Wilson César Machado da Rocha

Gerenciamento de Tarefas Para Ambientes Paralelos

Dissertação apresentada ao Instituto de Pesquisas Tecnológicas do Estado de São Paulo – IPT, para obtenção do título de Mestre em Engenharia de Computação.

Área de concentração: Redes de Computadores

Orientador: Dr. Marco Dimas Gubitoso

São Paulo

Dez./2006

Ficha Catalográfica
Elaborada pelo Departamento de Acervo e Informação Tecnológica – DAIT
do Instituto de Pesquisas Tecnológicas do Estado de São Paulo – IPT

R672g **Rocha, Wilson César Machado da**
Gerenciamento de tarefas para ambientes paralelos. / Wilson César Machado da Rocha. São Paulo, 2006.
96p.

Dissertação (Mestrado em Engenharia de Computação) - Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Área de concentração: Redes de Computadores.

Orientador: Prof. Dr. Marco Dimas Gubitoso

1. Processamento paralelo 2. Processamento distribuído 3. MPI – Message Passing Interface 4. LAM – Local Area Multicomputer or Multiprocessor 5. MPI – Message Passing Interface 6. LPR – License Plate Recognition 7. Tese I. Instituto de Pesquisas Tecnológicas do Estado de São Paulo. Centro de Aperfeiçoamento Tecnológico II.Título

07-108 CDU 004.75(043)

Dedico este trabalho primeiramente a Deus, Origem de Tudo, Idealizador do Universo e Senhor Soberano sobre todas as vontades. Senhor, Teu é o Reino, o Poder e Toda a Glória Para Todo o Sempre.

Dedico também a Sebastião Machado Rocha e Helena de Paula Rocha, pais queridos e dedicados, responsáveis diretos pela educação que me possibilitou atingir o presente objetivo.

Dedico à minha esposa Ester Baptista dos Santos, incessante incentivadora da conclusão deste trabalho.

Dedicação especial ao Jared França Silva Neto, filho e inseparável companheiro com quem muito tenho aprendido sobre o real sentido da vida nesta nossa existência.

Dedico ainda ao Professor Dr. Marco Dimas Gubitoso, orientador e, acima de tudo, um grande amigo.

E, finalmente, para ser justo, dedico a todos que, de alguma forma, concorreram para a boa conclusão deste trabalho.

Muito Obrigado.

Agradeço primeiramente a Deus, Soberano Senhor do Universo e de tudo que nele existe.

Agradeço especialmente a meus pais Sebastião Machado Rocha e Helena de Paula Rocha, cujas incansáveis e incessantes dedicação e empenho me permitiram estar aqui redigindo este texto hoje.

Agradeço a minha querida esposa Ester Baptista dos Santos pela compreensão e constante incentivo.

Agradeço a minhas queridas irmãs Ilka Machado da Rocha Pinheiro e Iracema Machado da Rocha Camerlingo pelo apoio e incentivo que nunca me deixaram faltar.

Agradeço ainda a minhas cunhadas, Edna Baptista dos Santos Gubitoso e Eunice Baptista dos Santos Holanda de Góes, incansáveis incentivadoras.

Ao Professor Dr. Marco Dimas Gubitoso, orientador presente e sempre acessível nos momentos mais necessários, deixo um especial agradecimento por toda a colaboração e incentivo que me dispensou.

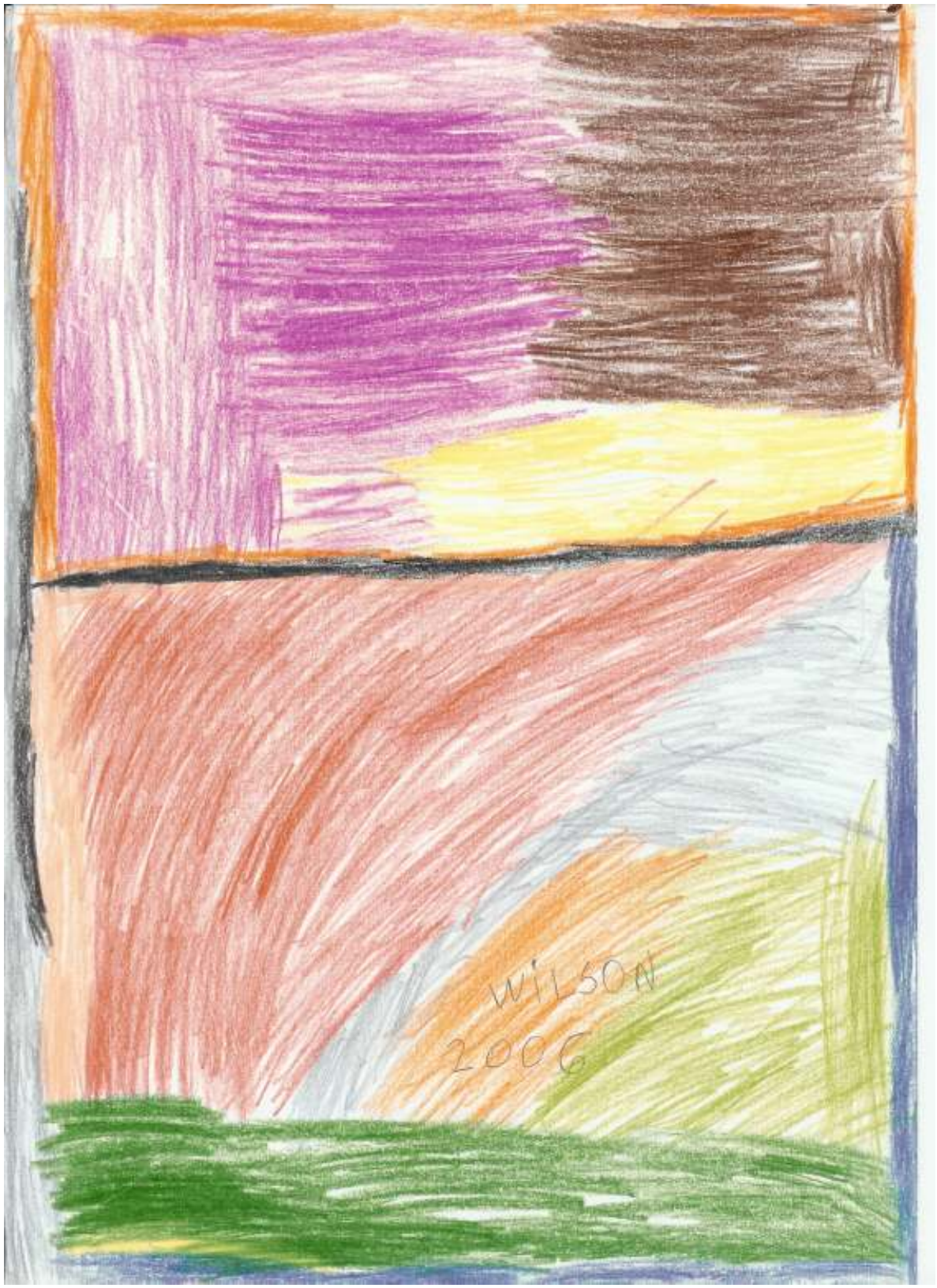
Agradeço ao Professor Dr. Flavius Portella Ribas Martins, examinador crítico que muito colaborou para a qualidade deste trabalho.

Especial agradecimento ao Professor Dr. Kunio Okuda, examinador e incentivador, cujas criticas concorreram definitivamente para a valorização do presente trabalho.

Agradeço ainda ao Professor Dr. Eduardo Aoun Tannuri, amigo, companheiro de trabalho e examinador suplente.

Finalmente, agradeço especialmente ao querido e amado filho Jared França Silva Neto, companheiro inseparável de todos os momentos, alguém que muito me ensinou sobre o real sentido da vida; criador da obra cuja reprodução ilustra a página seguinte (reservei o direito de manter o original comigo) e sem a qual este trabalho estaria definitiva e irrecuperavelmente incompleto.

Muito Obrigado a Todos que, de alguma forma, concorreram para a consecução a bom termo, do presente trabalho.


WILSON
2006

RESUMO

Este trabalho apresenta um mecanismo de uso geral capaz de distribuir e gerenciar aplicações passíveis de serem divididas em tarefas menores e independentes, permitindo sua distribuição entre diversas máquinas interligadas em rede e o seu processamento em paralelo. Para atingir tal objetivo foram empregadas técnicas de paralelização e processamento paralelo sobre ambientes distribuídos que seguem o modelo de passagem de mensagem de acordo com o padrão consensual MPI (*Message Passing Interface*) segundo a implementação LAM – MPI (*Local Área Multicomputer (or Multiprocessor) – Message Passing Interface*). Alguns conceitos fundamentais são apresentados seguidos de uma explanação sucinta do mecanismo proposto, da descrição de uma simulação de um caso específico (LPR – *License Plate Recognition*) que motivou o trabalho. Por fim, são apresentados os resultados obtidos.

Palavras-chave: processamento paralelo; processamento distribuído; passagem de mensagem; MPI.

ABSTRACT

MANAGEMENT OF TASKS FOR PARALLEL ENVIROMENTS

This work proposes a generic mechanism which is able to distribute and manage applications which may be divided into smaller and independent tasks, allowing their distribution in a computer network and processed in parallel. To reach this goal, the parallellization and parallel processing techniques over distributed enviroments were used, following message passing paradigm and according to MPI standard and implementation LAM – MPI. Some fundamental concepts associated to parallel and distributed processing and message passing model as its implementation LAM – MPI are presented, followed by a brief explanation of the proposed mechanism, the description of a simulation of a specific case (LPR – License Plate Recognition) which has motiveted this work. At the end the obtained results are shown.

Keywords: parallel processing ; distributed processing ; message passing; MPI.

Lista de Figuras

Lista de Quadros

Lista de Tabelas

Lista de Abreviaturas e Siglas

MPI	**Message Passing Interface – Interface de Passagem de Mensagem**
LAM MPI	**Local Área Multiprocessor (Multicomputer) Message Passing Interface – Interface de Passagem de Mensagem em ambiente de Multiprocessadores (Multicomputadores) em Área Local**
SISD	**Single Instruction Single Data – Instrução Única Dado Único**
MISD	**Multiple Instructions Single Data – Instruções Múltiplas Dado Único**
SIMD	**Single Instruction Multiple Data – Instrução Única Dados Múltiplos**
MIMD	**Multiple Instructions Multiple Data – Instruções Múltiplas Dados Múltiplos**
SPSD	**Single Program Single Data – Programa Único Dado Único**
MPSD	**Multiple Programs Single Data – Programas Múltiplos Dado Único**
SPMD	**Single Program Multiple Data – Programa Único Dados Múltiplos**
MPMD	**Multiple Programs Multiple Data – Programas Múltiplos Dados Múltiplos**
PVM	**Parallel Virtual Machine – Máquina Virtual Paralela**
IPT	**Instituto de Pesquisas Tecnológicas do Estado de São Paulo**
RAM	**Random Access Memory – Memória de Acesso Aleatório**
TCP IP	**Transmission Control Protocol Internet Protocol – Protocolo de de Controle de Transmissão Protocolo Internet**
MPICH	**Message Passing Interface Chamaleon – Interface de Passagem de Mensagem Camaleão**

SUMÁRIO

1 INTRODUÇÃO

Existem aplicações que dependem de dados provenientes de eventos aleatórios no tempo sobre os quais não se tem nenhum controle, ou mesmo previsão, sobre o exato momento em que irão acontecer; quando possível, pode-se apenas assegurar que irão ocorrer sob determinadas circunstâncias e dentro de um período de tempo. Normalmente estão relacionadas a fenômenos naturais como, por exemplo, descargas atmosféricas e abalos sísmicos, mas podem ocorrer também sobre outros temas

Apesar desta peculiaridade, a detecção desses eventos e o registro dos dados a eles associados são imprescindíveis para o seu estudo. Assim, os equipamentos responsáveis por monitorar e coletar esses dados têm que estar instalados, funcionando e aptos a exercer suas funções a qualquer instante.

Aplicações assim caracterizadas necessitam de recursos reservados exclusivamente para a realização de tais tarefas ou, minimamente, que elas sejam priorizadas em relação a outras.

A prioridade dessas tarefas garante que as operações de detecção, coleta e armazenamento dos dados associados interrompam qualquer outro processo em execução na máquina destinada a esse fim. Entretanto, a ocorrência de situações limite em que o fluxo de dados seja intenso, inviabiliza a execução de qualquer outra tarefa, pois as interrupções seriam tão freqüentes que a máquina praticamente se limitaria à detecção desses eventos, a coleta dos dados associados e seu respectivo armazenamento.

A análise de situações como esta conduz à conclusão de que, se os dados são armazenados e não se consegue processá-los, então, em algum momento a capacidade de armazenamento da máquina também será exaurida e, consequentemente, dados serão perdidos e o sucesso da aplicação estará comprometido, condição preliminarmente indesejável e que conduziu ao presente raciocínio, portanto, esta situação também deve ser evitada.

Por outro lado, a reserva de recursos exclusivos para a realização das tarefas de detecção do evento, coleta e armazenamento dos dados a ele associados e a posterior transmissão destes para serem processados em outra máquina, em princípio solucionaria esta incoerência, porém, há que se observar que o simples deslocamento dos dados para outra máquina

apenas transfere também o problema, pois se o fluxo de dados for suficientemente intenso para exceder as capacidades de armazenamento e processamento desta segunda máquina, então o problema perdura, apenas em local diferente.

Adicionalmente, se for desejável, por qualquer razão que seja, que os dados coletados sejam processados em curto espaço de tempo ou mesmo em tempo real, o problema se agrava.

Aplicações assim caracterizadas sugerem que a distribuição do processamento e, portanto, a ampliação dos recursos disponíveis, é uma alternativa que merece ser estudada.

Este universo contempla ainda subconjuntos que trazem consigo grande quantidade de paralelismo inerente, como normalmente ocorre em aplicações que abrangem visão computacional cujas operações envolvidas são naturalmente paralelizáveis (ROMIG III; SAMAL, 1995).

Aplicações que congregam essas peculiaridades sugerem um ambiente propício, simultaneamente, à distribuição e paralalelização do processamento.

O presente trabalho propõe um mecanismo genérico cuja expectativa é de que seja capaz de viabilizar a distribuição e o gerenciamento de processos associados a aplicações desse tipo com desempenho superior. Para a consecução de tais objetivos são utilizadas técnicas de paralelização de processamento em ambientes distribuídos mediante o uso do modelo de *passagem de mensagem*, conforme estabelecido no padrão *MPI – Message Passing Interface* (Interface de Passagem de Mensagem)[*] e segundo a implementação de livre uso *LAM - MPI – Local Area Multiprocessors (Multicomputers) – Message Passing Interface* (Multiprocessadores ou Multicomputadores Locais – Interface de Passagem de Mensagem)[**].

Neste âmbito, a exposição que se segue apresenta, na seção 2 – *Noções Preliminares*, conceitos fundamentais associados ao processamento paralelo e / ou distribuído

[*] MPI é uma especificação de biblioteca de passagem de mensagem cujos detalhes são expostos adiante.
[**] LAM MPI é uma implementação de livre uso que segue o padrão estabelecido no MPI, cujos detalhes podem ser obtidos em http://www.lam-mpi.org.

indispensáveis à compreensão do restante do texto; a seção 3 discute processamento paralelo e distribuído e sua relação com o modelo de *passagem de mensagem explicita*[*]; a seção 4 trata da proposta *MPI – Message Passing Interface*; a seção 5 discute um mecanismo de uso geral para paralelização / distribuição; a seção 6 descreve um exemplo de aplicação, especificamente aquele que motivou este trabalho, a seção 7 explana o mecanismo genérico para distribuição e gerenciamento de processos proposto para a solução de problemas similares ao do exemplo; a secção 8 trata da simulação do exemplo de aplicação, apresenta os resultados obtidos e sua respectiva análise; finalmente, a seção 10 é reservada para as considerações finais e perspectivas para futuros desenvolvimentos.

[*] O modelo de passagem de mensagem explicita é um meio atraves do qual se provê sincronismo e intercambio de dados entre processos paralelos e / ou distribuídos.

2 NOÇÕES PRELIMINARES

Alguns conceitos básicos relacionados, de alguma forma, ao processamento paralelo e / ou distribuído, são fundamentais à compreensão deste texto e, portanto, são expostos a seguir.

2.1 Memória Compartilhada e Memória Distribuída

Máquinas paralelas admitem dois tipos básicos de arquitetura quanto a sua disposição do espaço de memória: as de memória distribuída e as de memória compartilhada.

Máquinas paralelas de memória distribuída são compostas essencialmente por um conjunto de máquinas seqüenciais ou seriais, dotadas de processadores e memória, denominadas *nós*, que trabalham coletiva e solidariamente na solução de um problema. Cada nó é capaz de rápida e eficazmente ter acesso a sua própria memória local por meio de um barramento interno de alta velocidade, bem como a de outros nós através de algum tipo de rede de comunicação. Tal configuração permite que dados sejam trocados e compartilhados entre os diversos nós por meio de mensagens que percorrem a rede. A figura 1 abaixo ilustra este tipo de arquitetura.

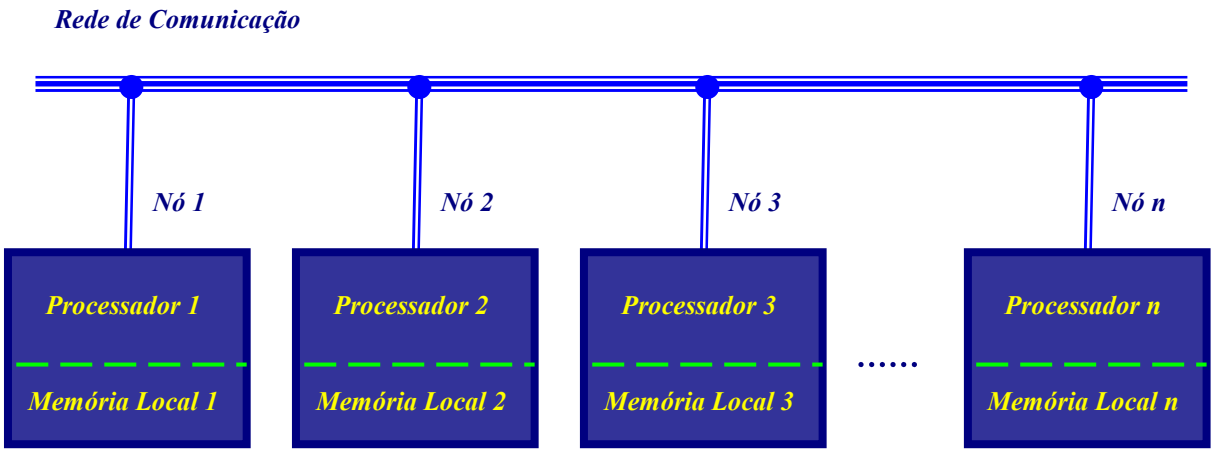


Figura 1 – Arquitetura Paralela de Memória Distribuída

Máquinas paralelas de memória compartilhada são aquelas dotadas de múltiplos processadores que compartilham o acesso a um espaço comum de memória, denominado *memória global*, por meio um barramento local de alta velocidade que possibilita aos processadores trocarem e / ou compartilharem o acesso aos dados de maneira rápida e eficiente. Atualmente, é comum que o número de processadores utilizados neste tipo de arquitetura não exceda algumas poucas dezenas, isto ocorre devido aos limites físicos impostos à velocidade de transmissão de dados através do barramento que, por sua vez, restringem a quantidade de dados que podem ser processados simultaneamente.

