NOW. In: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), Las Vegas, USA, 2001. **Anais**.
- [LIKU99a] LI K.C.; SATO, L.M. Applying Parasys for parallel algorithm analysis. Artigo em evento. In: International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), vol. 3, Las Vegas, EUA, 1999. **Anais**.



- [LIKU99b] LI K.C.; SATO, L.M. Using Parasys for parallel algorithm performance analysis. Artigo em evento. In: International Congress on Informatic Engineering (ICIE'99), Buenos Aires, Argentina, 1999. **Anais**.
- [LIYS95] LI, Y.S., MALIK, S. Performance Analysis on embedded software using implicit path enumeration. In: ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time systems, p.95-105, La Jolla, EUA, 1995. **Anais**.
- [LOVE93] LOVEMAN, D.B. High Performance Fortran. **IEEE Parallel and Distributed Technology**, v.1, n.1, p.25-42, 1993.
- [MEND93] MENDES, C. L. Performance prediction by trace transformation. In: V SBAC-PAD Simpósio Brasileiro de arquiteturas de computadores - Processamento de alto desempenho, 1993. **Anais**.
- [MOKA89] MOK, A. K., AMERASINGHE, P., CHEN, M., TANTISIRIVAT, K. Evaluating tight execution time bounds of programs by annotations. In: 6<sup>th</sup> IEEE Workshop on Real-Time Operating Systems and Software, pp. 74-80, EUA, 1989. **Anais**.
- [MOOR01] MOORE, S., CRONK, D., LONDON, K., DONGARRA, J. Review of performance analysis tools for MPI parallel programs. In: Euro PVM/MPI 2001 meeting. **Anais**.
- [MPBE01] MPI Benchmark Home Page. <http://icl.cs.utk.edu/projects/llcbench>. Acessado em julho de 2001.
- [MPI196] **MPI Primer / Developing with LAM**. Ohio Supercomputer Center, The Ohio State University, <http://www.mpi.nd.edu>, <http://www.osc.edu>, 1996. (Manual)
- [MPI298] **MPI Tutorial**, Laboratory of Scientific Computing, Universidade de Notre Dame, Notre Dame, EUA, [http://www.mpi.nd.edu/mpi\\_tutorials/fall\\_1998/](http://www.mpi.nd.edu/mpi_tutorials/fall_1998/), 1998. (Manual)
- [MPI301] **MPI Tutorial On-Line**, Ohio Supercomputing Center, <http://oscinfo.osc.edu/training>, 2001. (Manual)

- [NAHU96] NAHUM, E.M. **Validating an architectural simulator**. Amherst, 1996. Dept of CS, University of Massachusetts at Amherst, EUA. (Technical Report 96-40)
- [NETP01] Network Performance Benchmark. <http://www.netperf.org>. Acessado em julho de 2001.
- [NEVI96] NEVIN, N. **The Performance of LAM 6.0 and MPICH 1.0.12 on a workstation cluster**. Columbus, 1996. Ohio Supercomputing Center, Columbus, Ohio, EUA. (Technical Report OSC-TR-1996-4)
- [NICO96] NICOL, D., HEIDELBERGER, P., Parallel Execution for serial simulators. **ACM Transactions on Modeling and Computer Simulation**, v.6, n.3, p.210-242, 1996.
- [PAIV97A] PAI, V.S., RANGANATHAN, P., ADVE, S.V. RSIM: an execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In: IEEE TCCA Newsletter, outubro, 1997. **Anais**.
- [PAIV97B] PAI, V.S., RANGANATHAN, P., ADVE, S.V. The impact of Instruction-Level Parallelism on multiprocessor performance and simulation methodology. In: IEEE HPCA-3, p. 72-83, 1997. **Anais**.
- [PANC96] PANCAKE, C.M. Is parallelism for you? In: **Computational Science and Engineering**, v.3, n.2, p.18-37, 1996.
- [PARK91] PARK, C.Y., SHAW, A.C. Experiments with a program-timing tool based on source-level timing schema. **IEEE Computer**, 24(5):48-57, 1991.
- [PARK92] PARK, C.Y. **Predicting deterministic execution times of real-time programs**. Seattle, 1992. Tese (Doutorado) - Universidade de Washington, Seattle, WA, EUA.
- [PUSC89] PUSCHNER, P., KOZA, C. Calculating the maximum execution time of real-time programs. **The Journal of Real-Time systems**, v.1, n.2, p.160-176, 1989.