A figura 2 abaixo ilustra este tipo e arquitetura.

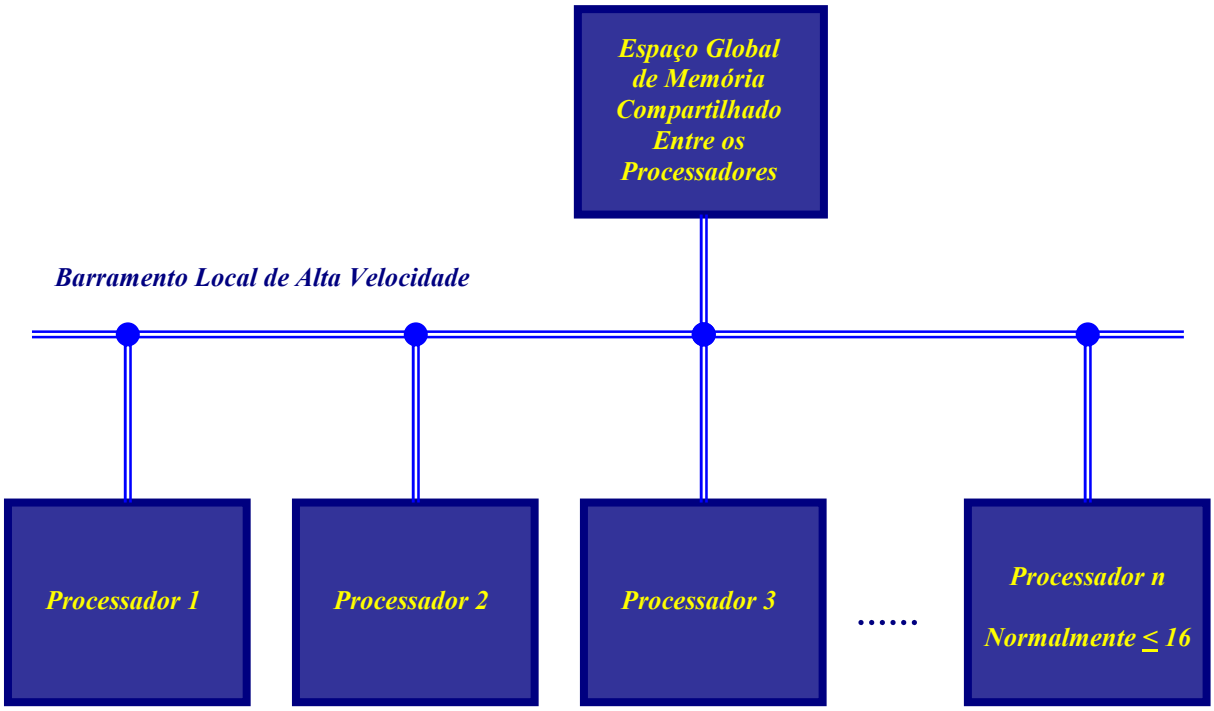


Figura 2 – Arquitetura Paralela de Memória Compartilhada

Um terceiro tipo, denominado arquitetura híbrida, combina os dois primeiros. Neste caso, cada nó pode consistir de um processador dotado de memória local ou de um grupo de algumas dezenas de processadores interligados a um banco de memória compartilhado. O conjunto assim composto se conecta a algum tipo de rede de comunicação capaz de prover o meio através do qual a troca e o compartilhamento de dados possa se efetivar.

Implementações desse tipo permitem que os processadores tenham acesso tanto à memória local do nó a que pertencem quanto a bancos de memória remotos pertencentes a outros nós.

2.2 Fluxo de Dados e Fluxo de Instruções

Os computadores podem também ser classificados segundo a maneira como dados e instruções fluem através da máquina.

Essencialmente, o fluxo de instruções está relacionado ao programa executado, enquanto que o fluxo e dados diz respeito aos operandos manipulados pelo programa.

Apesar de se relacionarem entre si, esses fluxos são considerados independentes.

Nesse contexto, Flynn (1972) apresentou uma proposta de classificação amplamente difundida e aceita. Nela as máquinas são divididas em quatro classes segundo o relacionamento estabelecido entre esses fluxos, conforme as combinações ilustradas na figura 3 abaixo.

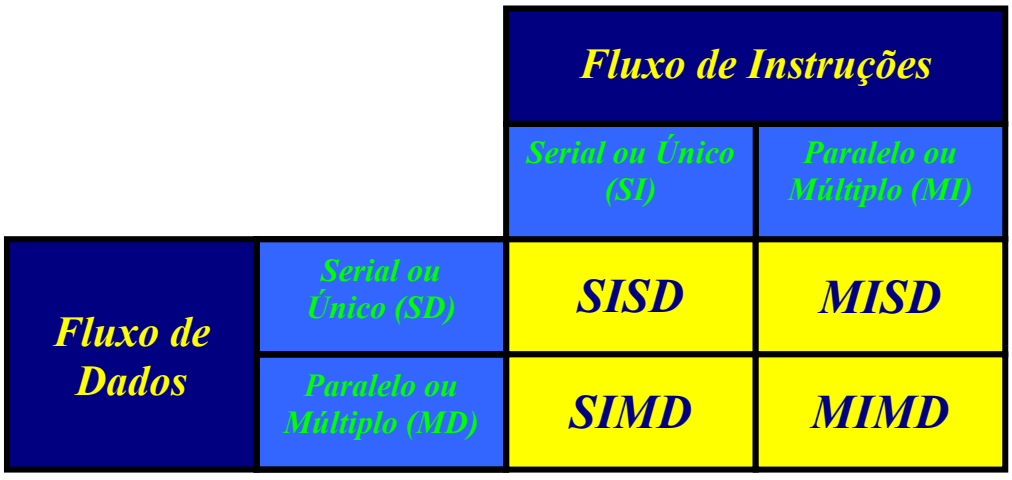


Figura 3 – Classificação de Flynn

2.2.1 SISD – Instrução Única Dado Único (*Single Instruction Single Data)*

Nesta classe um único fluxo de instruções opera sobre um único fluxo de dados. Corresponde ao processamento seqüencial característico da máquina de Von Neumann - o computador serial (ou seqüencial). Nela, um único fluxo de instruções (SI), ordenado por um ponteiro denominado *contador de programa*, opera sobre um único dado de cada vez (SD).

Neste caso, apesar de os programas estarem organizados através de instruções seqüenciais, estas podem ser executadas sobrepostas em diferentes estágios ("*pipelining*").

Arquiteturas SISD caracterizam-se por terem apenas uma unidade de controle mas podem possuir mais de uma unidade de processamento.

A figura 4 abaixo ilustra o modelo SISD.

Figura 4 – Modelo SISD

2.2.2 SIMD – Instrução Única Dados Múltiplos (*Single Instruction Multiple Data*)

Nesta classe, vários dados são operados simultaneamente mediante o comando de uma única instrução. Maquinas que seguem este modelo possuem uma única unidade de controle e diversas unidades de processamento. Nelas o programa ainda é organizado de forma seqüencial e, para possibilitar o acesso a múltiplos dados, é preciso que a memória seja dividida em diversos módulos.

Arquiteturas assim são encontradas nos computadores vetoriais e matriciais.

A figura 5 subseqüente ilustra o modelo SIMD.

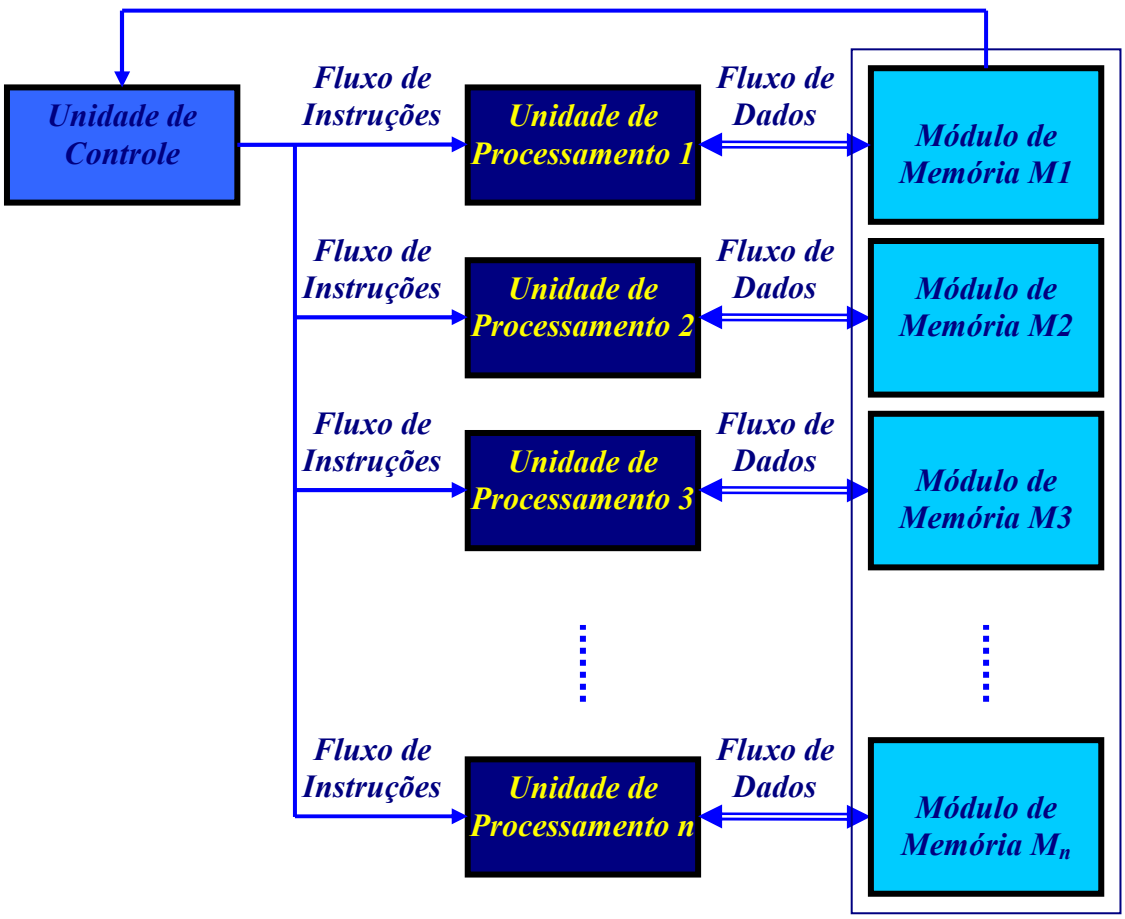


Figura 5 – Modelo SIMD

2.2.3 MISD – Instruções Múltiplas Dado Único (*Multiple Instructions Single Data*)

Esta classe prevê múltiplas unidades de controle executando, simultaneamente, distintas instruções sobre o mesmo dado. "Não faz muito sentido, embora alguns autores procurem classificar desta maneira algumas máquinas que não se encaixam bem nos outros casos" (GUBITOSO – 2005). Na realidade, a tecnologia que se dispõe atualmente é incapaz de possibilitar a implementação de uma máquina assim, portanto, esta classe não representa nenhum modelo de programação existente.

A figura 6 subseqüente ilustra o modelo MISD.

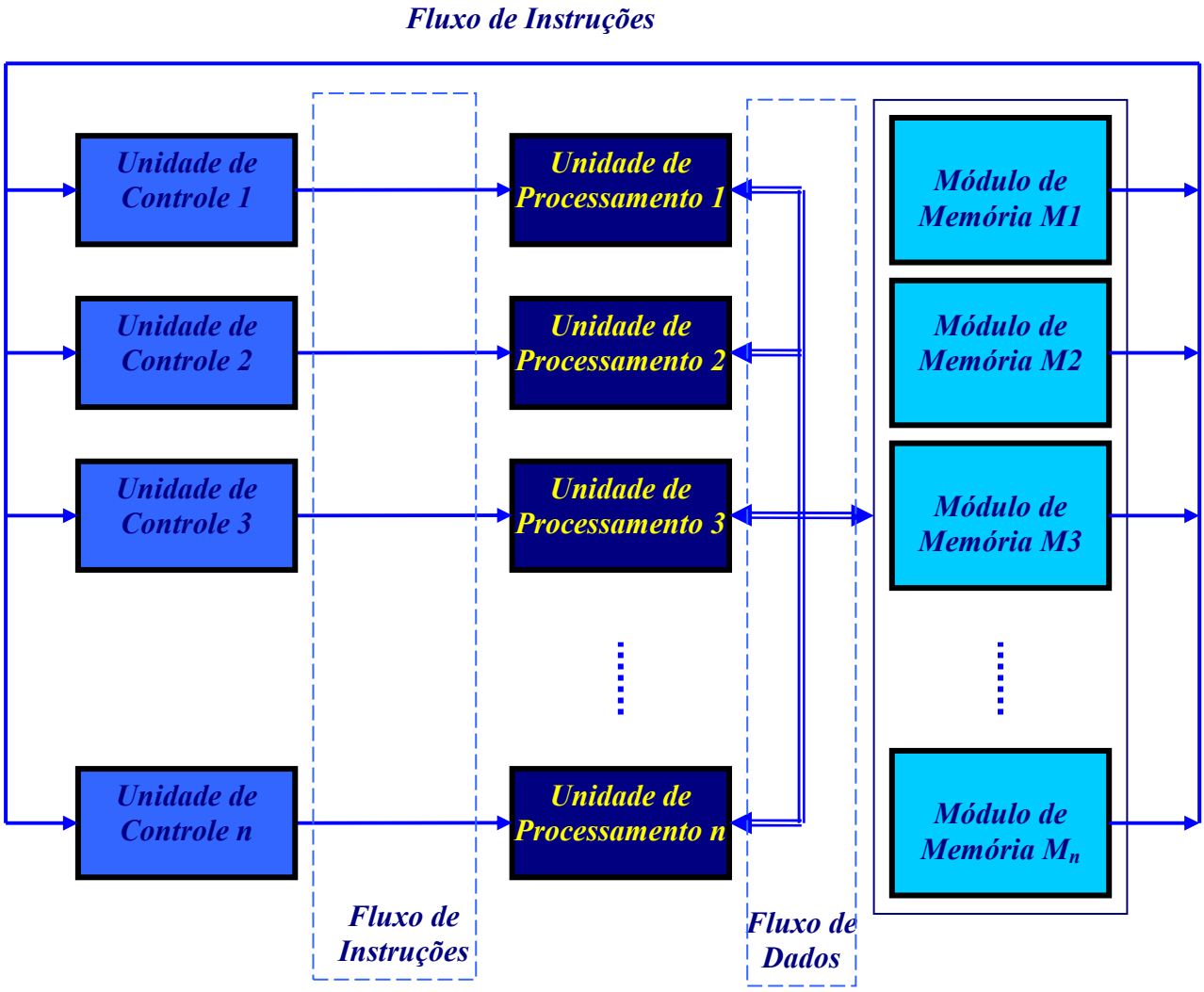


Figura 6 – Modelo MISD

2.2.4 MIMD – Instruções Múltiplas Dados Múltiplos (*Multiple Instructions Multiple Data*)

Esta classe prevê o processamento de múltiplos dados por múltiplas instruções simultaneamente. Neste modelo a sincronização e a comunicação entre os processadores são imprescindíveis.

A figura 7 abaixo ilustra o modelo MIMD.

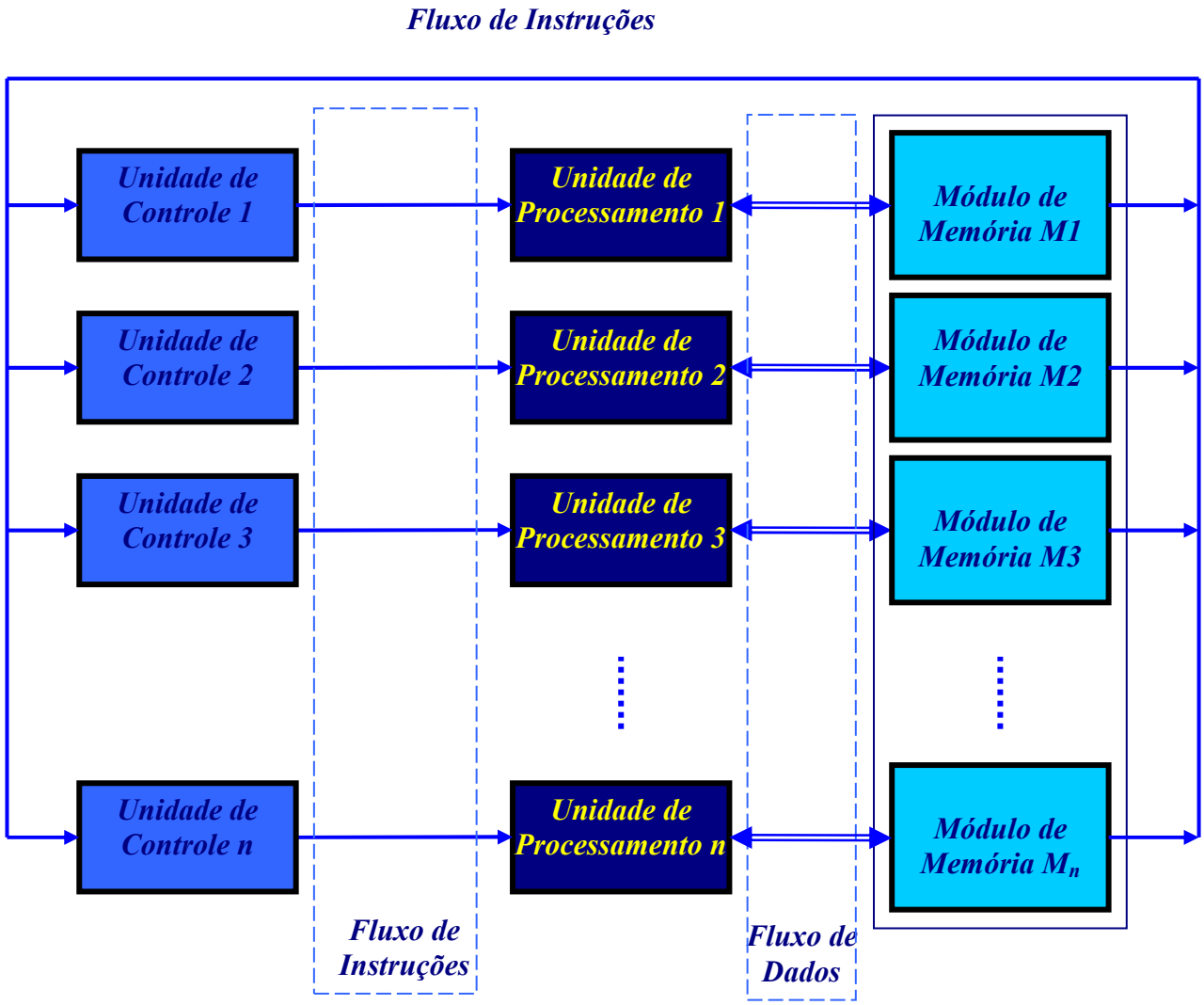


Figura 7 – Modelo MIMD

Em ambientes de processamento paralelo / distribuído os programas, constituídos por seqüências de instruções, operam sobre dados, assim, para adequar a nomenclatura proposta por Flynn a esses ambientes, é comum encontrar as designações: *SPSD – Single Program Single Data*, *MPSD – Multiple Program Single Data*, *SPMD – Single Program Multiple Data* e *MPMD – Multiple Program Multiple Data*, que correspondem, respectivamente, às classes SISD, MISD, SIMD e MIMD originalmente propostas

2.3 Decomposição

Quando se distribui e paraleliza processamento, procura-se essencialmente melhorar o desempenho de programas utilizando com maior eficiência os recursos de máquinas disponíveis. Para tanto, a primeira tarefa a ser executada quando se elabora algoritmos paralelos é dividir o problema original em problemas menores cujas soluções possam ser alcançadas de forma independente, possibilitando que sejam processados simultaneamente, contribuindo, dessa forma, decisivamente para a eficiência global do processo na obtenção do resultado final do problema originalmente proposto.