- [PUSC97] PUSCHNER, P.; SCHEDL, A. Computing maximum task execution times - a graph-based approach. **Journal of Real-Time Systems**, v.13, n.1, p.67-91, 1997.
- [REED87] REED, D.A.; GRUNWALD, D.C. The performance of multicomputer interconnection networks. **Computer**, v.20, n.6, p. 63-73, 1987.
- [REED93] REED, D.A. Performance instrumentation techniques for parallel systems. **Performance Evaluation**, p. 463-490, 1993.
- [SATO92] SATO, L.; MIDORIKAWA, E.; BERNAL, V. Práticas em programação paralela. In: SBAC-PAD Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho, v.1, p.1-38, São Paulo, 1992. **Anais**.
- [SCOT95] SCOTT, P.; LAURIA, M.; CHIEN, A. High performance messaging on workstations Illinois Fast Message (FM) for Myrinet. In: Supercomputing'95, San Diego, CA, 1995. ([www.supercomp.org/conferen/sc95/proceedings](http://www.supercomp.org/conferen/sc95/proceedings)). **Anais**.
- [SHAW89] SHAW, A.C. Reasoning about time in high-level language software. Artigo em revista. **IEEE Transactions on Software Engineering**, v.15, n.7, p.875-889, 1989.
- [SNIR96] SNIR, M. et al. **MPI: the complete reference - the MPI core, vol. 1**. The MIT Press, 1998.
- [SORI00] SORIN, D.J. et al. **Analytic evaluation of shared-memory architectures for heterogeneous applications**. Madison, 2000. Departamento da ciência de computação, Universidade de Wisconsin-Madison. (Technical report #1404b)
- [STRO97] STROHMAIER, E. **Statistical performance modeling: case study of the NPB 2.1 results**. Knoxville, 1997. Departamento da ciência de computação, Universidade de Tennessee. (Technical Report UTK-CS-97-354)
- [STRO97] STROHMAIER, E. et al. High-Performance Computing in Industry. **Supercomputer**, 1997.
- [TABE99] TABE, T.B.; STOUT, Q.F. **The use of MPI communication library in the NAS parallel benchmarks**. Michigan, 1999. Departamento da ciência de

computação e de engenharia, Universidade de Michigan. (Technical report CSE-TR-386-99)

- [THEB99] The Beowulf Underground. EUA. Disponível em (<http://www.beowulf-underground.org/documentation.html>), 1999.
- [VRSA88] VRSALOVIC, D.F. et al. Performance Prediction and Calibration for a Class of Multiprocessors. **IEEE Transactions on Computers**, v.37, n.11, p.1353-1364, 1988.
- [WOOS95] WOO, S.C. et al. The SPLASH-2 programs: characterization and methodological considerations. In: 22<sup>nd</sup> Annual International Symposium on Computer Architecture, p. 24-36, 1995. **Anais**.
- [XIAO95] XIAOHAN, Q., BAER, J-L., A performance evaluation of cluster architectures. In: SIGMETRICS, 1995. **Anais**.
- [ZHAN95] ZHANG, X., YAN, Y. A framework of performance prediction of parallel computing on nondedicated heterogeneous NOW. In: 1995 International Conference on Parallel Processing, p.163-167, vol. I, 1995. **Anais**.

# Apêndice I

## Programas de Benchmark

### 1. Programa de Benchmark NAS

Os programas de benchmark NPB (*NASA Parallel Benchmarks*) consistem de 8 problemas, composta por 5 *kernels* e 3 aplicações de dinâmica de fluidos. O objetivo deste conjunto de programas é avaliar o desempenho de computadores paralelos em problemas de dinâmica de fluidos computacional (CFD), e desenvolvidos pelo Programa NAS (*Numerical Aerodynamics Simulation*) do Centro de Pesquisas Ames da NASA.

As três aplicações deste conjunto de programas é apresentado na primeira tabela abaixo, enquanto os cinco *kernels* (núcleos) são apresentados na segunda tabela.

Sigla do Algoritmo	Descrição do Algoritmo
<b>BT</b>	Este resolve múltiplos sistemas independentes, de equações blocos diagonais, não diagonalmente dominantes.
<b>LU</b>	Este resolve um sistema triangular inferior e superior bloco 5x5 esparsos regulares, com a aplicação de SSOR.
<b>SP</b>	Este resolve múltiplos sistemas independentes de equações escalares pentadiagonais, não diagonalmente dominantes.

Sigla do Algoritmo	Descrição do Algoritmo
<b>CG</b>	Este calcula uma aproximação para o menor valor do autovalor de uma matriz grande, esparsa e simétrica, além de exigir comunicações irregulares de longa distância.
<b>EP</b>	Este gera pares de desvios randômicos com um esquema adequado para computação paralela e tabula os números em pares sucessivamente. Estes desvios randômicos são estatísticas bidimensionais, que são resultados do acúmulo de uma grande quantidade de números pseudo-aleatórios gaussianos.
<b>FT</b>	Este representa um núcleo computacional do método espectral baseado em FFT de três dimensões. Este teste é bastante rigoroso para a avaliação do desempenho de comunicações de longa distância.
<b>IS</b>	Este representa computações de códigos utilizando método de partícula. O objetivo é obter desempenho de computação com inteiros e de comunicação de dados.
<b>MG</b>	Este resolve equação diferencial parcial de Poisson 3-D. Exige ainda comunicação de longa distância, testando ainda comunicações de dados de curta e média distância.

## 2. Programa de Benchmark MPBench

O conjunto de programas de benchmark MPBench tem como objetivo avaliar desempenho de principais funções da Interface de Passagem por mensagens MPI em MPPs e redes de estações de trabalho (NOW).