A decomposição do problema pode ocorrer de diversas maneiras, entre elas, duas são aqui tratadas: a por *domínio* e a por *tarefas*.

Na *decomposição de domínio*, também denominada *paralelismo de dados*, os dados são divididos em partes de aproximadamente igual tamanho e distribuídos entre os diversos processos que compõem o conjunto. Cada processo opera sobre a porção de dados que lhe é atribuída. Neste caso, quase sempre é necessário que esses processos se comuniquem e troquem dados periodicamente.

Note-se que programas consistem de seqüências de instruções elementares que são sequencialmente aplicadas aos dados, assim, uma instrução só é iniciada depois que sua antecessora foi finalizada. Esta maneira de atuar compara-se à classe SIMD estipulada por Flynn, pois conjuntos de dados diferentes são entregues à diversos processos que executam sobre eles, simultaneamente, o mesmo código.

Estratégias deste tipo são normalmente empregadas na solução de problemas em que os processos podem operar independentemente sobre grandes quantidades de dados, efetuando, a cada interação, apenas trocas de conjuntos diminutos de dados de fronteira que têm que ser compartilhados.

Muitas vezes a decomposição de domínio pode necessitar de ajustes que venham a distribuir melhor a carga de trabalho entre os diversos processos participantes ou ainda pode se revelar uma estratégia inadequada para uma aplicação específica. Isto pode ocorrer, por exemplo, quando as porções de dados entregues aos diferentes processos, embora de

tamanhos similares, necessitam de tempos muito diferentes para serem processados, comprometendo, dessa forma, o desempenho global do programa que fica, assim, vinculado a finalizações de processos cujos tempos de execução diferem muito, limitando a velocidade global àquela do processo mais lento, ao mesmo tempo em que o restante dos processos permanecem ociosos enquanto aguardam o término do mais lento sem realizar nenhum trabalho útil. Situações assim podem obter melhor solução adequando-se o *balanceamento de carga*[*] ou utilizando-se *decomposição funcional*.

Na *decomposição funcional* ou *paralelismo de tarefas*, o problema original é decomposto em tarefas menores que são distribuídas aos processos tão logo eles se tornem disponíveis, fazendo com que aqueles que finalizam suas tarefas mais rapidamente recebam novos trabalhos, evitando a ociosidade e acrescentando eficiência ao processo global.

Este tipo de decomposição é normalmente implementado sobre um modelo cliente – - servidor. Nele, um processo designado "*mestre*" distribui diferentes tarefas a um conjunto de processos denominados "*operários*". O modelo não impede que o processo mestre também realize algumas das tarefas e pode ser implementado, virtualmente, a qualquer nível de um programa, por exemplo: se simplesmente se deseja executar um programa com múltiplas entradas, uma implementação deste tipo distribui cópias do código serial entre os processos operários (*clientes*) enquanto o *servidor* remete diferentes dados de entrada para cada *cliente*; cada vez que um processo finaliza sua tarefa, um novo dado de entrada lhe é enviado.

A figura 8 subseqüente ilustra este modelo.

[*] Balancemento de carga estipula que a carga de trabalho deve ser igualmente distribuída entre processos paralelos e / ou distribuídos, de modo que os tempos de execução dos algoritmos envolvidos em cada um dos processos envolvidos na solução de um determinado problema global sejam muito próximos uns dos outros, de maneira que nenhum processo emperre o trabalho dos outros ou ainda que uma grande quantidade de processos permaneçam ociosos aguardando as finalizações de outros poucos.

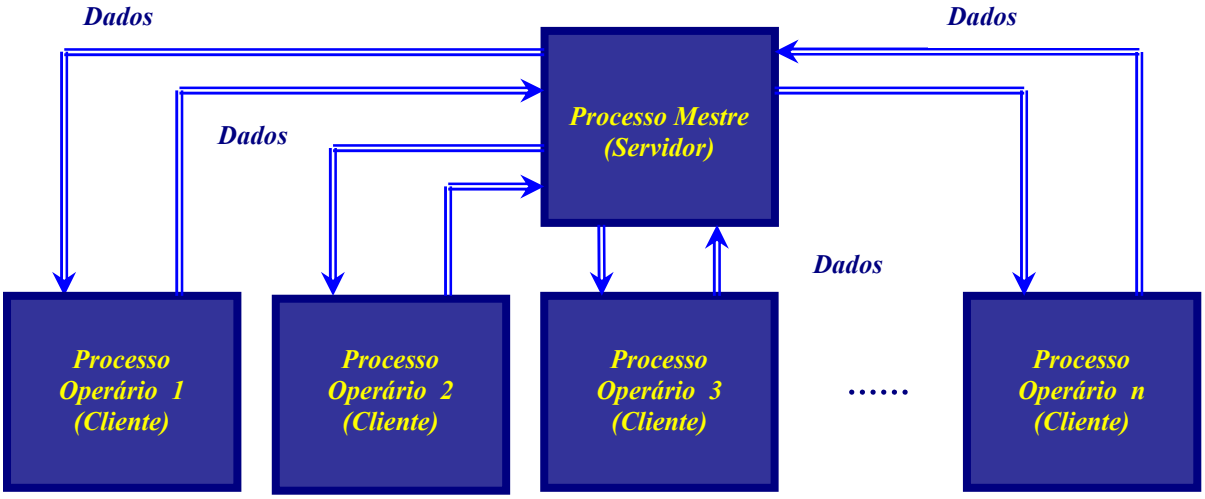


Figura 8 – Modelo Cliente Servidor

2.4 Modelos Dados Paralelos e Passagem de Mensagem

Historicamente, a programação paralela foi abordada de duas maneiras. A primeira delas utilizando *linguagem de dados paralelos baseadas em diretivas* e a outra usando a *passagem de mensagem explicita* através de chamadas a bibliotecas de linguagens de programação padrão como o C ou o Fortran.

Linguagens de dados paralelos baseadas em diretivas fazem com que um programa seqüencial torne-se paralelo mediante a adição de diretivas que aparecem como comentários no código seqüencial e indicam ao compilador como distribuir dados e tarefas entre os processos e este se encarrega dos detalhes de como realizar a distribuição dos dados, a computação e a comunicação. *High Performance Fortran* – HPF e *OpenMP* são exemplos de linguagens de dados paralelos baseadas em diretivas.

Na passagem de mensagem o programador fica responsável pela divisão explicita de dados e tarefas entre os processos bem como pelo gerenciamento da comunicação entre eles, o que atribui grande flexibilidade ao modelo.

2.5 Questões Relativas a Programação Paralela

O principal objetivo da programação paralela é obter desempenho superior àquele apresentado pela versão seqüencial análoga. Este pensamento conduz a questionamentos que necessitam ser considerados sempre que se implementa um programa paralelo que persiga o melhor desempenho possível mediante as restrições inerentes ao problema apresentado.

Neste cenário, três questões básicas merecem atenção: o balanceamento de carga, a minimização da comunicação e a sobreposição desta com a computação.

2.5.1 Balanceamento de Carga

Balanceamento de carga é a tarefa de dividir igualmente a carga de trabalho entre os processos. Isto é fácil de se conseguir quando todos os processos executam as mesmas operações sobre diferentes porções de dados, mas não é trivial quando o tempo de processamento depende dos valores dos dados que estão sendo trabalhados. Se as variações nos tempos de processamento forem grandes e não for possível diminuí-las então a possibilidade de processamento paralelo deve ser abandonada.

2.5.2 Minimização da Comunicação

Quando se trabalha com programação paralela, o tempo de execução total de um programa compõe-se, basicamente, por três elementos: o *tempo de computação*, o *tempo ocioso* e o *tempo de comunicação*.

Tempo de computação é aquele gasto efetivamente na realização de operações sobre os dados. Idealmente espera-se que se N processos são empregados paralelamente na solução de um problema, então o problema deve ser resolvido num período N vezes menor que aquele gasto na versão seqüencial. Isto seria verdade se todo tempo gasto pelos processos fosse empregado apenas na computação, infelizmente isto não é verdade.

Tempo ocioso é aquele em que o processo fica aguardando dados provenientes de outros processos e não realiza trabalho útil, devendo, portanto, ser minimizado.

Finalmente, *tempo de comunicação* é aquele utilizado pelos processos para transmitir e receber mensagens. O peso da comunicação no tempo de execução total de um programa paralelo pode ser avaliado em termos de *latência* e *largura de banda*. *Latência* é o tempo gasto para montar o pacote de comunicação, enquanto que a *largura de banda* é a velocidade de transmissão de dados dada em bits por unidade de tempo. Considerando que nos programas seqüenciais não ocorre comunicação inter processos, é bastante aconselhável que este parâmetro seja minimizado para que se alcance o melhor desempenho possível.

2.5.3 Sobreposição de Comunicação e Computação

Existem diversas maneiras de se reduzir o tempo ocioso nos processos. Uma delas é sobrepor comunicação e computação. Isto envolve ocupar um processo com uma ou mais novas tarefas enquanto ele aguarda pela finalização de uma comunicação.

Isto é possível mediante o emprego de técnicas de comunicação não bloqueante[*], porém, na prática, intercalar comunicação e computação não é tarefa trivial.

[*] Comunicação não bloqueante permite ao processo associado realizar outras tarefas enquanto aguarda a conclusão da comunicação.

3 PROCESSAMENTO PARALELO / DISTRIBUÍDO – MODELO DE PASSAGEM DE MENSAGEM

Processamentos paralelo e distribuído são áreas distintas da ciência da computação que têm experimentado forte inter relacionamento e rápida convergência, sobretudo nas últimas décadas, conforme demonstram o crescente número de trabalhos orientados no sentido de aplicar computação paralela sobre plataformas distribuídas apresentados nesse período (LINHALEIS; FIATS, 1998).

Originalmente a computação paralela tem por objetivo diminuir o tempo de execução de programas mediante a divisão de tarefas a serem realizadas simultaneamente enquanto que a computação distribuída visa o compartilhamento de recursos e serviços (GUBITOSO, 2005). É de se esperar que a convergência dessas duas áreas venha a proporcionar melhor desempenho global dos programas tanto pelo aumento da velocidade de execução quanto pela utilização racionalizada dos recursos e serviços disponibilizados em um ambiente de processamento paralelo aplicado sobre plataformas distribuídas.

Composições desse tipo exigem comunicação e sincronismo entre os processos, sejam eles locais ou distribuídos.

A computação paralela consiste de processos que operam dados localmente, cada qual possuindo apenas variáveis locais e desprovidos de mecanismos que permitam que uns tenham acesso diretamente à memória dos outros, é portanto necessário que se estipulem meios para o intercâmbio de informações entre eles.

A proposta de *passagem de mensagem* atende esta necessidade estabelecendo mecanismos que contemplam os requisitos de comunicação e sincronismo exigidos, propiciando o compartilhamento de dados mediante a transmissão e recepção explicitas destes entre os processos. Adicionalmente, permite que o programador divida e distribua, explicitamente, dados e tarefas entre os processos e gerencie a comunicação entre eles, atribuindo, dessa forma, grande flexibilidade ao modelo.

Apesar de a literatura normalmente tratar o modelo de passagem de mensagem como uma forma de troca de dados entre *processadores*, é preciso lembrar que ele considera *processos* que podem ou não serem executados em diferentes processadores. Geralmente assume-se que diferentes processos sejam executados em diferentes processadores (*processamento distribuído* e *paralelismo real*) e os termos *processos* e *processadores* se misturam, mas isto não é, em absoluto, uma regra, ao contrário, o modelo admite, indistintamente, processos diferentes sendo executados em um único processador (processamento *pseudo paralelo*) ou em vários, portanto, pode ocorrer a divisão de tarefas baseada em compartilhamento de tempo de uso dos recursos de um único processador entre diversos processos correntes.

Por ser extremamente generico, o modelo de passagem de mensagem é de grande utilidade. Essencialmente, qualquer tipo de computação paralela pode se utilizar dele, podendo ser implementado sobre uma ampla variedade de plataformas que abrange desde multiprocessadores de memória compartilhada até redes de estações de trabalho ou ainda máquinas dotadas de um único processador. Geralmente permite maior controle sobre a alocação e o fluxo de dados que, por exemplo, aquele apresentado pelo modelo de memória compartilhada. Assim, os programas podem, freqüentemente, alcançar melhor desempenho ao se utilizarem do modelo passagem de mensagem explícita.

Seguramente, desempenho é a razão fundamental pela qual é bastante improvável que o modelo de passagem de mensagem venha a desaparecer do universo da programação paralela a curto prazo. Nem mesmo o surgimento de redes de comunicação mais velozes ou ainda a utilização de máquinas paralelas de arquitetura híbrida devem ameaçá-lo de obsolescência, pois, sempre, em algum nível e de alguma forma, a troca de mensagens se fará necessária (LINHALIS; FIATS, 1998).

Entre os modelos propostos para passagem de mensagem dois se destacam: o PVM – *Parallel Virtual Machine* (*Máquina Paralela Virtual*) e o MPI – *Message Passing Interface* (*Interface de Passagem de Mensagem*).

Ambos são amplamente difundidos e aplicados. Encontram-se em constante aperfeiçoamento graças ao empenho de grupos ao redor do mundo que se dedicam ao desenvolvimento de ambos.

O PVM constitui-se de implementações não padronizadas compostas, essencialmente, por um conjunto integrado de ferramentas de *software* e bibliotecas que emulam uma estrutura computacional concorrente, de propósito geral, flexível e heterogênea, sobre computadores de arquitetura variada interligados através de uma rede de comunicação. Tal estrutura, denominada "*máquina virtual*", tem por objetivo primordial propiciar que as máquinas componentes de tal arranjo sejam utilizadas cooperativamente em aplicações de processamento concorrente ou paralelo.

O MPI por sua vez é uma especificação e constitui um padrão consensual na medida em que foi elaborado com a participação de fornecedores, pesquisadores, acadêmicos, desenvolvedores e usuários representantes de mais ou menos quarenta instituições diferentes que o adotaram (ALVIM, 2006).

Portabilidade, desempenho e funcionalidade são características presentes na proposta MPI, porém o fato de ser uma especificação e, portanto, permitir diferentes implementações que sigam o padrão consensual, confere a essa proposta um alto grau de flexibilidade que, associado à variedade de implementações de qualidade já disponíveis, contribui decisivamente para sua difusão e adoção.

O presente trabalho utiliza o MPI.

4 INTERFACE DE PASSAGEM DE MENSAGEM – MPI (MESSAGE PASSING INTERFACE)

MPI é uma especificação de biblioteca de funções ou sub rotinas de linguagens de programação padrão (como a C ou a Fortran) que obedece as regras estabelecidas pelo modelo de passagem de mensagem.

Quando inseridas no código fonte, essas funções ou sub rotinas estabelecidas pelas diferentes implementações que seguem o padrão MPI promovem a comunicação de dados e o sincronismo entre os processos.

Cada implementação é livre, dentro das regras estipuladas no MPI, para fazer o conjunto de funções ou sub rotinas destinadas especificamente a promover o sincronismo e a comunicação de dados entre processos que irão compor sua biblioteca, a qual será acrescentada à respectiva linguagem padrão a que são destinadas.

A primeira versão do MPI surgiu em 1994 e foi denominada MPI – 1. Nela foram especificados os nomes, as seqüências de chamadas e os resultados das sub rotinas e funções a serem chamadas pelo Fortran 77 e / ou pelo C respectivamente. Todas as implementações do MPI devem estar em conformidade com as regras ali estabelecidas para que se assegure a portabilidade.

Programas MPI devem poder ser compilados e executados sobre qualquer plataforma que admita o padrão. A implementação detalhada da biblioteca é deixada a cargo dos fabricantes e vendedores individualmente, tornando-os livres para produzirem versões otimizadas para suas máquinas.

Uma versão MPI – 2, também já definida, adiciona características complementares à sua antecessora, incluindo ferramentas para entradas e saídas paralelas, ligações para o C++ e Fortran 90, além de propiciar o gerenciamento de processos dinâmicos.

Fundamentalmente o MPI tem por objetivos prover portabilidade do código fonte (programas MPI devem poder ser compilados e executados, no estado, sobre qualquer plataforma) e permitir implementações eficientes sobre ampla variedade de arquiteturas.

Adicionalmente o MPI também oferece um grande número de recursos, incluindo diferentes tipos de comunicação, rotinas especiais destinadas a operações "coletivas" comuns, e a capacidade de manusear tipos de dados e topologias definidas pelo usuário além de admitir arquiteturas paralelas heterogeneas.

O MPI é recomendado quando se deseja escrever programas paralelos portáveis, alcançar alto desempenho em programação paralela (quando se escreve bibliotecas destinadas a paralelização) e sempre que se quiser manusear um problema que envolva relações de dados irregulares ou dinâmicas que não se encaixam bem no modelo de dados paralelos.

4.1 Características Básicas dos Programas de Passagem de Mensagem

Programas de passagem de mensagem são basicamente constituídos de partes de um programa seqüencial. Devem ser capazes de se comunicar utilizando chamadas a bibliotecas. Essas chamadas podem ser divididas, grosseiramente, em quatro classes:

- Chamadas para inicializar, gerenciar e finalizar comunicações;
- Chamadas para promover comunicação entre pares de processos;
- Chamadas para promover operações de comunicação entre grupos de processos;
- Chamadas para criar tipos de dados arbitrários.

A primeira delas destina-se a disparar a comunicação, identificar o número de processos utilizados, criar subgrupos de processos e identificar qual processo está executando determinada parte do programa.

A segunda, denominada operações de comunicação ponto a ponto, consiste de diferentes tipos de operações de transmissão e recepção.

A terceira consiste de operações coletivas que provêm a troca eficiente de dados entre grupos de processos.

A última delas provê flexibilidade no tratamento de estruturas de dados complexas.

4.2 Comunicação

O padrão MPI estabelece rotinas tanto para comunicação ponto a ponto quanto para coletivas.

A comunicação pode ocorrer de modo a impedir que os processos participantes realizem outras tarefas enquanto ela não for finalizada, neste caso, são ditas *bloqueantes*; caso contrário, isto é, se for permitido aos processos envolvidos numa operação de comunicação que realizem outras tarefas enquanto aguardam sua conclusão, então esta é dita *não bloqueante*.

Existem critérios, denominados *critérios de conclusão*, que definem quando uma operação de comunicação pode ser considerada finalizada.

4.2.1 Modos de Comunicação e Critérios de Conclusão

MPI oferece grande flexibilidade na especificação de como as mensagens são transferidas.

Existe uma variedade de modos de comunicação que definem o procedimento a ser utilizado na troca de uma mensagem bem como o critério para se determinar quando uma comunicação foi concluída.

Para a transmissão são previstos os seguintes modos:

- Padrão
- Síncrono
- Armazenado
- Pronto

O modo padrão, de propósito geral, é o mais utilizado. Apesar dos outros três modos de transmissão previstos serem muito úteis em circunstâncias especiais, nenhum deles proporciona a versatilidade encontrada no modo padrão.

Quando o MPI executa uma transmissão no modo padrão, uma entre duas situações ocorre: ou a mensagem é copiada para dentro de um *buffer* interno e transferida assíncronamente para o processo destinatário ou os processos origem e destino se sincronizam para a transferência da mensagem. A implementação MPI é livre para escolher, caso a caso, qual opção adotar considerando diferentes parâmetros como, por exemplo, tamanho da mensagem, disponibilidade de recursos, etc...

Se a primeira opção é utilizada, então a operação de transmissão é dita formalmente concluída quando a cópia para o *buffer* é realizada. Por outro lado, se os dois processos se sincronizam para a troca de uma mensagem, então a operação de transmissão é dita formalmente concluída apenas após o processo destino acusar o inicio da recepção da mensagem. Este procedimento se aplica tanto para chamadas a rotinas de comunicação bloqueantes quanto para as não bloqueantes. A satisfação do critério de conclusão pode significar tanto que a mensagem foi copiada para dentro de um *buffer* interno quanto que os processos de transmissão e recepção se sincronizaram para a transferência da mensagem.

Uma das vantagens do modo padrão de transmissão é que a escolha entre armazenar ou sincronizar é deixada a cargo de cada implementação, caso a caso, que geralmente possui perspectiva mais nítida sobre os intercâmbios, especialmente se estão envolvidos recursos de baixo nível ou internos ao MPI.

O modo de transmissão síncrono exige que os processos transmissor e receptor sejam

sincronizados pelo MPI.

Uma operação de transmissão em modo síncrono é assumida como finalizada pelo processo origem quando o processo destino inicia a recepção da mensagem, note-se portanto que o processo destino não precisa realizar a recepção completa da mensagem para que o critério de conclusão seja satisfeito, basta apenas tê-la iniciado, para que o processo origem entenda que a comunicação foi concluída.

O modo armazenado exige que sejam reservadas áreas de memória para o armazenamento dos dados.

O aspecto adverso deste modo de transmissão reside no fato de que a responsabilidade de gerenciar o área de armazenamento (*buffer*) tem que, necessariamente, ser assumida pelo programador. Se, em algum momento, não houver *buffer* suficiente para completar uma chamada a uma rotina de transmissão, os resultados são imprevisíveis, mas o MPI também dispõe de rotinas que permitem a alocação de *buffer* para isto.

O modo pronto exige que um recibo (previamente combinado) seja colocado no processo destino antes que ocorra sua chamada, se isto não acontecer, não se pode prever os resultados, portanto, é responsabilidade do programador assegurar tal fato quando este modo de transmissão é utilizado.

Para a recepção é previsto apenas um modo o qual estipula que o fim da comunicação ocorre quando os dados forem recebidos e encontrarem-se disponíveis para uso.

4.2.2 Comunicação Ponto a Ponto

A operação de comunicação elementar do MPI é a ponto a ponto, isto é, comunicação direta entre dois processos onde um deles transmite e o outro recebe. Ela ocorre em duplo sentido, o que significa que tanto uma transmissão explicita quanto uma recepção também explicita são exigidas; os dados não são trocados sem a participação de ambos processos.

Numa operação de comunicação ponto a ponto genérica, uma mensagem, contendo um bloco de dados, é trocada entre dois processos.

Uma mensagem é composta por um *envelope*, que indica os processos fonte e destino, e um *corpo*, que contém os dados que efetivamente se deseja transferir.

MPI utiliza três parâmetros de informação para caracterizar o corpo da mensagem de maneira flexível:

- ***Buffer*** – a posição inicial da memória onde os dados de saída (no caso do processo transmissor) ou de entrada (no caso do processo receptor) deverão ser armazenados.
- ***Datatype*** – o tipo de dado que será trocado.
- ***Count*** – o número de itens do tipo *Datatype* que serão transferidos.

Uma característica importante a ser lembrada é que o MPI padroniza a designação de tipos elementares de dados, assim, não há necessidade de se preocupar com a maneira como as máquinas de um ambiente heterogeneo os representam, eles se adequarão à representação de cada máquina em particular, esta é uma característica muito útil e desejável quando se manipulam números de ponto flutuante em ambientes heterogeneos.

4.2.3 Comunicação Bloqueante e Não Bloqueante

Independente do modo de comunicação utilizado, uma transmissão ou recepção pode ser bloqueante ou não.

Bloqueantes são aquelas que impedem que o processo retorne da chamada de sub rotina (Fortran) ou função (C) de transmissão ou recepção até que a operação seja finalizada. Isto assegura que o critério de conclusão seja satisfeito antes de se permitir que o processo avance. Assim, por exemplo, numa transmissão bloqueante fica assegurado que o processo

transmissor pode sobrescrever as variáveis transmitidas, pois, seguramente, se ele retornou da chamada da rotina de transmissão, então é porque ela foi finalizada. Analogamente, numa recepção bloqueante, assegura-se que se o processo receptor retornou da chamada da rotina de recepção, então os dados já foram recebidos e estão prontos para uso.

Por outro lado, uma transmissão ou recepção não bloqueante retorna imediatamente da chamada, sem nenhuma informação sobre se o critério de conclusão foi satisfeito ou não. Este procedimento apresenta a vantagem de liberar o processo para outras tarefas enquanto a comunicação prossegue, podendo-se verificar mais tarde se a operação foi completada a contento. Assim, por exemplo, uma transmissão síncrona não bloqueante retorna imediatamente da chamada, apesar de não se poder considera-la finalizada até que sua recepção seja acusada. Neste caso, o processo transmissor pode se ocupar de outras tarefas úteis enquanto a comunicação prossegue e proceder a verificação de sua efetiva finalização mais tarde, porém, não se deve sobrescrever os dados que se pretende transmitir até que essa verificação aconteça, pois não se pode assumir, com total segurança, que a mensagem foi recebida a contento.

4.2.4 Comunicações Coletivas

Apesar de a comunicação ponto a ponto ser elementar na especificação MPI, estão previstas também rotinas para comunicações coletivas que permitem que grupos compostos por mais de dois processos se comuniquem de diferentes formas (por exemplo, um para muitos ou muitos para um).

Rotinas de comunicação coletiva apresentam vantagens sobre suas equivalentes ponto a ponto, entre elas pode-se citar:

- significativa redução de possibilidades de erros (uma linha de código – a chamada para a rotina coletiva – normalmente substitui muitas chamadas ponto a ponto);
- o código fonte fica mais compreensível e simplifica significativamente correções, depuração e manutenção;

- formas otimizadas de rotinas coletivas são, freqüentemente, mais rápidas que suas equivalentes ponto a ponto.

Operações de difusão (*broadcast*), redução (*reduction*), espalhamento (*scatter*) e agrupamento (*gather*) são exemplos de operações de comunicação coletiva inclusas nas implementações MPI.

A difusão é o tipo mais simples delas. Nela, um processo transmite uma cópia de alguns de seus dados locais para todos os outros participantes de um determinado grupo de comunicação, conforme ilustrado na figura 9, onde cada coluna representa um processo diferente e cada bloco colorido numa coluna representa o local de uma parte dos dados. Blocos de mesma cor alocados em múltiplos processos contém cópias dos mesmos dados.

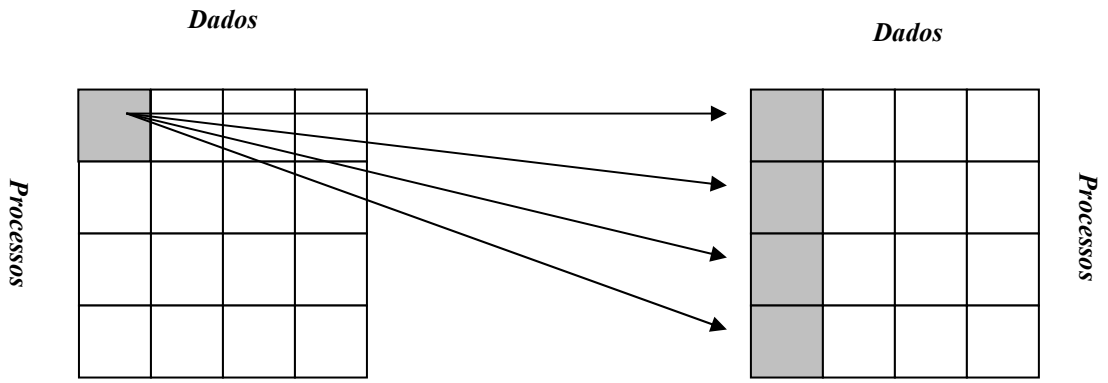


Figura 9 – Operação de Difusão

Talvez as mais importantes classes de operações coletivas sejam aquelas que distribuem dados de um processo para um grupo ou vice versa. Essas operações são denominadas de *agrupamento* e *espalhamento* (*gather* e *scatter*) e encontram-se ilustradas na figura 10.

No espalhamento, todos os dados (um arranjo de algum tipo) são inicialmente agrupados num único processo (o lado esquerdo da figura 10). Após a operação, partes desses dados são distribuídos à diferentes processos (lado direito da figura). As caixas coloridas refletem a possibilidade de que os dados possam não ser uniformemente divididos entre os processos. O agrupamento se comporta de maneira inversa: coleta partes dos dados que

estão distribuídos entre um grupo de processos e os reagrupa na ordem apropriada num único processo.

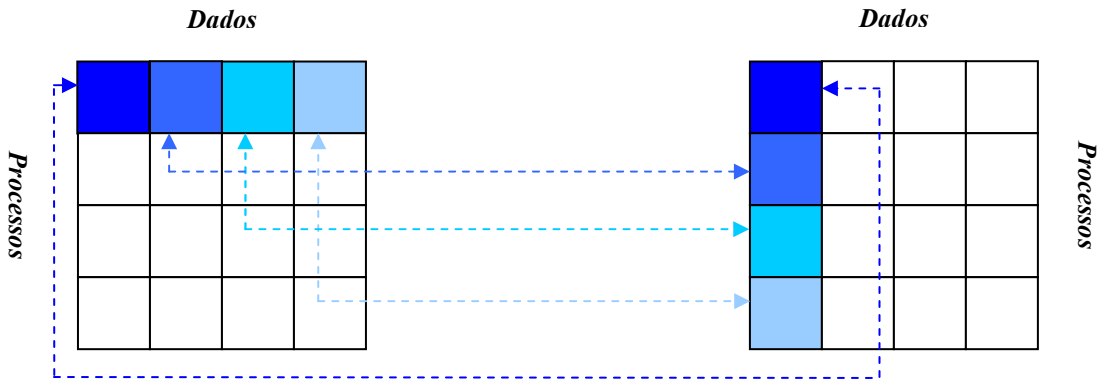


Figura 10 – Operações de Agrupamento e Espalhamento

A redução é uma operação coletiva na qual um único processo, denominado *processo raiz*, coleta dados de outros processos de um grupo e os combina num único item.

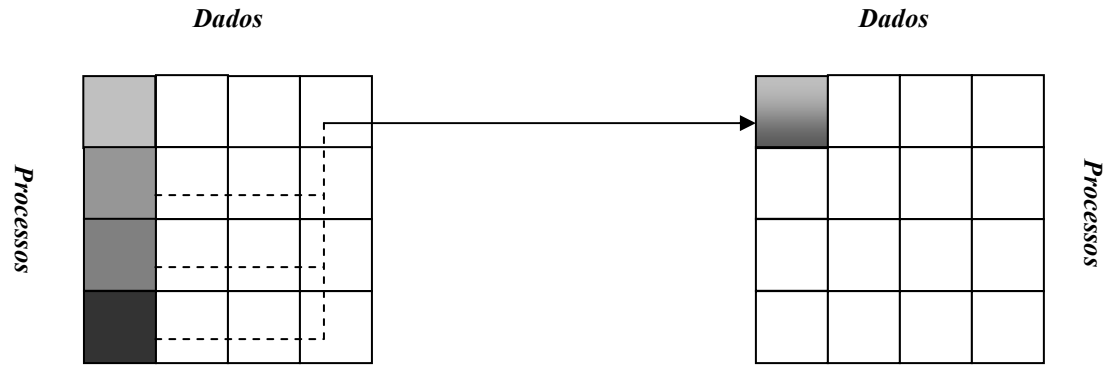


Figura 11 – Operação de Redução

A figura 11 ilustra a operação de redução. Nela, os dados, que podem ser arranjos ou valores escalares, são inicialmente distribuídos entre os processos e após a operação, os dados já reduzidos (arranjos ou escalares) são alocados no processo raiz.

4.3 Compilação e Execução de Programas MPI

O MPI não especifica como os programas são iniciados, assim, cada implementação estabelece um procedimento próprio que deve ser consultado.

Ao se compilar um programa MPI é necessário liga-lo à biblioteca MPI. Normalmente isto se faz pela inclusão da opção – *lmpi no* carregador.

Para se executar um programa MPI é comum se utilizar um "*wrapper*" denominado *mpirun* ou *mpprun*, por exemplo, o comando *mpirun –np 4 a.out* executará o programa *a.out* sobre quatro processos.

Detalhes específicos de cada implementação normalmente se encontram disponibilizados na documentação associada e devem ser sempre consultados.

4.4 Programas MPI

De maneira geral, um programa MPI obedece a seguinte seqüência de procedimentos:

- Inicializa o ambiente para a troca de mensagens;
- Comunica dados entre processos e
- Finaliza o ambiente de troca de mensagens.

MPI dispõe de um extenso conjunto de funções (em C) ou sub rotinas (no caso do Fortran), porém, seis delas são básicas e normalmente utilizadas na maioria dos programas:

- ***MPI_Init*** – Inicializa o ambiente MPI;
- ***MPI_Comm_size*** – retorna o número de processos envolvidos;

- ***MPI_Comm_rank*** – retorna o índice identificador do processo;
- ***MPI_Send*** – envia uma mensagem;
- ***MPI_Recv*** – recebe uma mensagem e
- ***MPI_Finalize*** – finaliza o ambiente MPI

Pode-se implementar uma variedade programas MPI utilizando apenas essas seis rotinas.

Rotinas que não são MPI, são locais e, portanto, são executadas localmente em cada processo.

A estrutura geral de um programa MPI obedece ao seguinte roteiro:

- Inclusão do arquivo cabeçalho MPI;
- Declaração de variáveis;
- Inicialização do ambiente MPI;
- Computação e chamadas de comunicação MPI;
- Encerramento das comunicações MPI.

O arquivo cabeçalho contém definições específicas e protótipos de funções MPI.

Após a declaração de variáveis, cada processo chama uma rotina MPI que inicializa o ambiente de passagem de mensagem. Todas as chamadas de rotinas de comunicação MPI devem, necessariamente, ocorrer somente após esta inicialização.

Finalmente, antes da finalização do programa, cada processo deve chamar uma rotina para encerrar o MPI, após o que, nenhuma rotina MPI pode ser chamada. Se algum processo não atinge este ponto durante a execução, então o programa aparentará estar suspenso.

4.4.1 Arquivos Cabeçalho

Os arquivos cabeçalho contém os protótipos de funções (em C) ou sub rotinas (em Fortran), as definições das macros, constantes especiais e tipos de dados específicos utilizados pelo MPI. Uma declaração "*include*" apropriada deve constar no código fonte.

4.4.2 Convenções de Nomes

Nomes das rotinas, constantes, tipos, etc... ligados ao MPI devem começar com "**MPI_**", por exemplo:

MPI_Init (&argc, &argv)

Os nomes de constantes MPI são sempre escritos com caracteres maiúsculos, por exemplo:

MPI_COMM_WORLD

MPI_REAL

MPI dispõe de tipos definidos na própria especificação que seguem a convenção estabelecida na linguagem C, por exemplo:

MPI_Comm é o tipo correspondente a um "comunicador" MPI.

4.4.3 Rotinas MPI e Retorno de Valores

Rotinas MPI são implementadas como funções em C e como sub rotinas em Fortran. Ambos casos normalmente prevêem o retorno de um código de erro que possibilita verificar se a rotina foi bem sucedida.

Em C as funções MPI retornam um valor inteiro que indica o status de saída da chamada, por exemplo:

int err;

...

err = MPI_Init (&argc, argv);

...

Se a rotina foi executada com êxito, então o código de erro retornado é *MPI_SUCCESS*, isto é: o inteiro retornado é a constante inteira pré definida MPI_SUCCESS que pode então ser verificada.

4.4.4 Manipuladores MPI

Manipuladores são o meio através dos quais estruturas de dados próprias, definidas e mantidas pelo MPI, relacionadas à comunicação e outros eventos são referenciadas. Esses manipuladores são retornados por várias chamadas MPI e podem ser utilizados como argumentos em outras.

Em C eles são ponteiros de tipos de dados especialmente definidos (criados através do mecanismo *typedef*).

São exemplos de manipuladores MPI:

MPI_SUCCESS

MPI_COMM_WORLD

4.4.5 Tipos de Dados MPI

MPI dispõe de seus próprios tipos de dados correspondentes a diversos tipos elementares de C ou Fortran. Normalmente as variáveis são declaradas como tipos de C ou Fortran.

Nomes de tipos de dados MPI são utilizados como argumentos nas rotinas sempre que necessário.

MPI esconde detalhes, por exemplo, da representação de ponto flutuante, o que se constitui numa vantagem para o implementador, pois permite a tradução automática entre diferentes representações dispostas em ambientes heterogeneos. Como regra geral, quando se utiliza MPI, um tipo de dado recebido normalmente é compatível com aquele especificado na transmissão, não importando se a representação utilizada nos processos transmissor e receptor são diferentes ou não.

MPI também permite que o programador defina tipos de dados arbitrários construídos a partir dos tipos básicos.

O quadro 1 subseqüente apresenta tipos básicos de dados MPI e seus respectivos correspondentes em C:

MPI	***C***
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Quadro.1 – Tipos de Dados MPI e seus Correspondentes em C
Fonte: Elaborado pelo autor

Existem ainda tipos de dados especiais previstos para o MPI, por exemplo:

- *MPI_Comm* – um comunicador (definido na seqüência);
- *MPI_Status* – uma estrutura contendo várias partes da informação de status de chamada MPI;
- *MPI_Datatype*.

Eles são utilizados nas declarações de variáveis, por exemplo:

MPI_Comm algum_comm;

...declara uma variável denominada *algum_comm* do tipo *MPI_Comm* (um comunicador).

4.4.6 Inicialização do MPI

A primeira rotina MPI a ser chamada num programa tem, necessariamente, que ser a de inicialização: *MPI_INIT*. Esta rotina estabelece o ambiente MPI e retorna um código de erro se ocorrer algum problema. Ela só pode ser chamada uma única vez no programa.