Os dados gerados e obtidos a partir da execução destes programas são característicos de um determinado sistema computacional. Os programas de benchmark deste conjunto tem como objetivo:

Programa de Benchmark	Número de Processos	Unidade dos Resultados
Bandwidth	2	Megabytes/s
Roundtrip	2	Transações/s
Application Latency	2	Micro Segundos
Broadcast	A ser definido	Megabytes/s
Reduce	A ser definido	Megabytes/s
AllReduce	A ser definido	Megabytes/s
Bidirectional Bandwidth	2	Megabytes/s
All-to-All	A ser definido	Megabytes/s

### Referência

[MUCC98] MUCCI, P. J., LONDON, K., THURMAN, J. The MPBench Report. Knoxville, 1998. Relatório Técnico. Departamento da Ciência de Computação, Universidade de Tennessee, EUA.

## Apêndice II

### Programa IS / NAS

Segue a partir da próxima página o código-fonte do programa IS, do conjunto de programas de benchmark NAS, utilizado como objeto de estudo e análise neste trabalho de pesquisa.



```

/*****
*****
*
*   N A S   P A R A L L E L   B E N C H M A R K S   2.3   *
*
*   I S
*
*****
*****
*
*   This benchmark is part of the NAS Parallel Benchmark 2.3 suite. *
*   It is described in NAS Technical Report 95-020.
*
*   Permission to use, copy, distribute and modify this software *
*   for any purpose with or without fee is hereby granted. We *
*   request, however, that all derived work reference the NAS *
*   Parallel Benchmarks 2.3. This software is provided "as is" *
*   without express or implied warranty.
*
*   Information on NPB 2.3, including the technical report, the *
*   original specifications, source code, results and information *
*   on how to submit new results, is available at:
*
*   http: www.nas.nasa.gov NAS NPB
*
*   Send comments or suggestions to npb@nas.nasa.gov
*   Send bug reports to npb-bugs@nas.nasa.gov
*
*   NAS Parallel Benchmarks Group

```

```

*   NASA Ames Research Center
*   Mail Stop: T27A-1
*   Moffett Field, CA 94035-1000
*
*   E-mail: npb@nas.nasa.gov
*   Fax: (415) 604-3957
*
*****
*****
*
*   Author: M. Yarrow
*
*****
*****/

#include "mpi.h"
#include "npbparams.h"
#include <stdlib.h>
#include <stdio.h>

/*****/
/* default values */
/*****/

#ifndef CLASS
#define CLASS 'S'
#define NUM_PROCS 1
#endif

```

```

/*****
/* CLASS S */
/*****
#if CLASS == 'S'
#define TOTAL_KEYS_LOG_2 16
#define MAX_KEY_LOG_2 11
#define NUM_BUCKETS_LOG_2 9
#endif

/*****
/* CLASS W */
/*****
#if CLASS == 'W'
#define TOTAL_KEYS_LOG_2 20
#define MAX_KEY_LOG_2 16
#define NUM_BUCKETS_LOG_2 10
#endif

/*****
/* CLASS A */
/*****
#if CLASS == 'A'
#define TOTAL_KEYS_LOG_2 23
#define MAX_KEY_LOG_2 19
#define NUM_BUCKETS_LOG_2 10
#endif

```

```

/*****
/* CLASS B */
/*****
#if CLASS == 'B'
#define TOTAL_KEYS_LOG_2 25
#define MAX_KEY_LOG_2 21
#define NUM_BUCKETS_LOG_2 10
#endif

/*****
/* CLASS C */
/*****
#if CLASS == 'C'
#define TOTAL_KEYS_LOG_2 27
#define MAX_KEY_LOG_2 23
#define NUM_BUCKETS_LOG_2 10
#endif

#define TOTAL_KEYS (1 << TOTAL_KEYS_LOG_2)
#define MAX_KEY (1 << MAX_KEY_LOG_2)
#define NUM_BUCKETS (1 << NUM_BUCKETS_LOG_2)
#define NUM_KEYS (TOTAL_KEYS/NUM_PROCS)

/*****
*****/

/* On larger number of processors, since the keys are (roughly) */
/* gaussian distributed, the first and last processor sort keys */
/* in a large interval, requiring array sizes to be larger. Note */

```

```

/* that for large NUM_PROCS, NUM_KEYS is, however, a small
number*/
/*****
*****/
#if NUM_PROCS < 256
#define SIZE_OF_BUFFERS 3*NUM_KEYS/2
#else
#define SIZE_OF_BUFFERS 3*NUM_KEYS
#endif

#define MAX_PROCS      256
#define MAX_ITERATIONS 10
#define TEST_ARRAY_SIZE 5

/*****
*/
/* Enable separate communication, */
/* computation timing and printout */
/*****
*/
/* #define TIMING_ENABLED */

/*****
*/
/* Typedef: if necessary, change the */
/* size of int here by changing the */
/* int type to, say, long */
/*****
*/
typedef int INT_TYPE;

```

```

/*****
*/
/* MPI properties: */
/*****
*/
int  my_rank,
     comm_size;

/*****
*/
/* Some global info */
/*****
*/
INT_TYPE *key_buff_ptr_global, /* used by full_verify to get */
          total_local_keys,   /* copies of rank info */
          total_lesser_keys;

int  passed_verification;

/*****
*/
/* These are the three main arrays. */
/* See SIZE_OF_BUFFERS def above */
/*****
*/
INT_TYPE key_array[SIZE_OF_BUFFERS],
          key_buff1[SIZE_OF_BUFFERS],
          key_buff2[SIZE_OF_BUFFERS],
          bucket_size[NUM_BUCKETS+TEST_ARRAY_SIZE], /* Top 5
elements for */

```

```

    bucket_size_totals[NUM_BUCKETS+TEST_ARRAY_SIZE], /*
part. ver. vals */
    bucket_ptrs[NUM_BUCKETS],

process_bucket_distrib_ptr1[NUM_BUCKETS+TEST_ARRAY_SIZE],

process_bucket_distrib_ptr2[NUM_BUCKETS+TEST_ARRAY_SIZE],
    send_count[MAX_PROCS], recv_count[MAX_PROCS],
    send_displ[MAX_PROCS], recv_displ[MAX_PROCS];

/*****
/* Partial verif info */
*****/
INT_TYPE test_index_array[TEST_ARRAY_SIZE],
    test_rank_array[TEST_ARRAY_SIZE],

    S_test_index_array[TEST_ARRAY_SIZE] =
        {48427,17148,23627,62548,4431},
    S_test_rank_array[TEST_ARRAY_SIZE] =
        {0,18,346,64917,65463},

    W_test_index_array[TEST_ARRAY_SIZE] =
        {357773,934767,875723,898999,404505},
    W_test_rank_array[TEST_ARRAY_SIZE] =
        {1249,11698,1039987,1043896,1048018},

    A_test_index_array[TEST_ARRAY_SIZE] =
        {2112377,662041,5336171,3642833,4250760},
    A_test_rank_array[TEST_ARRAY_SIZE] =

```

```

    {104,17523,123928,8288932,8388264},

    B_test_index_array[TEST_ARRAY_SIZE] =
        {41869,812306,5102857,18232239,26860214},
    B_test_rank_array[TEST_ARRAY_SIZE] =
        {33422937,10244,59149,33135281,99},

    C_test_index_array[TEST_ARRAY_SIZE] =
{44172927,72999161,74326391,129606274,21736814},
    C_test_rank_array[TEST_ARRAY_SIZE] =
        {61147,882988,266290,133997595,133525895};

/*****
/* function prototypes */
*****/
double randlc( double *X, double *A );

void full_verify( void );

void c_print_results( char *name,
                    char class,
                    int n1,
                    int n2,
                    int n3,
                    int niter,
                    int nprocs_compiled,
                    int nprocs_total,

```

```

double t,
double mops,
    char *optype,
int passed_verification,
char *npbversion,
char *compiletime,
char *mpicc,
char *clink,
char *cmpi_lib,
char *cmpi_inc,
char *cflags,
char *clinkflags );

void timer_clear( int n );
void timer_start( int n );
void timer_stop( int n );
double timer_read( int n );

/*
 * FUNCTION RANDLC (X, A)
 *
 * This routine returns a uniform pseudorandom double precision
number in the
 * range (0, 1) by using the linear congruential generator
 *
 *  $x_{k+1} = a x_k \pmod{2^{46}}$ 
 *

```

```

 * where  $0 < x_k < 2^{46}$  and  $0 < a < 2^{46}$ . This scheme generates  $2^{44}$ 
numbers
 * before repeating. The argument A is the same as 'a' in the above
formula,
 * and X is the same as  $x_0$ . A and X must be odd double precision
integers
 * in the range (1,  $2^{46}$ ). The returned value RANDLC is normalized to
be
 * between 0 and 1, i.e.  $RANDLC = 2^{(-46)} * x_1$ . X is updated to
contain
 * the new seed  $x_1$ , so that subsequent calls to RANDLC using the
same
 * arguments will generate a continuous sequence.
 *
 * This routine should produce the same results on any computer with at
least
 * 48 mantissa bits in double precision floating point data. On Cray
systems,
 * double precision should be disabled.
 *
 * David H. Bailey   October 26, 1990
 *
 * IMPLICIT DOUBLE PRECISION (A-H, O-Z)
 * SAVE KS, R23, R46, T23, T46
 * DATA KS/0/
 *
 * If this is the first call to RANDLC, compute  $R23 = 2^{-23}$ ,  $R46 = 2^{-46}$ ,
 *  $T23 = 2^{23}$ , and  $T46 = 2^{46}$ . These are computed in loops, rather
than

```

\* by merely using the \*\* operator, in order to insure that the results are  
 \* exact on all systems. This code assumes that 0.5D0 is represented  
 exactly.

\*/

```

/*****
*****/
/*****      R A N D L C      *****/
/*****      *****/
/***** portable random number generator *****/
/*****
*****/

```

```

double randlc(X, A)
double *X;
double *A;
{
    static int    KS=0;
    static double R23, R46, T23, T46;
    double       T1, T2, T3, T4;
    double       A1;
    double       A2;
    double       X1;
    double       X2;
    double       Z;
    int          i, j;

    if (KS == 0)
    {

```

```

R23 = 1.0;
R46 = 1.0;
T23 = 1.0;
T46 = 1.0;

```

```

for (i=1; i<=23; i++)
{
    R23 = 0.50 * R23;
    T23 = 2.0 * T23;
}
for (i=1; i<=46; i++)
{
    R46 = 0.50 * R46;
    T46 = 2.0 * T46;
}
KS = 1;
}