Exemplo:

```
int err;
...
err = MPI_Init (&argc, &argv);
...
```

4.4.7 Comunicadores

Um comunicador é um manipulador que representa um grupo de processos que podem se comunicar uns com os outros.

Toda operação de comunicação precisa de um nome de comunicador como argumento que deve ser o mesmo empregado tanto na chamada de transmissão quanto na de recepção associada para que a comunicação ocorra.

Podem existir muitos comunicadores num programa e um determinado processo pode participar de diferentes comunicadores. Dentro de cada comunicador, os processos são numerados consecutivamente, a partir de zero. Este número de identificação é denominado *rank* (posto) do processo naquele comunicador. O posto é utilizado também para especificar a fonte e o destino em chamadas de transmissão e recepção. Se um processo participa de mais de um comunicador, seu posto em cada um deles pode (e normalmente é) ser diferente.

MPI fornece automaticamente um comunicador básico, denominado MPI_COMM_WORLD, que congrega todos os processos. Sua utilização permite a comunicação entre todos os processos envolvidos. Pode-se também definir outros comunicadores que agrupem subconjuntos específicos dos processos disponíveis.

Um processo pode determinar seu posto (*rank*) num comunicador através da chamada *MPI_COMM_RANK*, por exemplo, em C:

*int MPI_Comm_rank (MPI_Comm comm, int *rank);*

Size é o número de processos que participam de um comunicador e pode ser obtido através da chamada, em C:

*int MPI_Comm_size (MPI_Comm comm, int *size);*

4.4.8 Finalização do MPI

A última rotina MPI a ser chamada em um programa deve, necessariamente, ser *MPI_FINALIZE*, a qual libera os recursos utilizados por todas as estruturas MPI, cancela operações que nunca terminaram, etc...

Deve ser chamada por todos os processos participantes, sob pena de se ter o processamento suspenso se algum deles não o fizer.

Uma vez invocada, nenhuma outra rotina MPI pode ser chamada, inclusive a de inicialização.

A título de exemplo, em C:

```
int err;
...
err = MPI_Finalize ( );
```

4.4.9 Exemplo de Programa MPI

O exemplo a seguir (PACHECO, 1998) ilustra o emprego de rotinas comuns do padrão MPI:

```
#include <stdio.h>
#include <mpi.h>

main (int argc, char ** argv)
{
        int             rank;                   /* Rank do Processo */
```

```
int         size;           /* Número de Processos */
int         origem;         /* Rank do Processo Remetente */
int         dest;           /* Rank do Processo Destinatário */
int         rotulo;         /* Rótulo Para as Mensagens */
char        mensagem[100];  /* Arranjo de 100 Caracteres */
MPI_Status  status;         /* Retorna o Status do Receptor */

/* Inicialização do MPI */

MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);

If (rank==0)
        {
        for (origem = 1; origem < size; origem++)
                {
                MPI_Recv (mensagem, 100, MPI_CHAR, origem, rotulo,
                MPI_COMM_WORLD, &status);
                Printf ("%s\n", mensagem);
                }
        }
else
        {
        sprintf (mensagem, "Congratulações do Processo %!d\n", rank);
```

```
            dest=0;
            MPI_Send (mensagem, strlen(mensagem)+1, MPI_CHAR, dest,
            rotulo, MPI_COMM_WORLD);
            }
        MPI_Finalize ( );
        Return ( ):
}
```

Executado sobre quatro processos, o programa acima produzirá a seguinte saída:

```
Congratulações do Processo 1!
Congratulações do Processo 2!
Congratulações do Processo 3!
```

Alguns elementos básicos, anteriormente expostos, de um programa MPI estão presentes no exemplo:

- Funções (ou sub rotinas, no caso do Fortran) MPI possuem nomes que, necessariamente, começam com ***MPI_***;
- O arquivo de cabeçalho ***mpi.h***, que contém definições e funções protótipos a serem importadas pelo programa através da diretiva de inclusão *#include <mpi.h>;*

Cada processo executa uma cópia do programa, exceto o processo 0, que apenas transmite o comando aos outros.

4.4.10 Rotinas MPI

Bibliotecas MPI dispõem de dezenas de rotinas úteis, algumas delas, fundamentais, empregadas em praticamente todos os programas que fazem uso do padrão, outras, especificas, são utilizadas em casos particulares onde se mostram mais eficientes e evitam grande parte do trabalho de programação.

Algumas delas, mais usuais, pertencentes portanto ao primeiro grupo, são resumidamente expostas na sequência.

Antes de prosseguir é necessário esclarecer que apesar do termo "*rotinas*" ser empregado indiscriminadamente tanto para designar sub rotinas (no caso do Fortran) quanto funções (no caso de C), a exposição subsequente trata, na realidade, destas últimas – funções – disponíveis na biblioteca *mpi.h* da linguagem C de programação. Isto ocorre porque esta foi a linguagem empregada no desenvolvimento do trabalho prático associado a este texto e, portanto, é bastante lógico que ela também seja utilizada nesta explanação. No entanto, é bom lembrar que estas sempre encontram suas correspondentes em Fortran.

MPI_Init (&argc, &argv)

> Rotina de inicialização do MPI, sempre deve estar presente e tem que, necessariamente, ser chamada um única vez e antes de qualquer outra rotina especifica do MPI que conste no programa.

MPI_Comm_rank

Sintaxe:

*int MPI_Comm_rank (MPI_Comm Comm, int *rank)*

Comportamento:

Atribui à posição apontada por *rank* o posto do processo corrente no comunicador *Comm*.

MPI_Comm_size

Sintaxe:

*int MPI_Comm_size (MPI_Comm Comm, int *size)*

Comportamento:

Atribui à posição apontada por *size* o número de processos envolvidos no comunicador *Comm*.

MPI_Send

Sintaxe:

*int MPI_Send (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm Comm)*

Comportamento:

Transmite o conteúdo da posição apontada por *buf* para o processo cujo posto é *dest* no comunicador *Comm*.

count indica o número de dados e *dtype* o tipo de dados MPI que serão transmitidos.

tag é o rótulo da mensagem.

MPI_Recv

Sintaxe:

*int MPI_Send (void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm Comm, MPI_Status *status)*

Comportamento:

Recebe uma mensagem contendo no máximo *count* itens do tipo *dtype* e a armazena na área de memória cuja posição inicial é apontada por *buf*.

source é o rank do processo remetente no comunicador *Comm*.

tag é o rótulo da mensagem.

Retorna o status da recepção na posição indicada por *status*.

MPI_Barrier

Sintaxe:

int MPI_Barrier (Comm)

Comportamento:

Bloqueia o processo que a chamou até que todos os processos do grupo envolvido pelo comunicador *Comm* chame a rotina, promovendo o sincronismo entre todos.

MPI_Bcast

Sintaxe:

*int MPI_Bcast (void *buf, int count, MPI_Datatype dtype, int rank, int tag, MPI_Comm Comm)*

Comportamento:

Transmite o conteúdo da posição apontada por *buf*, para todos os processos participantes do comunicador *Comm*.

rank é o posto do processo raiz que originou a operação de difusão no comunicador *Comm*.

count indica o número de dados e *dtype* o tipo de dados MPI que serão transmitidos.

tag é o rótulo da mensagem.

MPI_Reduce

Sintaxe:

*int MPI_Reduce (void *send_buf, void *recv_buf, int count, MPI_Datatype dtype, MPI_Op operation, int rank, MPI_Comm Comm)*

Comportamento:

Combina os elementos contidos na memória cuja posição inicial é apontada por *send_buf*, aplica a operação especificada por *operation* e devolve o resultado para o processo raiz cujo posto no comunicador *Comm* é *rank*, armazenando-o na memória cuja posição inicial é apontada por *recv_buf*.

MPI_Gather

Sintaxe:

*int MPI_Gather (void *send_buf, int send_count, MPI_Datatype send_type, void *recv_buf, int recv_count, MPI_Datatype recv_type, int rank, MPI_Comm Comm)*

Comportamento:

Cada processo, inclusive o que originou a operação de agrupamento (*processo raiz*), envia o conteúdo da área de memória cuja posição inicial é apontada por *send_buf* para o processo raiz, o qual recebe as mensagens e as armazena, ordenadas segundo o posto de cada processo no comunicador *Comm*, na área de memória cuja posição inicial é apontada por *recv_buf*.

send_count e *recv_count* indicam a quantidade de dados dos tipos *send_type* e *recv_type* respectivamente, que estão sendo trocados.

MPI_Scatter

Sintaxe:

*int MPI_Scatter (void *send_buf, int send_count, MPI_Datatype send_type, void *recv_buf, int recv_count, MPI_Datatype recv_type, int rank, MPI_Comm Comm)*

Comportamento:

O processo que originou a operação de espalhamento (*processo raiz*), envia o conteúdo da área de memória cuja posição inicial é apontada por *send_buf* para cada processo, participante do comunicador *Comm*, ordenadamente segundo seus postos, os quais recebem as mensagens e, cada qual, as armazena, na área de memória cuja posição inicial é apontada por *recv_buf*.

send_count e *recv_count* indicam a quantidade de dados dos tipos *send_type* e *recv_type* respectivamente, que estão sendo trocados.

MPI_Abort

Sintaxe:

int MPI_Abort (Comm)

Comportamento:

Termina a execução no ambiente MPI encerrando todos os processos envolvidos pelo comunicador *Comm*.

MPI_Finalize

Sintaxe:

MPI_Finalize (int codigo de erro)

Comportamento:

Finaliza o MPI. Se, por alguma razão a finalização não se efetivou, então retorna o inteiro *codigo de erro*. Após ela ser invocada, nenhuma outra rotina MPI pode ser chamada, nem mesmo a de inicialização.

Dezenas de outras rotinas úteis são oferecidas no MPI e poder-se-ia escrever um trabalho inteiro sobre elas e suas aplicações, descrevendo pormenorizadamente seus empregos, porém este não é o objetivo aqui estipulado, assim, o conjunto acima exposto reflete um pequeno, mas significativo, quadro de rotinas usuais que propiciam solução a grande parte de problemas que utilizam MPI, portanto, seu conhecimento, mesmo que superficial, é fundamental para se aplicar MPI.

5 MECANISMO GENÉRICO PARA PARALELIZAÇÃO E DISTRIBUIÇÃO

O mecanismo genérico aqui proposto obedece a estrutura ilustrada na figura 12:

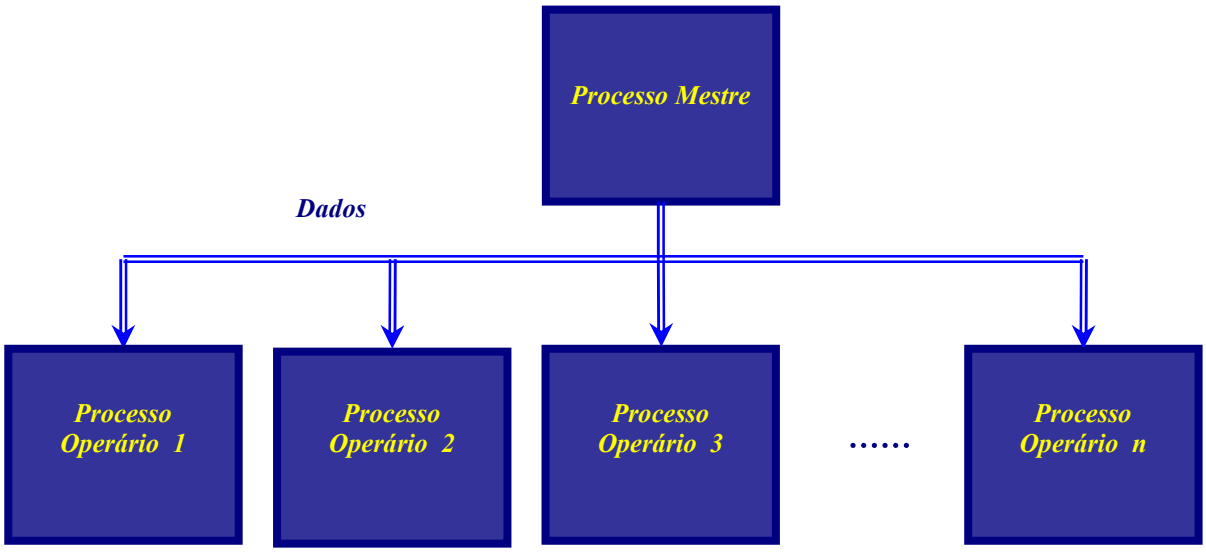


Figura 12 – Estrutura Geral de Distribuição

Nela, um processo designado *Mestre* se ocupa de distribuir dados a *n* processos denominados *Operários*, cada qual capaz de operar, de maneira autônoma e independente, sobre os dados a ele designados e devolver ao processo *Mestre* as respostas associadas. Esse mecanismo se repete tantas vezes quanto forem necessárias até que se obtenha a solução do problema global originalmente apresentado ao sistema assim configurado.

Dois modelos de programas MPI são previstos para contemplar um mecanismo assim definido: um para o processo *Mestre* e outro para os processos *Operários*.

A estrutura geral desses programas obedece aos fluxogramas expostos na seqüência:

- ***Processo Mestre***

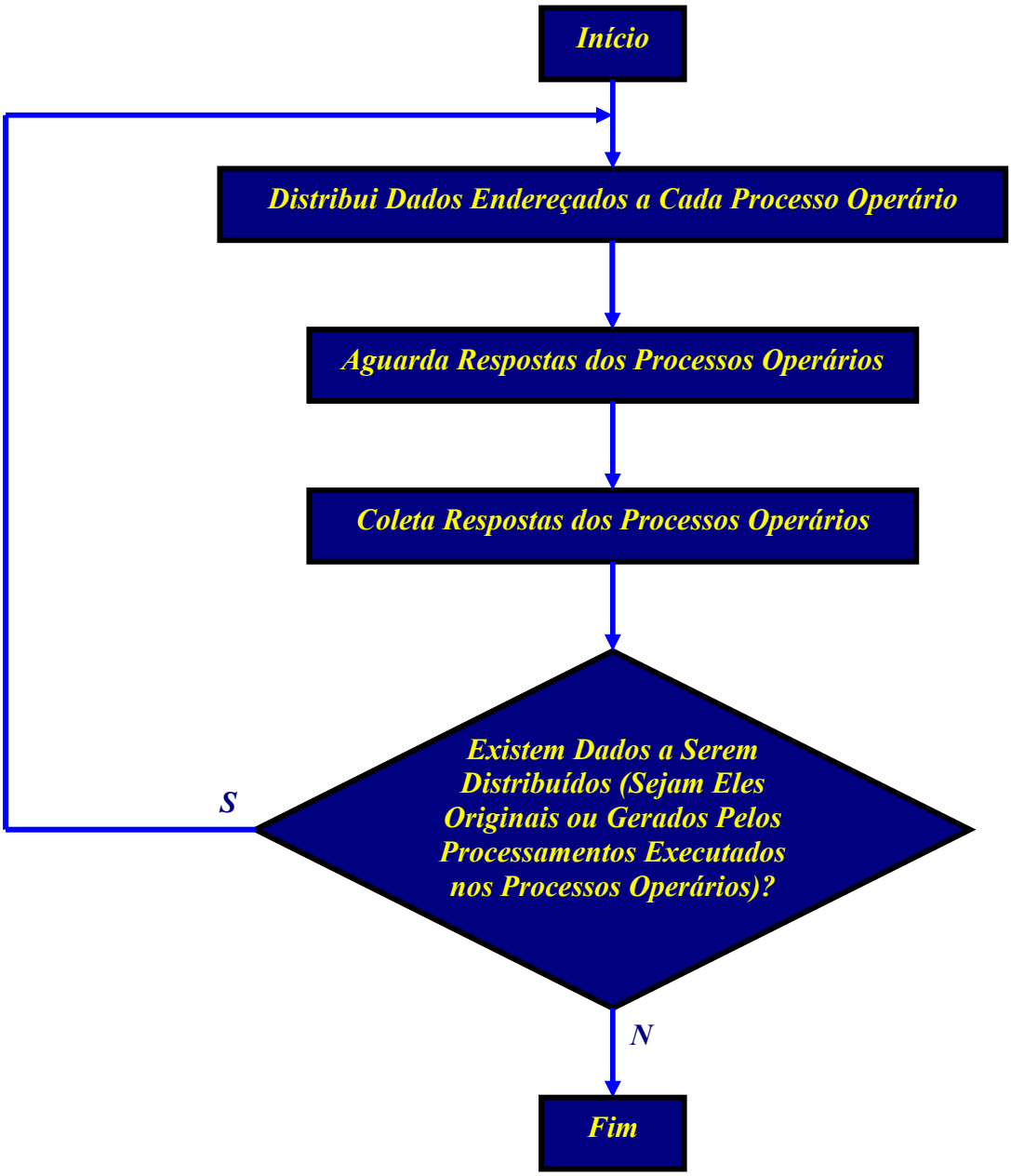


Figura 13 – Fluxograma Geral do Processo Mestre

- ***Processos Operários***

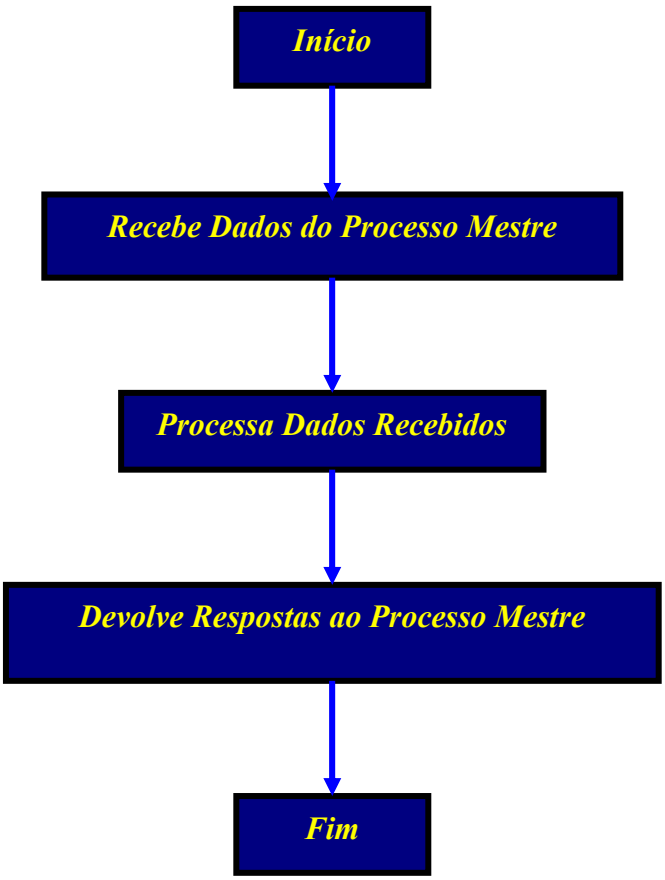


Figura 14 – Fluxograma Geral dos Processos Operário

Um mecanismo assim permite incluir ou excluir facilmente processos *Operários* sempre que necessário bem como realizar a distribuição dos dados segundo os requisitos específicos de cada aplicação.