```

```

/* Break A into two parts such that  $A = 2^{23} * A1 + A2$  and set  $X = N$ .
*/

```

```

T1 = R23 * *A;
j = T1;
A1 = j;
A2 = *A - T23 * A1;

```

```

/* Break X into two parts such that  $X = 2^{23} * X1 + X2$ , compute
 $Z = A1 * X2 + A2 * X1 \pmod{2^{23}}$ , and then
 $X = 2^{23} * Z + A2 * X2 \pmod{2^{46}}$ .
*/

```

```

T1 = R23 * *X;
j = T1;
X1 = j;
X2 = *X - T23 * X1;
T1 = A1 * X2 + A2 * X1;

j = R23 * T1;
T2 = j;
Z = T1 - T23 * T2;
T3 = T23 * Z + A2 * X2;
j = R46 * T3;
T4 = j;
*X = T3 - T46 * T4;
return(R46 * *X);
}

```

```

/*****
*****/
/***** F I N D _ M Y _ S E E D *****/
/***** *****/
/***** returns parallel random number seq seed *****/
/***** *****/
*****/

```

```

/*
* Create a random number sequence of total length nn residing
* on np number of processors. Each processor will therefore have a
* subsequence of length nn/np. This routine returns that random

```

```

* number which is the first random number for the subsequence
* belonging
* to processor rank kn, and which is used as seed for proc kn ran # gen.
*/

```

```

double find_my_seed( long kn, /* my processor rank, 0<=kn<=num
procs */
                    long np, /* np = num procs */
                    long nn, /* total num of ran numbers, all procs */
                    double s, /* Ran num seed, for ex.: 314159265.00 */
                    double a ) /* Ran num gen mult, try 1220703125.00 */
{
    long i;

    double t1,t2,t3,an;
    long mq,nq,kk,ik;

    nq = nn / np;

    for( mq=0; nq>1; mq++,nq/=2 )
        ;

    t1 = a;

    for( i=1; i<=mq; i++ )
        t2 = randlc( &t1, &t1 );

```

```

an = t1;

kk = kn;
t1 = s;
t2 = an;

for( i=1; i<=100; i++ )
{
    ik = kk / 2;
    if( 2 * ik != kk )
        t3 = randlc( &t1, &t2 );
    if( ik == 0 )
        break;
    t3 = randlc( &t2, &t2 );
    kk = ik;
}

return( t1 );
}

/*****
*****/
/***** CREATE_SEQ *****/
/*****
*****/

```

```

void create_seq( double seed, double a )
{
    double x;
    int i, j, k;

    k = MAX_KEY/4;

    for( i=0; i<NUM_KEYS; i++)
    {
        x = randlc(&seed, &a);
        x += randlc(&seed, &a);
        x += randlc(&seed, &a);
        x += randlc(&seed, &a);

        key_array[i] = k*x;
    }
}

/*****
*****/
/***** FULL_VERIFY *****/
/*****
*****/

void full_verify()
{

```



```

MPI_Status status;
MPI_Request request;

INT_TYPE i, j;
INT_TYPE k;
INT_TYPE m, unique_keys;

/* Now, finally, sort the keys: */
for( i=0; i<total_local_keys; i++ )
    key_array[--key_buff_ptr_global[key_buff2[i]]-
              total_lesser_keys] = key_buff2[i];

/* Send largest key value to next processor */
if( my_rank > 0 )
    MPI_Irecv( &k,
              1,
              MPI_INT,
              my_rank-1,
              1000,
              MPI_COMM_WORLD,
              &request );
if( my_rank < comm_size-1 )
    MPI_Send( &key_array[total_local_keys-1],
             1,
             MPI_INT,
             my_rank+1,
             1000,
             MPI_COMM_WORLD );

```

```

if( my_rank > 0 )
    MPI_Wait( &request, &status );

/* Confirm that neighbor's greatest key value
is not greater than my least key value */
j = 0;
if( my_rank > 0 )
    if( k > key_array[0] )
        j++;

/* Confirm keys correctly sorted: count incorrectly sorted keys, if any */
for( i=1; i<total_local_keys; i++ )
    if( key_array[i-1] > key_array[i] )
        j++;

if( j != 0 )
{
    printf( "Processor %d: Full_verify: number of keys out of sort:
%d\n",
           my_rank, j );
}
else
    passed_verification++;
}

```

```

/*****
*****/
/*****          R A N K          *****/
/*****
*****/

void rank( int iteration )
{

    INT_TYPE  i, j, k;
    INT_TYPE  l, m;

    INT_TYPE          shift      =      MAX_KEY_LOG_2      -
    NUM_BUCKETS_LOG_2;
    INT_TYPE  key;
    INT_TYPE  bucket_sum_accumulator;
    INT_TYPE  local_bucket_sum_accumulator;
    INT_TYPE  min_key_val, max_key_val;
    INT_TYPE  *key_buff_ptr;

/* Iteration alteration of keys */
if(my_rank == 0 )
{
    key_array[iteration] = iteration;

```

```

    key_array[iteration+MAX_ITERATIONS] = MAX_KEY - iteration;
}

/* Initialize */
for( i=0; i<NUM_BUCKETS+TEST_ARRAY_SIZE; i++ )
{
    bucket_size[i] = 0;
    bucket_size_totals[i] = 0;
    process_bucket_distrib_ptr1[i] = 0;
    process_bucket_distrib_ptr2[i] = 0;
}

/* Determine where the partial verify test keys are, load into */
/* top of array bucket_size */
for( i=0; i<TEST_ARRAY_SIZE; i++ )
    if( (test_index_array[i]/NUM_KEYS) == my_rank )
        bucket_size[NUM_BUCKETS+i] =
            key_array[test_index_array[i] % NUM_KEYS];

/* Determine the number of keys in each bucket */
for( i=0; i<NUM_KEYS; i++ )
    bucket_size[key_array[i] >> shift]++;

/* Accumulative bucket sizes are the bucket pointers */
bucket_ptrs[0] = 0;
for( i=1; i< NUM_BUCKETS; i++ )

```

```

    bucket_ptrs[i] = bucket_ptrs[i-1] + bucket_size[i-1];

/* Sort into appropriate bucket */
for( i=0; i<NUM_KEYS; i++ )
{
    key = key_array[i];
    key_buff1[bucket_ptrs[key >> shift]++] = key;
}

#ifdef TIMING_ENABLED
    timer_stop( 2 );
    timer_start( 3 );
#endif

/* Get the bucket size totals for the entire problem. These
   will be used to determine the redistribution of keys */
MPI_Allreduce( bucket_size,
               bucket_size_totals,
               NUM_BUCKETS+TEST_ARRAY_SIZE,
               MPI_INT,
               MPI_SUM,
               MPI_COMM_WORLD );

#ifdef TIMING_ENABLED
    timer_stop( 3 );
    timer_start( 2 );
#endif

/* Determine Redistribution of keys: accumulate the bucket size totals

```

till this number surpasses NUM\_KEYS (which the average number of keys per processor). Then all keys in these buckets go to processor 0. Continue accumulating again until surpassing 2\*NUM\_KEYS. All keys in these buckets go to processor 1, etc. This algorithm guarantees that all processors have work ranking; no processors are left idle. The optimum number of buckets, however, does not result in as high a degree of load balancing (as even a distribution of keys as is possible) as is obtained from increasing the number of buckets, but more buckets results in more computation per processor so that the optimum number of buckets turns out to be 1024 for machines tested. Note that process\_bucket\_distrib\_ptr1 and ...\_ptr2 hold the bucket number of first and last bucket which each processor will have after the redistribution is done. \*/

```

bucket_sum_accumulator = 0;
local_bucket_sum_accumulator = 0;
send_displ[0] = 0;
process_bucket_distrib_ptr1[0] = 0;
for( i=0, j=0; i<NUM_BUCKETS; i++ )
{
    bucket_sum_accumulator += bucket_size_totals[i];
    local_bucket_sum_accumulator += bucket_size[i];
    if( bucket_sum_accumulator >= (j+1)*NUM_KEYS )
    {
        send_count[j] = local_bucket_sum_accumulator;
        if( j != 0 )
        {
            send_displ[j] = send_displ[j-1] + send_count[j-1];
            process_bucket_distrib_ptr1[j] =

```

```

                process_bucket_distrib_ptr2[j-1]+1;
            }
            process_bucket_distrib_ptr2[j++] = i;
            local_bucket_sum_accumulator = 0;
        }
    }

#ifdef TIMING_ENABLED
    timer_stop( 2 );
    timer_start( 3 );
#endif

/* This is the redistribution section: first find out how many keys
each processor will send to every other processor: */
MPI_Alltoall( send_count,
              1,
              MPI_INT,
              recv_count,
              1,
              MPI_INT,
              MPI_COMM_WORLD );

/* Determine the receive array displacements for the buckets */
recv_displ[0] = 0;
for( i=1; i<comm_size; i++ )
    recv_displ[i] = recv_displ[i-1] + recv_count[i-1];

/* Now send the keys to respective processors */
MPI_Alltoallv( key_buff1,

```

```

                send_count,
                send_displ,
                MPI_INT,
                key_buff2,
                recv_count,
                recv_displ,
                MPI_INT,
                MPI_COMM_WORLD );

#ifdef TIMING_ENABLED
    timer_stop( 3 );
    timer_start( 2 );
#endif

/* The starting and ending bucket numbers on each processor are
multiplied by the interval size of the buckets to obtain the
smallest possible min and greatest possible max value of any
key on each processor */
min_key_val = process_bucket_distrib_ptr1[my_rank] << shift;
max_key_val = ((process_bucket_distrib_ptr2[my_rank] + 1) <<
shift)-1;

/* Clear the work array */
for( i=0; i<max_key_val-min_key_val+1; i++ )
    key_buff1[i] = 0;

/* Determine the total number of keys on all other
processors holding keys of lesser value */
m = 0;
for( k=0; k<my_rank; k++ )