6 EXEMPLO DE APLICAÇÃO

O trabalho aqui apresentado foi motivado pela realização de um projeto proposto à Secretaria de Ciência, Tecnologia e Desenvolvimento Econômico do Estado de São Paulo e desenvolvido ao longo dos anos de 2002 e 2003 no Instituto de Pesquisas Tecnológicas do Estado de São Paulo – IPT, denominado "*Identificação do Código de Licença de Veículos Automotores Utilizando Visão Computacional*".

A idéia fundamental estabelecida para o projeto era a composição de uma unidade que, instalada a beira de uma via pública, fosse capaz de detectar automaticamente a intrusão de veículos em uma determinada área trafegando num sentido de interesse, capturar uma seqüência de imagens que registrasse o deslocamento do objeto móvel ao longo dessa área e processar essas imagens com a finalidade de obter a identificação do automóvel mediante a determinação de seu código de licença contido na respectiva placa.

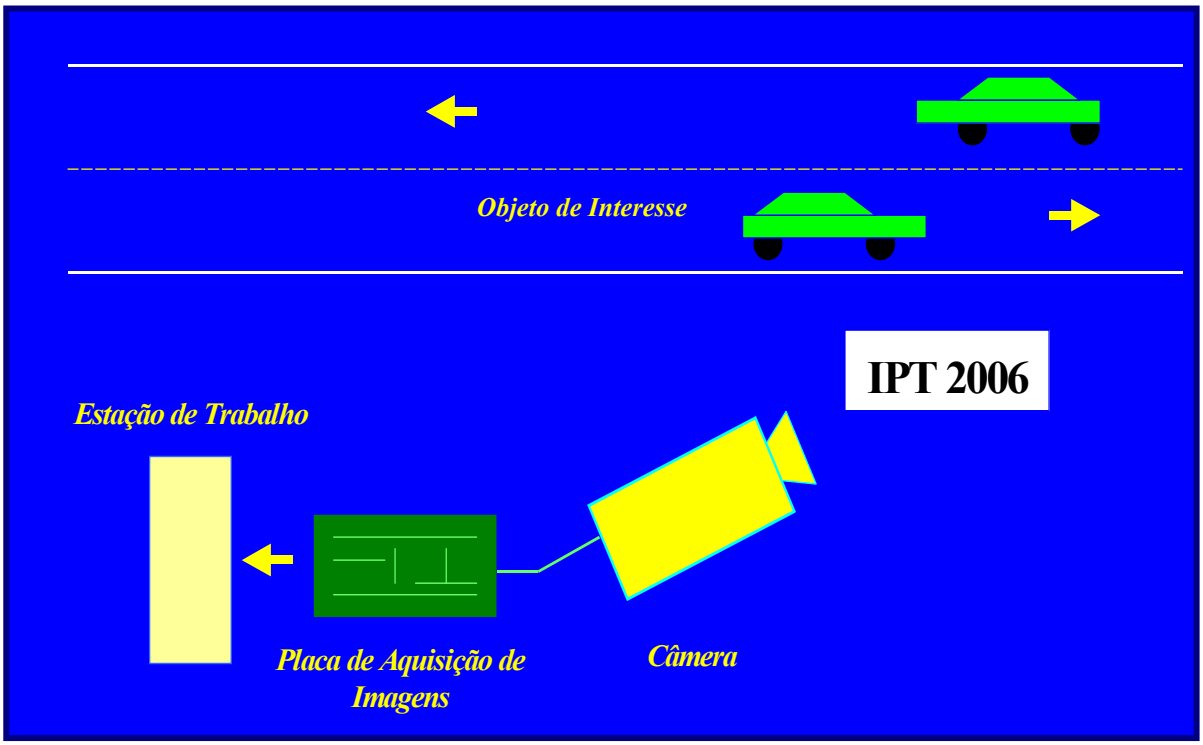


Figura 15 – Ilustração do Exemplo de Aplicação

Inicialmente foram desenvolvidas ferramentas de software capazes de detectar a intrusão do veículo na cena trafegando no sentido de interesse e promover a coleta e o armazenamento da seqüência de imagens que registra o deslocamento do automóvel (dados). Essas ferramentas terminaram por gerar programas que atenderam a contento tais requisitos e, portanto, esta fase pode ser considerada solucionada ou, pelo menos, satisfatoriamente bem encaminhada, conforme pode ser constatado em Martins (2003).

Uma segunda fase tratou do desenvolvimento de programas capazes de processar os dados coletados (seqüências de imagens capturadas) lançando mão de técnicas de visão computacional com o objetivo de obter a identificação do veículo sob as mais diversas condições, que englobam tanto variações no ambiente quanto no objeto alvo - o veículo – tais como luminosidade, posicionamento e velocidade de deslocamento do automóvel, condições climáticas, etc...

Resumidamente, a aplicação em evidência dispõe de uma câmera de vídeo que, posicionada à beira da via de trafego, é capaz de monitorar uma área de interesse (doravante denominada *cena*) e, atuando em conjunto com software específico, ao detectar a presença de um provável veículo invadindo a cena e trafegando no sentido de interesse, inicia um procedimento de coleta de uma seqüência de imagens (quadros, aqui denominados *dados*) que deverão conter o registro do deslocamento do automóvel ao longo da via. Essas imagens são então submetidas a uma série de processamentos que têm por objetivos localizar regiões da cena que possam conter a placa de licença do automóvel, segmentar o código alfanumérico nela contido e, finalmente, promover a identificação do veículo.

No atual estágio de desenvolvimento as operações de detecção da presença de automóveis na cena e a subseqüente coleta das imagens associadas encontram-se muito bem definidas e operam satisfatoriamente. Entretanto, etapas posteriores, que envolvem o processamento dessas imagens, podem e devem ser melhoradas, fato que motivou este trabalho.

Considerando apenas aspectos relacionados ao *software*, após a coleta das imagens, o reconhecimento automático das placas de licença de automóveis envolve basicamente três etapas: a localização das possíveis áreas da cena que podem conter a placa de licença, a

segmentação do código de licença provavelmente contido nessas áreas e, finalmente, o reconhecimento dos caracteres segmentados obtidos na etapa anterior.

Para efeito deste trabalho o conjunto de operações citado acima compõem o *processo global* e cada uma das etapas são *fases do processo global*, assim, sob o ponto de vista de *software*, o *processo global* é composto por 3 *fases*. As duas primeiras encerram procedimentos típicos de visão computacional e, portanto, trazem naturalmente consigo um significativo volume de paralelismo inerente conforme salientam Romig e Samal (1995).

O termo "*processamento de imagem*" é normalmente empregado para denotar a geração de imagens que, partindo de outras, tem por objetivos fundamentais a remoção de ruídos e defeitos variados, o aumento do contraste, a focalização, a correção de efeitos causados pela movimentação da câmera e o aprimoramento da estrutura (organização) e das cores. Entretanto, no âmbito de visão computacional, tal qual é aqui empregado, "*tratamento*" ou "*processamento de imagens*" se refere a funções que têm por finalidade produzir objetos característicos a partir de imagens capturadas, realizando tipicamente, entre outras, tarefas de separação da imagem original em diferentes partes, identificação de objetos característicos, construção da representação tri dimensional de objetos, obtenção da representação de objetos baseada em linhas, estimação de campos de velocidade a partir de seqüências de imagens e a medição de objetos contidos numa imagem.

A última fase do processo global é realizada pela submissão dos resultados obtidos na fase anterior a uma rede neural cuja finalidade é, a partir do código de licença segmentado, obter a identificação do automóvel.

Apesar de exemplos encontrados na literatura especializada de distribuição e paralelização de processos que envolvem redes neurais (ENDLER; GUBITOSO; KOCK; SONG, 1996; FIALLOS; MELCIADES; PIMENTEL, 1999), este não será assunto aqui tratado. A matéria em discussão neste trabalho se limita às duas primeiras fases do processo global anteriormente citadas. Conforme dito por Martins, Lima, Quiróz, Bruna e Almeida (2004), aplicações semelhantes desenvolvidas ao redor do mundo nos últimos anos têm se tornado comercialmente viáveis, porém, há que se destacar que a automação do processo de reconhecimento de placas de licença (LPR – License Plate Recognition) na maioria desses

sistemas ocorre “à custa de considerável simplificação da cena observada, prática, aliás, bastante recomendada” sempre que se trabalha com visão computacional. Nesses casos, as principais restrições observadas “se referem à instalação do sistema junto a estações limitadoras de velocidade, à utilização de sensores independentes para registrar a presença do veículo, à orientação quase normal do plano da placa de licença em relação ao ângulo de visada da câmera e à utilização de lentes com grande fator de amplificação geométrica” (MARTINS; LIMA; QUIRÓZ; BRUNA; ALMEIDA, 2004] Todas essas precauções tornam “as tarefas de reconhecimento bem menos complexas, já que as variações de luminância são menos intensas, não há que se lidar com veículos trafegando em direções opostas e as sub imagens das placas passam a abranger considerável parcela da imagem”. Há que se ressaltar que já se encontram disponíveis comercialmente sistemas que independem dessas restrições, como aquele utilizado pela Prefeitura Municipal de São Paulo para a identificação de automóveis infratores do sistema de rodízio municipal que operam expostos ao ambiente e têm obtido resultados bastante satisfatórios. Este fato no entanto não anula a aplicação do presente trabalho.

As condições acima expostas, embora extremamente desejáveis, nem sempre podem ser alcançadas, principalmente nas circunstâncias a que esta aplicação se propõe atuar: ambientes abertos, desprovidos de controle sobre a iluminação, o posicionamento, a velocidade e o sentido de deslocamento do objeto alvo, assim, é esperado que o sistema absorva diferentes combinações de variações desses parâmetros decorrentes das circunstâncias a que a cena foi submetida no momento da aquisição das imagens e promova tratamento igualmente eficiente para todas elas.

Sob condições bem controladas, é possível obter respostas adequadas do sistema mediante a aplicação de um único método ou ainda de um conjunto reduzido de métodos que absorvam pequenas variações, previamente determinadas e admitidas, de alguns desses parâmetros; este não é o caso aqui tratado.

A aplicação aqui discutida foi originalmente concebida para atuar sob circunstâncias em que não se pode exercer nenhum tipo de controle sobre variáveis importantes para o processo tais como iluminação ambiente, posição da placa de licença em relação ao plano de visada da câmara, posicionamento e velocidade do automóvel, etc... Esta é a razão pela

qual não se pode estabelecer com precisão um método único capaz de tratar igualmente imagens coletadas sob as mais diferentes condições e obter êxito. Para contornar esta situação decidiu-se por adotar um conjunto de métodos potencialmente capazes de, independentemente, tratar as imagens capturadas sob as mais diversas possibilidades de composição das variáveis importantes envolvidas e que fossem bem sucedidos na maioria dos casos.

As duas primeiras fases do processo global passaram então a serem compostas por métodos seqüencialmente encadeados, todos independentes e aptos a manipular as imagens a eles entregues e potencialmente capazes de oferecer uma solução à questão que lhes é destinada.

Visando facilitar a compreensão da exposição que se segue e unificar o vocabulário empregado ao longo do restante do texto, os métodos empregados na solução de cada uma das fases do processo aqui tratadas serão doravante denominados *tarefas*.

A estrutura atualmente em uso na aplicação e que abriga o processo aqui tratado pode ser vista na ilustração da figura 16.

Algumas peculiaridades do processo global devem ser ressaltadas antes de se prosseguir com a explanação para que não se comprometa a compreensão do exposto adiante.

A primeira fase do processo opera sobre o conjunto de imagens originalmente coletado e associado a um único evento (somente um automóvel deve ser o alvo da aplicação) segundo regras estabelecidas pela visão computacional que podem ser observadas em Martins (2003). Essas operações podem resultar em uma única sub imagem da cena capaz de conter a placa de licença do automóvel em foco, em um conjunto delas ou ainda pode não apresentar resultado algum. Nos dois primeiros casos diz-se que esta fase do processo obteve êxito e, portanto ocorre a continuidade. A terceira hipótese determina a ineficácia da fase em questão e, consequentemente, conduz ao insucesso do processo global.

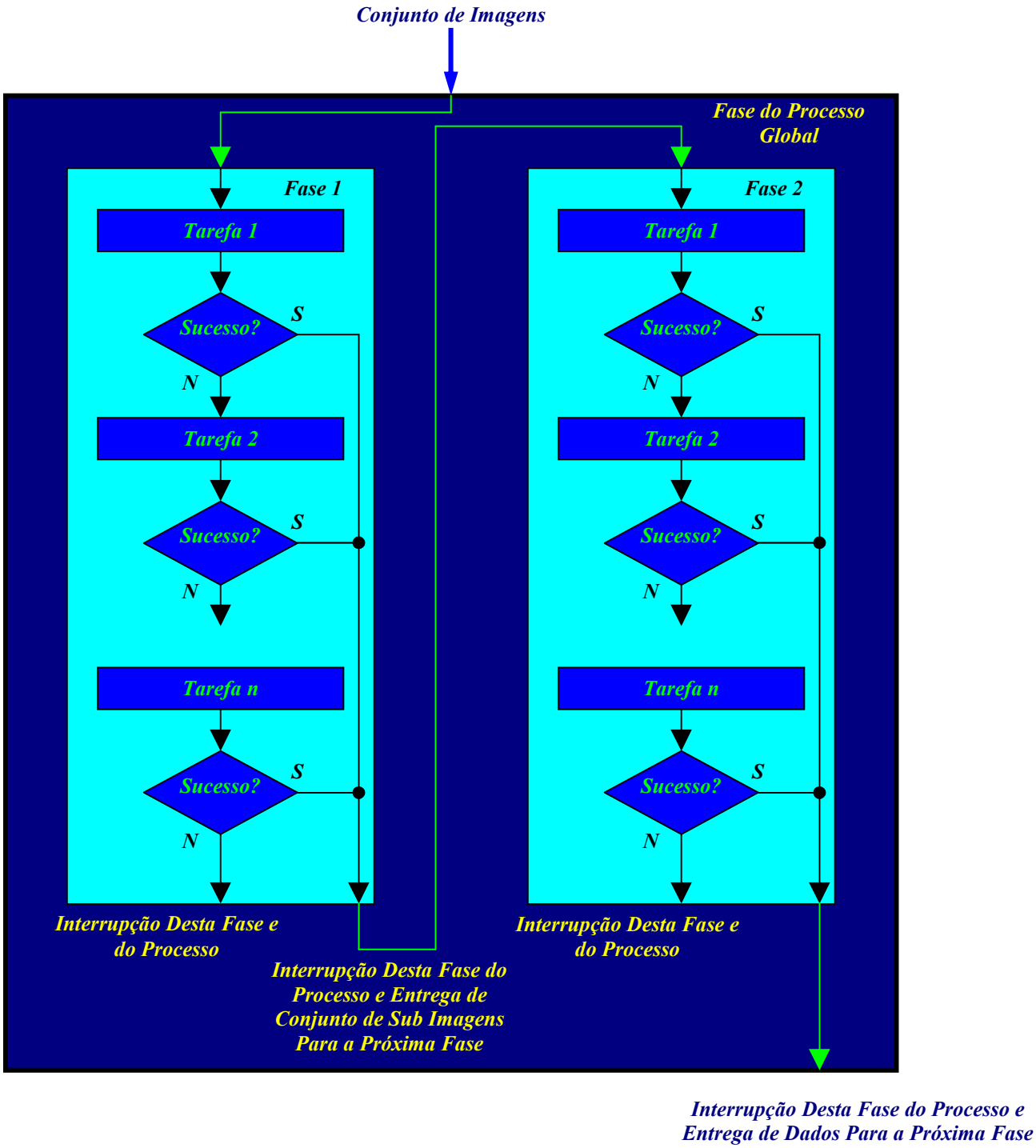


Figura 16 – Estrutura Seqüencial

Se apenas uma sub imagem resulta da primeira fase, então a segunda se comporta de maneira análoga à primeira, isto é, esta sub imagem irá ser submetida à sucessão de tarefas que integram a segunda fase até que um resultado seja obtido determinando, desta forma, o encerramento desta fase e a transferência dos dados por ela gerados (o código segmentado) para a próxima fase do processo global. No entanto, se mais de uma sub imagem é gerada pela primeira fase, então elas são também encadeadas e transferidas para a próxima fase de maneira similar a uma linha de montagem (*pipeline*), onde, assim que uma tarefa repassa a sub imagem por ela tratada para sua subseqüente, significando que esta primeira não obteve êxito, uma nova sub imagem da seqüência lhe é entregue para que seja por ela manipulada. Este procedimento continua até que alguma tarefa obtenha sucesso sobre alguma sub imagem ou até que todas sub imagens percorram toda a cadeia de tarefas e nenhum resultado seja alcançado. Na primeira hipótese, esta fase do processo global é imediatamente suprimida e os dados gerados (código segmentado) são transferidos para a próxima fase onde serão devidamente tratados. Isto ocorre porque se a placa de licença do veículo estiver contida em alguma das sub imagens que compõem o conjunto, então ela estará em apenas uma delas, portanto, uma vez encontrada, não há necessidade de se processar as sub imagens restantes. Caso ocorra a segunda possibilidade, então a aplicação é interrompida pois não houve sucesso nesta fase de operação.

Toda a exposição anterior sugere que a paralelização dessas duas fases pode melhorar o desempenho global da aplicação e, portanto, merece ser cogitada.

Considerando apenas a primeira fase, a coleção de imagens originalmente coletadas pode ser apresentada a uma estrutura paralela similar àquela apresentada na figura 17 sem prejuízo de funcionalidade e auferindo ganho no desempenho, mesmo quando não se obtém sucesso, na maioria das vezes.