```

```

for( i= process_bucket_distrib_ptr1[k];
    i<=process_bucket_distrib_ptr2[k];
    i++ )
    m += bucket_size_totals[i]; /* m has total # of lesser keys */

/* Determine total number of keys on this processor */
j = 0;
for( i= process_bucket_distrib_ptr1[my_rank];
    i<=process_bucket_distrib_ptr2[my_rank];
    i++ )
    j += bucket_size_totals[i]; /* j has total # of local keys */

/* Ranking of all keys occurs in this section: */
/* shift it backwards so no subtractions are necessary in loop */
key_buff_ptr = key_buff1 - min_key_val;

/* In this section, the keys themselves are used as their
own indexes to determine how many of each there are: their
individual population */
for( i=0; i<j; i++ )
    key_buff_ptr[key_buff2[i]]++; /* Now they have individual key */
/* population */

/* To obtain ranks of each key, successively add the individual key
population, not forgetting to add m, the total of lesser keys,
to the first key population */
key_buff_ptr[min_key_val] += m;
for( i=min_key_val; i<max_key_val; i++ )
    key_buff_ptr[i+1] += key_buff_ptr[i];

```

```

/* This is the partial verify test section */
/* Observe that test_rank_array vals are */
/* shifted differently for different cases */
for( i=0; i<TEST_ARRAY_SIZE; i++ )
{
    k = bucket_size_totals[i+NUM_BUCKETS]; /* Keys were hidden
here */
    if( min_key_val <= k && k <= max_key_val )
        switch( CLASS )
        {
            case 'S':
                if( i <= 2 )
                {
                    if( key_buff_ptr[k-1] != test_rank_array[i]+iteration )
                    {
                        printf( "Failed partial verification: "
                            "iteration %d, processor %d, test key %d\n",
                            iteration, my_rank, i );
                    }
                }
            else
                passed_verification++;
        }
    else
    {
        if( key_buff_ptr[k-1] != test_rank_array[i]-iteration )
        {
            printf( "Failed partial verification: "
                "iteration %d, processor %d, test key %d\n",

```

```

        iteration, my_rank, i );
    }
    else
        passed_verification++;
}
break;
case 'W':
if( i < 2 )
{
    if( key_buff_ptr[k-1] !=
        test_rank_array[i]+(iteration-2) )
    {
        printf( "Failed partial verification: "
            "iteration %d, processor %d, test key %d\n",
            iteration, my_rank, i );
    }
    else
        passed_verification++;
}
else
{
    if( key_buff_ptr[k-1] != test_rank_array[i]-iteration )
    {
        printf( "Failed partial verification: "
            "iteration %d, processor %d, test key %d\n",
            iteration, my_rank, i );
    }
    else
        passed_verification++;
}
}

```

```

        break;
case 'A':
    if( i <= 2 )
    {
        if( key_buff_ptr[k-1] !=
            test_rank_array[i]+(iteration-1) )
        {
            printf( "Failed partial verification: "
                "iteration %d, processor %d, test key %d\n",
                iteration, my_rank, i );
        }
        else
            passed_verification++;
    }
    else
    {
        if( key_buff_ptr[k-1] !=
            test_rank_array[i]-(iteration-1) )
        {
            printf( "Failed partial verification: "
                "iteration %d, processor %d, test key %d\n",
                iteration, my_rank, i );
        }
        else
            passed_verification++;
    }
    break;
case 'B':
    if( i == 1 || i == 2 || i == 4 )
    {

```

```

if( key_buff_ptr[k-1] != test_rank_array[i]+iteration )
{
    printf( "Failed partial verification: "
           "iteration %d, processor %d, test key %d\n",
           iteration, my_rank, i );
}
else
    passed_verification++;
}
else
{
    if( key_buff_ptr[k-1] != test_rank_array[i]-iteration )
    {
        printf( "Failed partial verification: "
               "iteration %d, processor %d, test key %d\n",
               iteration, my_rank, i );
    }
    else
        passed_verification++;
}
break;
case 'C':
    if( i <= 2 )
    {
        if( key_buff_ptr[k-1] != test_rank_array[i]+iteration )
        {
            printf( "Failed partial verification: "
                   "iteration %d, processor %d, test key %d\n",
                   iteration, my_rank, i );
        }
    }
}

```

```

else
    passed_verification++;
}
else
{
    if( key_buff_ptr[k-1] != test_rank_array[i]-iteration )
    {
        printf( "Failed partial verification: "
               "iteration %d, processor %d, test key %d\n",
               iteration, my_rank, i );
    }
    else
        passed_verification++;
}
break;
}
}

```

/\* Make copies of rank info for use by full\_verify: these variables in rank are local; making them global slows down the code, probably since they cannot be made register by compiler \*/

```

if( iteration == MAX_ITERATIONS )
{
    key_buff_ptr_global = key_buff_ptr;
    total_local_keys = j;
    total_lesser_keys = m;
}

```

```

    }
}

/*****
*****/
/*****          M A I N          *****/
/*****
*****/

main( argc, argv )
    int argc;
    char **argv;
{

    int        i, iteration, itemp;

    double     timecounter, maxtime;

/* Initialize MPI */
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
    MPI_Comm_size( MPI_COMM_WORLD, &comm_size );

/* Initialize the verification arrays if a valid class */
    for( i=0; i<TEST_ARRAY_SIZE; i++ )
        switch( CLASS )