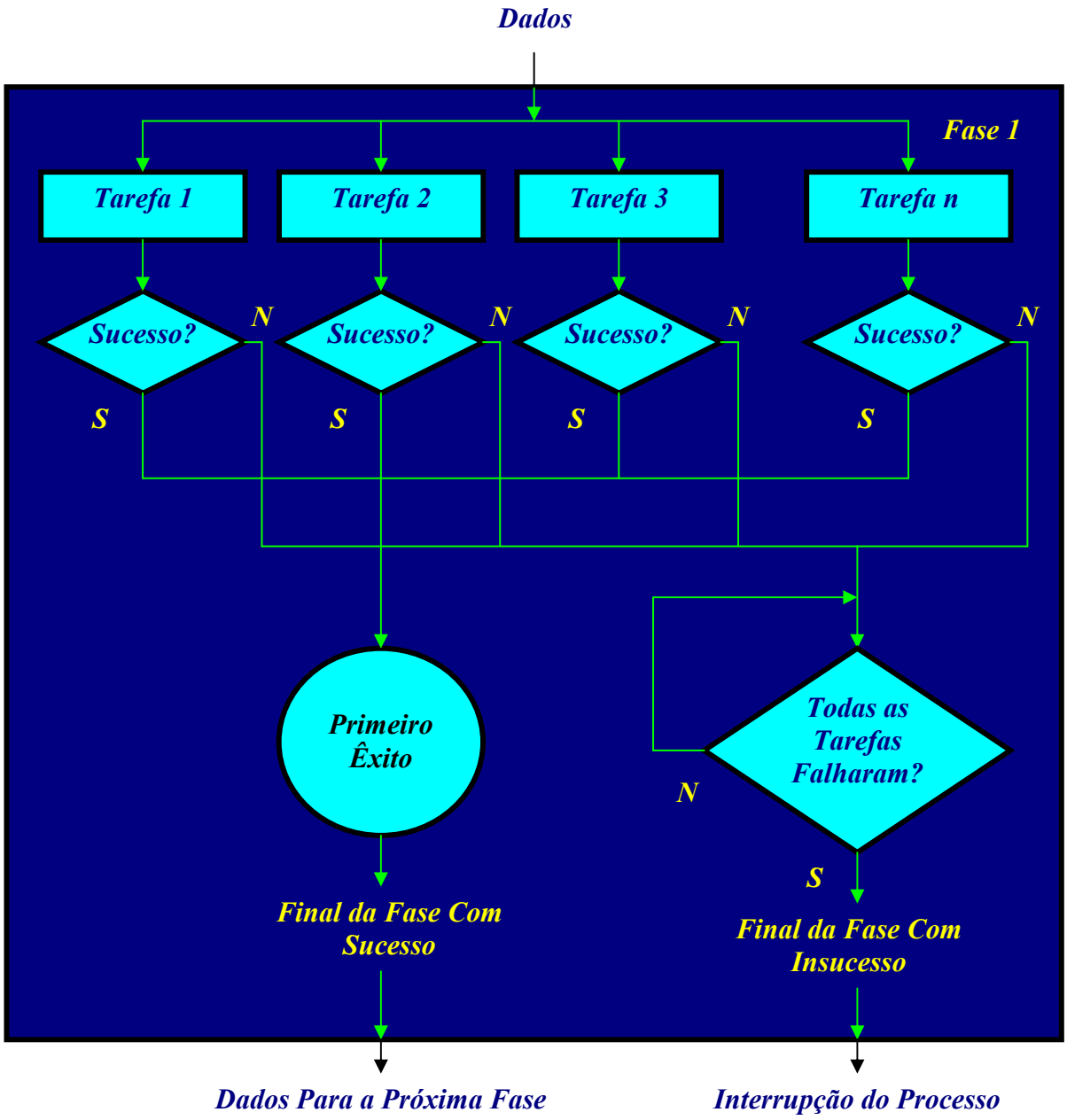


Figura 17 – Estrutura Paralela – Modelo MPSD

A estrutura ilustrada encaixa-se à primeira fase do processo global e revela um modelo MPSD (*Multiple Program Simple Data*), onde o mesmo conjunto de imagens (dados) é apresentado simultaneamente a várias tarefas (programas) distintas, todas capazes de,

independentemente, operar sobre o conjunto de dados e fornecer respostas, sejam elas positivas ou não.

Admitindo como objetivo primordial o desempenho, então a primeira das tarefas que obtiver sucesso, apresenta seu resultado à próxima fase, interrompe o processamento de suas pares e finaliza esta fase. Em termos de desempenho, este procedimento parece só perder para seu análogo seqüencial se a primeira das tarefas da seqüência obtiver êxito, pois há que se contabilizar, no esquema paralelo, o tempo gasto na comunicação (*overhead*). Qualquer outro caso, desde que a sobrecarga de comunicação não consuma tempo comparável ao processamento das tarefas, deverá apresentar melhor desempenho, e esta melhoria deverá ser crescente na medida em que a tarefa que venha a obter sucesso esteja mais distanciada da primeira na seqüencial original , atingindo seu ápice se todas as tarefas falharem ou se apenas a última da cadeia vier a ser bem sucedida.

Quando se trata da segunda fase, este cenário pode se alterar.

Se a primeira fase fornecer como resultado apenas uma sub imagem, o que equivale a dizer que somente em uma sub imagem foram detectados indícios da presença da placa de licença do automóvel, então o procedimento a ser aplicado é análogo ao anterior e valem as considerações tecidas acima. Porém, se mais de uma sub imagem for apresentada a esta fase do processo, o que é fato comum, então uma estrutura MPMD é delineada.

Uma possibilidade de distribuição, esquematizada na figura 18, estabelece que cada sub imagem gerada é entregue ao conjunto de tarefas que compõem a segunda fase e que são potencialmente capazes de segmentar o código de licença (desde que a imagem completa da placa do automóvel esteja contida na sub imagem tratada) e são, dessa forma, processadas simultaneamente. Esta solução privilegia o desempenho, porém exige recursos computacionais que não podem ser dimensionados com antecedência, visto que não se pode prever a quantidade máxima de sub imagens que podem ser geradas, assim, tal modelo não é aplicável na prática.

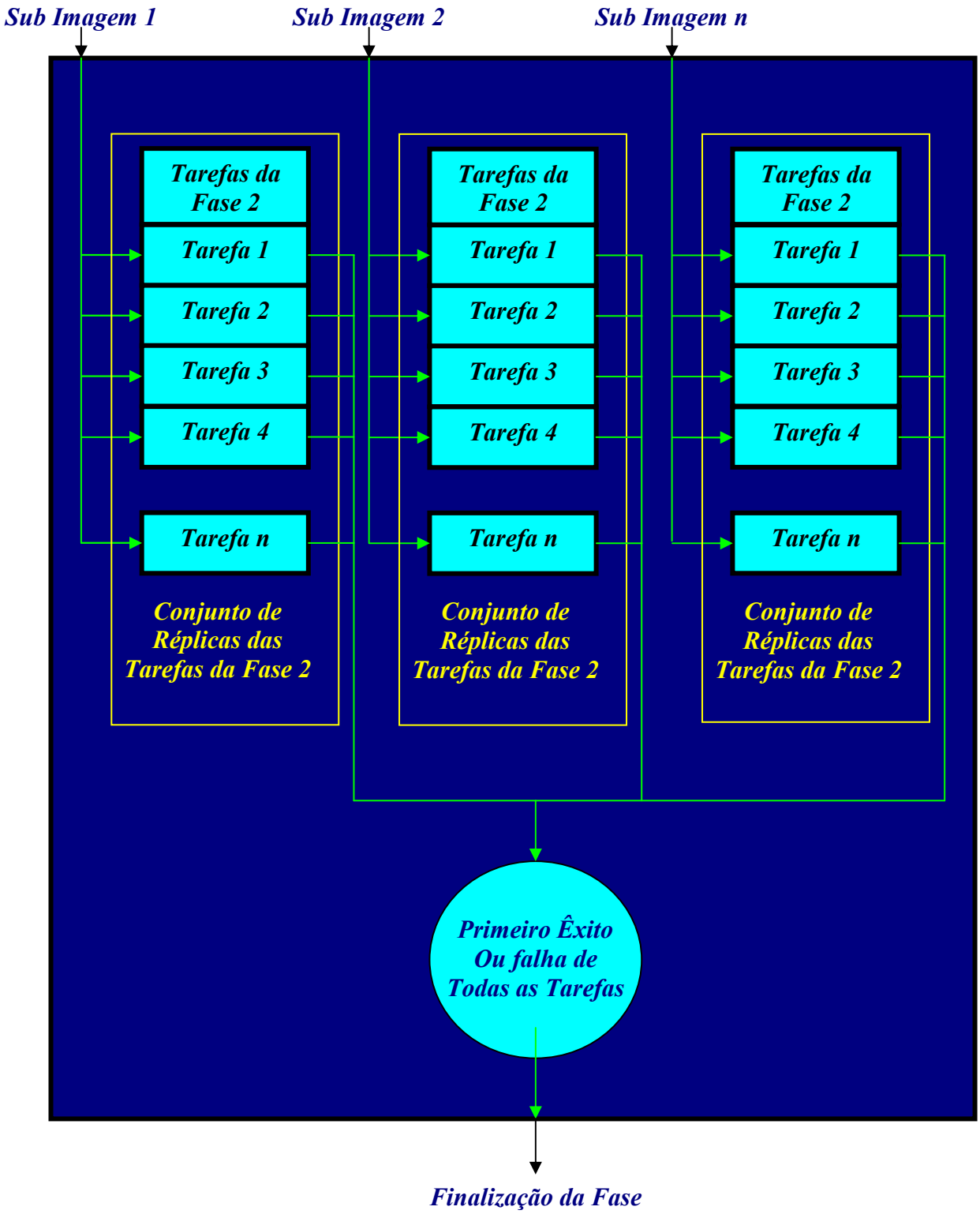


Figura 18 – Esquema MPMD

Uma outra estratégia pode ser a seqüencialização da apresentação das sub imagens, obedecendo ao modelo MPSD. Neste caso, cada tarefa busca uma sub imagem, processa e, se não produziu resultado positivo, então busca a próxima e repete o procedimento. O processo se repete até que se obtenha êxito ou que todas sub imagens tenham sido processadas por todas as tarefas. A figura 19 ilustra tal estrutura.

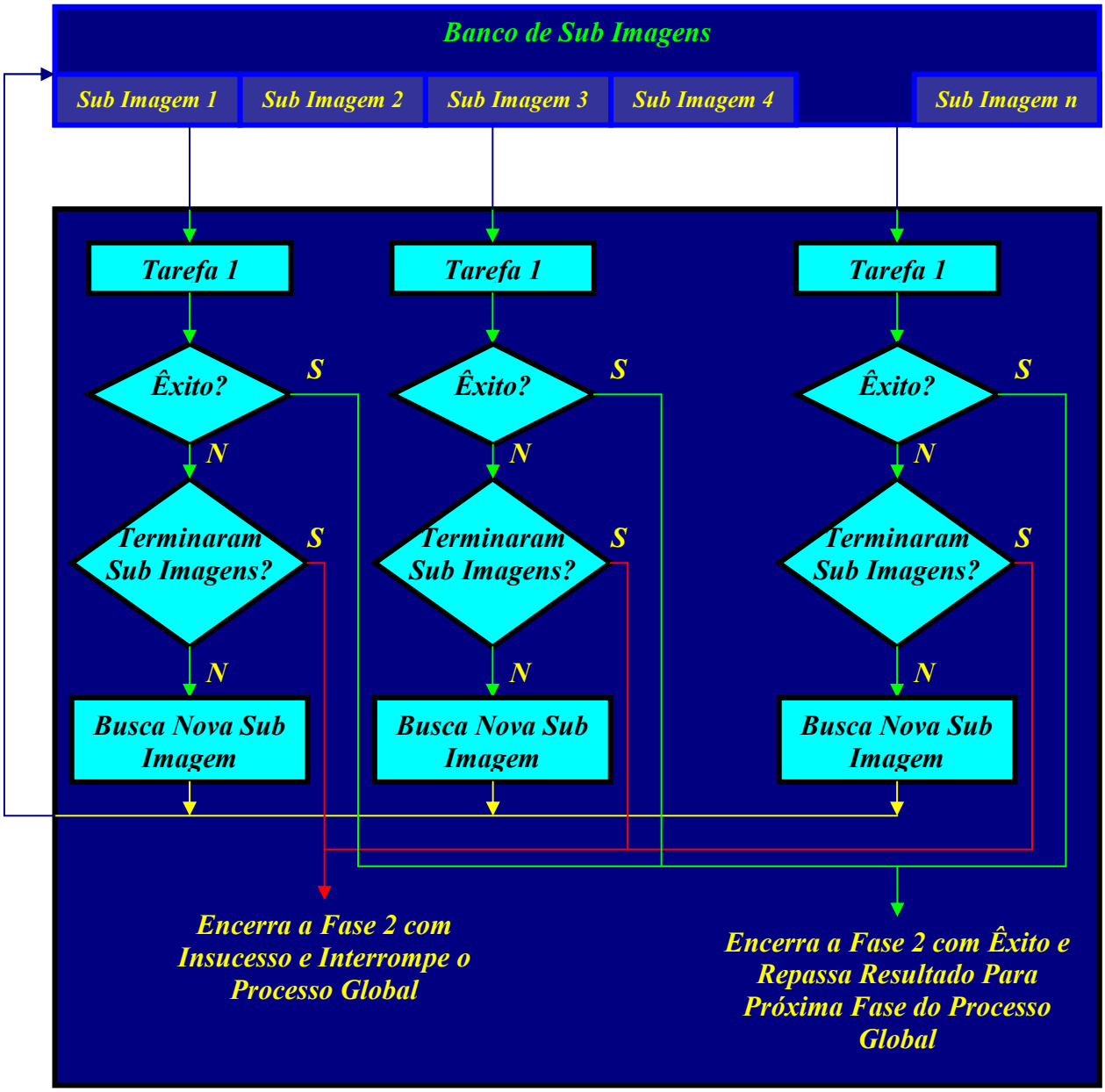


Figura 19 – Esquema MPSD – Multiple Program Simple Data

Pode acontecer do veículo estar desprovido da placa de licença ou ainda que esta apresente características de deterioração e / ou adulteração que impossibilitem sua identificação no conjunto de sub imagens apresentado, no entanto, é preciso lembrar que, se isso não ocorrer, apenas uma sub imagem conterá a placa de licença e, portanto, assim que ela for processada pela segunda fase e seu código for segmentado o processamento se encerra.

Todo o processo de paralelização até aqui discutido pode ser implementado em uma única máquina dotada de um único processador, em uma única máquina multiprocessada ou ainda em várias máquinas constituídas por um ou mais processadores.

No primeiro os recursos de máquina são disponibilizados temporariamente a cada tarefa por um período determinado, dentro do qual ela executa trabalho, isto caracteriza o *pseudo paralelismo*, onde, num determinado instante, apenas uma tarefa realiza trabalho, todas as outras aguardam.

Máquinas multiprocessadas são eficientes para o tratamento de casos análogos, mas o custo envolvido na aquisição e manutenção de equipamentos que incorporam essa tecnologia restringem seu uso a aplicações de alto valor econômico (aeroespaciais, previsão de tempo e prospecção de petróleo são exemplos).

A última proposição vem sendo considerada com relevante empenho e tem obtido adeptos ao redor de todo o mundo, sobretudo nas últimas décadas, por se mostrar uma solução economicamente viável, principalmente quando se considera a possibilidade de aproveitar períodos de ociosidade a que boa parte das máquinas são submetidas. Propostas tem sido apresentadas, estudadas e implementadas visando o aperfeiçoamento de técnicas que possam conduzir a implementações desse tipo sempre com objetivo de melhorar desempenho, compartilhar recursos e minimizar custos.

Num ambiente de processamento paralelo as *tarefas* até aqui tratadas originam processos que podem ou não serem executados em processadores diferentes. Se ocorrer de serem distribuídos em máquinas diversas, de alguma maneira interligadas por uma rede de comunicação, então o processamento, além de paralelo, é também distribuído entre essas máquinas.

Para este exemplo de aplicação específico, a ocorrência de elevado fluxo de veículos por prolongados períodos de tempo pode conduzir ao esgotamento da capacidade de armazenamento de uma única máquina destinada a esse fim e, em decorrência disso, a conseqüente perda de dados, além de impedi-la de executar qualquer processamento.

Uma solução para esse problema pode ser a transferência das imagens coletadas para que sejam processadas em outra máquina, deixando a primeira exclusivamente para a coleta dessas imagens. No entanto, isto só viria a transferir também o problema, pois se a quantidade de imagens exceder as capacidades de processamento e armazenamento desta segunda máquina, o problema persiste, apenas em lugar diferente. Assim, uma solução aparentemente mais adequada para esta situação pode ser a distribuição das imagens entre diversas máquinas onde seriam localmente processadas. Esta hipótese pressupõe a disponibilidade de máquinas interligadas em rede e capazes de trocar informações, conforme ilustrado no esquema da figura 20 abaixo:

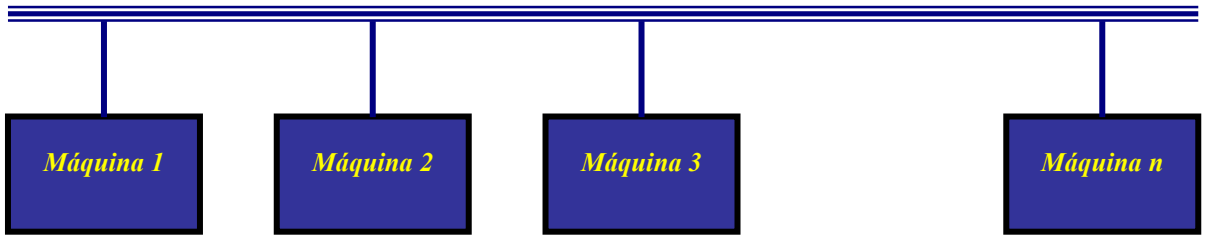


Figura 20 – Máquinas Distribuídas em Rede

Computação paralela e computação distribuída são áreas do conhecimento da ciência da computação que embora tenham surgido com objetivos diferentes (a primeira procura essencialmente melhorar o desempenho mediante o aumento de velocidade de execução dos programas, enquanto que a segunda tem como meta fundamental promover o compartilhamento de recursos e serviços, conforme cita Gubitoso (2005), têm experimentado forte inter relacionamento, sobretudo na última década, tendendo a convergirem rapidamente convergência, conforme demonstram o crescente número de

trabalhos que têm sido orientados no sentido de aplicar computação paralela sobre plataformas distribuídas.

As particularidades apresentadas por este exemplo de aplicação levam a crer que o emprego de processamento paralelo sobre plataformas distribuídas venha a acrescentar substancial melhora de desempenho na solução do problema.

7 MECANISMO GENERICO PARA DISTRIBUIÇÃO E GERENCIAMENTO

O presente trabalho apresenta um mecanismo genérico que tem por objetivo viabilizar a distribuição e paralelização de processos que apresentem necessidades semelhantes àquelas expostas no exemplo de aplicação.

Para tanto foi adotado o modelo Mestre – Operário e desenvolvido um *mecanismo de distribuição e gerenciamento de processos globais* que consiste, fundamentalmente, de dois modelos básicos de programas que utilizam a técnica de *passagem de mensagem explícita* para prover sincronismo e comunicação entre processos: um para o *processo Mestre* e outro para os *processos Operários*.

A implementação utilizada baseia-se na especificação de biblioteca para passagem de mensagem MPI – *Message Passing Interface* desenvolvida e mantida pelo Laboratório de Sistemas Abertos da Universidade de Indiana – EUA designada LAM MPI – *Local Area Multicomputer (Multiprocessor) Message Passing Interface*, de código aberto e livre uso que obedece todas as especificações contidas no padrão consensual MPI - ! e grande parte daquelas contidas no MPI - 2.

O mecanismo foi desenvolvido em linguagem C e executado sobre um sistema operacional Linux baseado na distribuição Debian 3.1 r2 acrescida dos componentes necessários à MPI.

Em essência, o modelo implementado para o *processo Mestre* é responsável pela distribuição dos dados para os *processos Operários* implementados pelo modelo Operário e pelo recebimento das respostas enviadas por esses últimos bem como pelo gerenciamento do processo global, podendo, redistribuir novos dados a cada interação. O modelo Operário foi desenvolvido para implementar os diversos processos operários envolvidos num processamento. Basicamente ele recebe dados do processo Mestre e opera sobre eles segundo um algoritmo nele previamente instalado, devolvendo a resposta obtida ao processo Mestre.

Da maneira como está concebido, o modelo Operário admite um algoritmo capaz de operar sobre dados a ele entregues de maneira autônoma e independente e devolver respostas ao processo mestre.

O modelo permite que se gere tantos processos Operários quantos sejam necessários, desde que devidamente acrescentados ao Mestre.