```

```

{
    case 'S':
        test_index_array[i] = S_test_index_array[i];
        test_rank_array[i] = S_test_rank_array[i];
        break;
    case 'A':
        test_index_array[i] = A_test_index_array[i];
        test_rank_array[i] = A_test_rank_array[i];
        break;
    case 'W':
        test_index_array[i] = W_test_index_array[i];
        test_rank_array[i] = W_test_rank_array[i];
        break;
    case 'B':
        test_index_array[i] = B_test_index_array[i];
        test_rank_array[i] = B_test_rank_array[i];
        break;
    case 'C':
        test_index_array[i] = C_test_index_array[i];
        test_rank_array[i] = C_test_rank_array[i];
        break;
};

/* Check that actual and compiled number of processors agree */
if( comm_size != NUM_PROCS )
{
    if( my_rank == 0 )
        printf( "\n ERROR: compiled for %d processes\n"

```



```

        " Number of active processes: %d\n"
        " Exiting program!\n\n", NUM_PROCS, comm_size );
MPI_Finalize();
exit( 1 );
}

/* Printout initial NPB info */
if( my_rank == 0 )
{
    printf( "\n\n NAS Parallel Benchmarks 2.3 -- IS Benchmark\n\n" );
    printf( " Size: %d (class %c)\n", TOTAL_KEYS, CLASS );
    printf( " Iterations: %d\n", MAX_ITERATIONS );
    printf( " Number of processes: %d\n", comm_size );
}
/* Generate random number sequence and subsequent keys on all procs
*/
create_seq( find_my_seed( my_rank,
                        comm_size,
                        4*TOTAL_KEYS,
                        314159265.00, /* Random number gen seed */
                        1220703125.00 ), /* Random number gen mult */
            1220703125.00 ); /* Random number gen mult */

/* Do one iteration for free (i.e., untimed) to guarantee initialization of
all data and code pages and respective tables */
rank( 1 );

/* Start verification counter */
passed_verification = 0;

if( my_rank == 0 && CLASS != 'S' ) printf( "\n iteration\n" );

/* Initialize timer */
timer_clear( 0 );

/* Initialize separate communication, computation timing */
#ifdef TIMING_ENABLED
    for( i=1; i<=3; i++ ) timer_clear( i );
#endif

/* Start timer */
timer_start( 0 );

#ifdef TIMING_ENABLED
    timer_start( 1 );
    timer_start( 2 );
#endif

/* This is the main iteration */
for( iteration=1; iteration<=MAX_ITERATIONS; iteration++ )
{
    if( my_rank == 0 && CLASS != 'S' ) printf( " %d\n", iteration );
    rank( iteration );
}

```

```

#ifdef TIMING_ENABLED
    timer_stop( 2 );
    timer_stop( 1 );
#endif

/* Stop timer, obtain time for processors */
timer_stop( 0 );

timecounter = timer_read( 0 );

/* End of timing, obtain maximum time of all processors */
MPI_Reduce( &timecounter,
            &maxtime,
            1,
            MPI_DOUBLE,
            MPI_MAX,
            0,
            MPI_COMM_WORLD );

#ifdef TIMING_ENABLED
{
    double  tmin, tsum, tmax;

    if( my_rank == 0 )
    {
        printf( "\ntimer 1/2/3 = total/computation/communication
time\n");
        printf( "          min          avg          max\n" );
    }
    for( i=1; i<=3; i++ )

```

```

{
    timecounter = timer_read( i );
    MPI_Reduce( &timecounter,
                &tmin,
                1,
                MPI_DOUBLE,
                MPI_MIN,
                0,
                MPI_COMM_WORLD );
    MPI_Reduce( &timecounter,
                &tsum,
                1,
                MPI_DOUBLE,
                MPI_SUM,
                0,
                MPI_COMM_WORLD );
    MPI_Reduce( &timecounter,
                &tmax,
                1,
                MPI_DOUBLE,
                MPI_MAX,
                0,
                MPI_COMM_WORLD );
    if( my_rank == 0 )
        printf( "timer %d:  %f          %f          %f\n",
                i, tmin, tsum/((double) comm_size), tmax );
    }
    if( my_rank == 0 )
        printf( "\n" );
}

```

```

#endif

/* This tests that keys are in sequence: sorting of last ranked key seq
   occurs here, but is an untimed operation */
full_verify();

/* Obtain verification counter sum */
itemp = passed_verification;
MPI_Reduce( &itemp,
           &passed_verification,
           1,
           MPI_INT,
           MPI_SUM,
           0,
           MPI_COMM_WORLD );

/* The final printout */
if( my_rank == 0 )
{
  if( passed_verification != 5*MAX_ITERATIONS + comm_size )
    passed_verification = 0;
  c_print_results( "IS",
                  CLASS,
                  TOTAL_KEYS,
                  0,
                  0,
                  MAX_ITERATIONS,
                  NUM_PROCS,
                  comm_size,
                  maxtime,
                  ((double) (MAX_ITERATIONS*TOTAL_KEYS))
                  /maxtime/1000000.,
                  "keys ranked",
                  passed_verification,
                  NPBVERSION,
                  COMPILETIME,
                  MPICC,
                  CLINK,
                  CMPI_LIB,
                  CMPI_INC,
                  CFLAGS,
                  CLINKFLAGS );
}

MPI_Finalize();

/*****
*/
} /* END PROGRAM */
/*****

```