Durante o desenvolvimento dos modelos Mestre e Operário foram realizadas ensaios que permitiram verificar a operação do sistema como um todo, entre elas destaca-se o final, que pretende simular, o mais próximo possível da realidade, as duas fases do processo global do exemplo de aplicação previamente discutido.

Para efeito desta simulação assume-se um processo global virtual dividido em duas fases, cada delas uma contendo dois e quatro processos respectivamente (essas quantidades podem variar sem prejuízo da simulação). Mantidos esses números, o processo Mestre simulado pode ser assumido como real para o exemplo de aplicação previsto sem que seja necessário modifica-lo.

Os processos Operários simulados (seis no total, sendo dois associados à primeira fase e quatro à segunda) executam toda a parte de comunicação e troca de informações de maneira exata a realidade, porém, os algoritmos que manipulam os dados são simulados por temporizadores, assim, as respostas geradas por eles também são simuladas.

Na primeira fase o processo obedece o modelo MPSD, onde um conjunto de arquivos é assumido como dados a serem manipulados pelos processos operários responsáveis. O conjunto é o mesmo para todos os processos envolvidos e são manipulados simultânea e conjuntamente por cada um deles, o que faz com que se comportem como dados únicos (SD) para todos os processos (MP), justificando a abordagem MPSD.

Para a segunda fase foi estabelecido o modelo MPMD, em que os arquivos gerados na primeira fase são assumidos como dados para cada um dos processos que compõem esta segunda fase, porém, diferentemente do que ocorre na fase anterior, cada um desses arquivos é tratado separadamente por cada um dos processos desta fase, comportando-se como múltiplos dados (MD) executados sobre múltiplos processos (MP) simultaneamente.

A simulação do processo ocorre segundo a seguinte seqüência de acontecimentos:

- O processo Mestre recebe a informação de quantos arquivos compõem o conjunto de dados originais. Convencionou-se que esses arquivos, designados por "*imagem_X.bmp*", onde *X* é o índice do arquivo iniciado por 1, sejam disponibilizados numa área conhecida por todos os processos, Mestre e Operários.

- O processo Mestre transmite aos processos Operários associados à primeira fase o número de arquivos que compõem o conjunto de dados originais e aguarda respostas deles.

- Cada processo Operário recebe a informação transmitida pelo Mestre, busca cada arquivo original na área a eles destinada e simula a aplicação do processo específico atribuído a cada Operário sobre estes. Esta simulação prevê a geração de arquivos que, por convenção, são designados por "*imagemPX_Y.bmp*", onde *X* é o número do posto (rank) ocupado pelo processo que originou o arquivo no grupo considerado e *Y* o seu índice iniciado por 1, e disponibilizados na área comum destinada a esse fim.

- Após a simulação, cada processo Operário transmite ao processo Mestre a quantidade de arquivos que gerou e que se encontram disponíveis na área previamente convencionada.

- Após receber as respostas de todos processos Operários, o processo Mestre atribui novos nomes aos arquivos gerados segundo a convenção "*imagem_X.bmp*", onde *X* é o índice atribuído a cada um e iniciado por 1. Esta operação faz com que todos os arquivos obtidos na fase anterior sejam ordenados, independente dos processos que os geraram. Realizada esta operação, o processo Mestre transmite aos processos Operários que compõem a segunda fase (quatro, para efeito desta simulação) a quantidade de arquivos gerados pelos processos da fase anterior e disponibilizados na área destinada a esse fim e aguarda respostas desses processos.

- Os processos Operários da Segunda fase, de posse da informação enviada pelo processo Mestre, buscam os arquivos gerados na fase anterior e simulam o seu processamento, fazendo com que seja simulado que um a um dos arquivos é

processado, até que um dos processos simule ter encontrado a resposta ou ainda que todos os arquivos tenham sido trabalhados por todos os processos e nenhum deles tenha obtido sucesso.

- Quando um dos processos Operários desta fase simula a obtenção de uma resposta, ele a transmite para o processo Mestre e encerra o processo associado. Se, ao contrário, todos os processos Operários simulam não terem obtido sucesso, então todos eles comunicam o fato ao processo Mestre e, a medida que tal resposta é obtida por cada processo Operário, ele encerra seu processo associado após transmitir a informação ao processo Mestre. Na hipótese de uma resposta positiva, convencionou-se que o processo Operário que a gera nomeie o arquivo a ela destinado segundo o tipo "*resultadoPX.bmp*", onde *X* é o número do posto (rank) ocupado pelo processo que a gerou no grupo.

- Se o processo Mestre recebe uma resposta bem sucedida, então ele a notifica e atribui o nome "*resultado.*bmp" ao arquivo resultante e encerra o processo global.. Caso contrário, isto é, se todos os processos Operários informam não terem obtido êxito, então o processo Mestre também notifica o insucesso da operação e encerra o processo global, porém, isto só ocorre após o recebimento de tal indicação de todos os processos Operários.

8 SIMULAÇÃO DO EXEMPLO DE APLICAÇÃO - RESULTADOS OBTIDOS E CONCLUSÕES

O desenvolvimento do mecanismo culminou com a simulação de três situações possíveis para o exemplo de aplicação considerado:

- O sistema falha na primeira fase do processo, portanto, não consegue discriminar regiões da cena que potencialmente possam conter a placa de licença do veículo;
- O sistema é bem sucedido na primeira fase do processo, portanto, discrimina regiões que potencialmente podem conter a placa de licença do automóvel, porém, falha na obtenção dos sete caracteres segmentados que identificam o veículo, operação atribuída à segunda fase do processo e, finalmente,
- O sistema obtém sucesso nas duas fases e, conseqüentemente, no processo global.

Para esta simulação a distribuição dos processos segue o esquema ilustrado na figura 21 subseqüente.

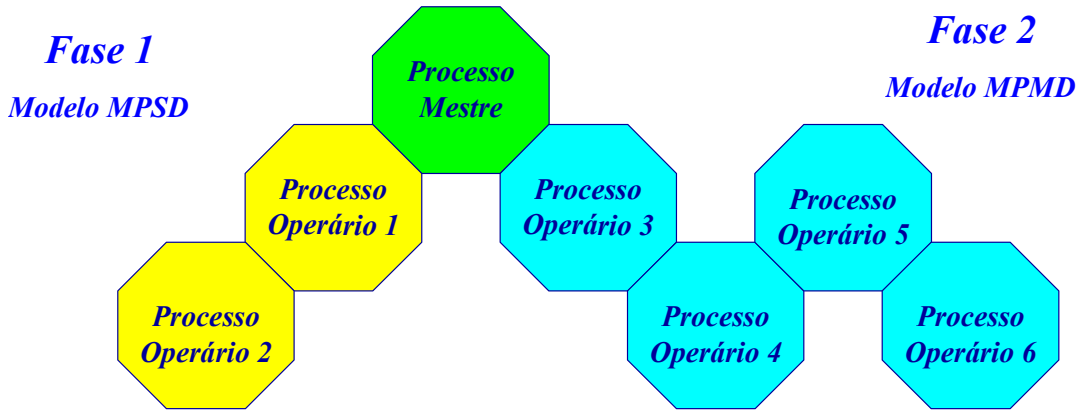


Figura 21 – Esquema de Distribuição dos Processos Para a Simulação

O Processo Mestre distribui os dados originais aos Processos Operários 1 e 2 e aguarda as respostas decorrentes dos processamentos neles executados.

Caso os Processos Operários 1 e 2 não forneçam respostas positivas, então o Processo Global é finalizado sem sucesso.

Após receber respostas positivas dos Processos Operários 1 e 2, o Processo Mestre distribui os novos dados (obtidos dos processamentos realizados nos Processos Operários 1 e 2) aos Processos Operários 3, 4, 5 e 6 e aguarda a primeira resposta positiva proveniente de um deles e determina o encerramento do Processo Global com êxito. Caso contrário, isto é, se os Processos Operários 3, 4, 5 e 6 finalizam suas tarefas e não obtém sucesso, então o fato é comunicado ao processo Mestre que, por sua vez, encerra o Processo Global sem sucesso.

Cada uma dessas possíveis situações foi submetida a três ambientes distintos de execução com o intuito de avaliar o desempenho do sistema em cada uma dessas situações sobre cada um dos ambientes de execução relacionados abaixo, mantendo-se sempre, rigorosamente, as mesmas condições de ensaio em todos os casos::

- Um microcomputador pessoal dotado de processador Athlon XP de 1,7 MHz e com 512 MB de memória RAM;
- Um notebook dotado de processador Pentium M de 1,5 MHz e com 256 MB de memória RAM e, finalmente,
- Uma rede constituída pelas duas máquinas acima especificadas, interligadas via placas ethernet a 100 Mbits / s utilizando protocolo TCP IP.

Os dois primeiros ambientes proporcionam a pseudo paralelização dos processos uma vez que um único processador é incumbido de executar todos os seis processos simulados e para tanto se utiliza da divisão de tempo para a execução de partes de cada processo a ser executado paralelamente.

O último, embora distribuído, também acaba por realizar um pseudo paralelismo, pois o número de processos executados simultaneamente, pelo menos na segunda fase, excede a

quantidade de processadores disponíveis, no caso, apenas dois; mas, nem por isso deixa de ser um ambiente distribuído.

A distribuição de tarefas para este último ambiente segue o ilustrado na figura 22 abaixo:

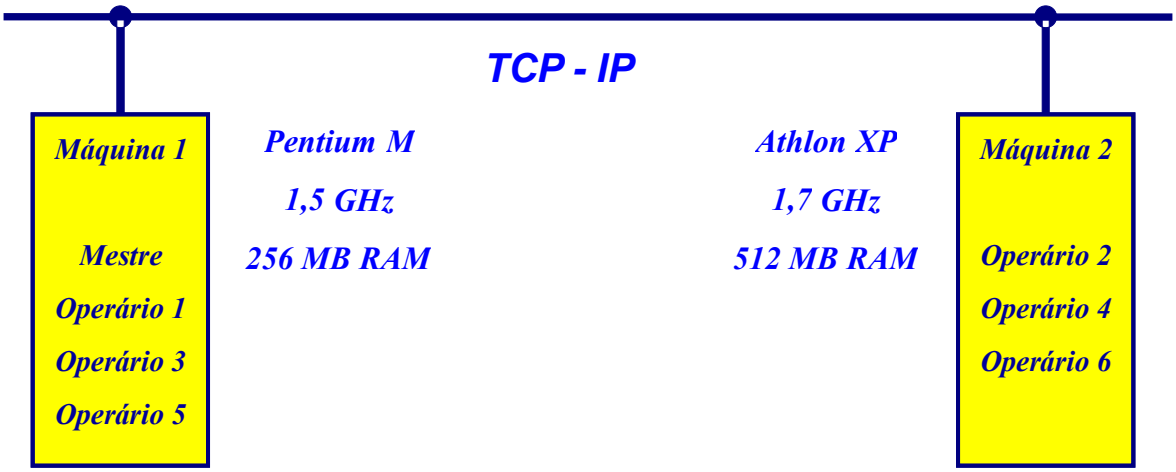


Figura 22 – Esquema de Distribuição de Tarefas em Rede

A máquina 1 recebe as tarefas atribuídas ao Processo Mestre e aos Processos Operários 1, 3 e 5, enquanto que à máquina 2 são atribuídas as tarefas designadas para os Processos Operários 2, 4 e 6.

Para cada situação foram executados cinco ensaios sobre cada um dos ambientes descritos, totalizando, portanto, quarenta e cinco resultados diferentes que são expostos na seqüência:

8.1 Situação 1 - Simulação de Falha na Primeira Fase do Processo

Para a simulação desta situação foram fornecidos oito arquivos de imagens originais como dados de entrada.

8.1.1 Ambiente de Execução 1 – Processador Athlon XP

Tabela 1 – Desempenho do Processador Athlon XP na Situação 1

Simulação	*Tempo de Execução*
1	***117,754038 s***
2	***116,124736 s***
3	***116,024726 s***
4	***116,188982 s***
5	***117,143379 s***
Média	***116,647172 s***

Fonte: Elaborado pelo autor

8.1.2 Ambiente de Execução 2 – Processador Pentium M

Tabela 2 – Desempenho do ProcessFador Pentium M na Situação 1

Simulação	*Tempo de Execução*
1	***82,105120 s***
2	***81,565913 s***
3	***81,605171 s***
4	***81,602655 s***
5	***81,628875 s***
Média	***81,701547 s***

Fonte: elaborado pelo autor

8.1.3 Ambiente de Execução 3 – Rede Athlon XP – Pentium M

Tabela 3 – Desempenho da Rede Athlon XP – Pentium M na Situação 1

Simulação	*Tempo de Execução*
1	***73,831243 s***
2	***73,869422 s***
3	***73,942012 s***
4	***73,927151 s***
5	***73,890623 s***
Média	***73,892090 s***

Fonte: Elaborado pelo autor

Pode-se verificar uma evolução no desempenho do processo global ao se passar do processamento no Athlon XP para o Pentium M da ordem de 30% (29,96%) e, mais significativo ainda é o crescimento deste desempenho quando se distribui o processamento empregando-se a rede constituída pelos dois processadores, da ordem de 37% (36,65%) em relação ao Athlon XP e quase 10% (9,56%) em relação ao Pentium M.

O quadro 2 sudseqüente resume todas essas informações:

Simulação	*Athlon*	*Pentium*	*Rede*
1	*117,754038s*	*82,105120 s*	*73,831243 s*
2	*116,124736 s*	*81,565913 s*	*73,869422 s*
3	*116,024726 s*	*81,605171 s*	*73,942012 s*
4	*116,188982 s*	*81,602655 s*	*73,927151 s*
5	*117,143379 s*	*81,628875 s*	*73,890623 s*
Média	*116,647172 s*	*81,701547 s*	*73,892090 s*
% Média	*-*	*29,96 %*	*36,65 - 9,56 %*

Quadro 2 – Comparação de Desempenho Para a Situação 1
Fonte: Elaborado pelo autor

8.2 Situação 2 – Simulação de Falha Apenas na Segunda Fase do Processo

Para esta simulação foram entregues ao sistema oito arquivos de imagens originais, cuja primeira fase do processo, constituída por duas tarefas, simula a obtenção de cinco e oito sub imagens resultantes do processamento de cada uma dessas tarefas, totalizando doze arquivos cujo processamento é simulado na segunda fase.

8.2.1 Ambiente de Execução 1 – Processador Athlon

Tabela 4 – Desempenho do Processador Athlon XP na Situação 2

Simulação	*Tempo de Execução*
1	***156,429388 s***
2	***156,389278 s***
3	***156,413687 s***
4	***156,397482 s***
5	***156,251757 s***
Média	***156,376390 s***

Fonte: Elaborado pelo autor

8.2.2 Ambiente de Execução 2 – Processador Pentium M

Tabela 5 – Desempenho do Processador Pentium M na Situação 2

Simulação	*Tempo de Execução*
1	***111,538726 s***
2	***111,611192 s***
3	***111,348538 s***
4	***111,349344 s***
5	***111,378573 s***
Média	***111,445275 s***

Fonte:Elaborado pelo autor

8.2.3 Ambiente de Execução 3 – Rede Athlon XP – Pentium M

Tabela 6 – Desempenho da Rede Athlon XP – Pentium M na Situação 2

Simulação	*Tempo de Execução*
1	***97,318220 s***
2	***97,135342 s***
3	***98,210862 s***
4	***98,353385 s***
5	***97,554014 s***
Média	***97,714385 s***

Fonte: Elaborado pelo autor

Também nesta situação pode-se verificar a significativa evolução no desempenho do processo global entre os ambientes dotados do processador Athlon XP e Pentium M da ordem de quase 30% (28,73%), bem como naquele verificado quando do emprego da rede constituída pelos dois processadores, cujo incremento foi da ordem de 38% (37,51%) em relação ao Athlon XP e de quase 12,5% (12,32%) em relação ao Pentium M, conforme resumo apresentado no quadro 3 subseqüente:

Simulação	*Athlon*	*Pentium*	*Rede*
1	*156,429388 s*	*111,538726 s*	*97,318320 s*
2	*156,389278 s*	*111,611192 s*	*97,135342 s*
3	*156,413687 s*	*111,348538 s*	*98,210862 s*
4	*156,397482 s*	*111,349344 s*	*98,353385 s*
5	*156,251757 s*	*111,378573 s*	*97,554014 s*
Média	*156,376390 s*	*111,445275 s*	*97,714385 s*
% Média	*-*	*28,73 %*	*37,51 - 12,32 %*

Quadro 3 – Comparação de Desempenho Para a Situação 2
Fonte: Elaborado pelo autor

8.3 Situação 3 – Simulação de Sucesso no Processo Global

Também nesta simulação foram entregues ao sistema oito arquivos de imagens originais, cuja primeira fase do processo, constituída por duas tarefas, simula a obtenção de cinco e oito sub imagens resultantes do processamento de cada uma dessas tarefas, totalizando doze arquivos cujos processamento e sucesso da operação são simulados na segunda fase.

8.3.1 Ambiente de Execução 1 – Processador Athlon

Tabela 7 – Desempenho do Processador Athlon XP na Situação 3

Simulação	*Tempo de Execução*
1	***150,131670 s***
2	***150,002492 s***
3	***149,656175 s***
4	***149,374208 s***
5	***149,309243 s***
Média	***149,894758 s***

Fonte: Elaborado pelo autor

8.3.2 Ambiente de Execução 2 – Processador Pentium M

Tabela 8 – Desempenho do Processador Pentium M na Situação 3

Simulação	*Tempo de Execução*
1	***111,921498 s***
2	***110,862836 s***
3	***111,636000 s***
4	***110,949997 s***
5	***111,595006 s***
Média	***111,393067 s***

Fonte: Elaborado pelo autor

8.3.3 Ambiente de Execução 3 – Rede Athlon – Pentium M

Tabela 9 – Desempenho da Rede Athlon XP – Pentium M na Situação 3

Simulação	*Tempo de Execução*
1	***91,624202 s***
2	***91,399676 s***
3	***91,929621 s***
4	***91,906172 s***
5	***91,456057 s***
Média	***91,663146 s***

Fonte: Elaborado pelo autor

Também nesta situação pode-se verificar a significativa evolução no desempenho do processo global entre os ambientes dotados do processador Athlon XP e Pentium M da ordem de quase 26% (25,69%), bem como naquele verificado quando do emprego da rede constituída pelos dois processadores, cujo incremento foi da ordem de 40% (38,85%) em relação ao Athlon XP e de quase 18% (17,71%) em relação ao Pentium M, conforme apresentado no quadro 4 subseqüente:

Simulação	*Athlon*	*Pentium*	*Rede*
1	*150,131670 s*	*111,921498 s*	*91,624202 s*
2	*150,002492 s*	*110,862836 s*	*91,399676 s*
3	*149,656175 s*	*111,636000 s*	*91,929621 s*
4	*149,374208 s*	*110,949997 s*	*91,906172 s*
5	*149,309243 s*	*111,595006 s*	*91,456057 s*
Média	*149,894758 s*	*111,393067 s*	*91,663146 s*
% Média	*-*	*25,69 %*	*38,85 - 17,71 %*

Quadro 4 – Comparação de Desempenho Para a Situação 3
Fonte: Elaborado pelo autor

8.4 Conclusões

Em todas as situações simuladas pode-se observar significativos acréscimos no desempenho do sistema quando se empregou a distribuição de tarefas através da diminuta rede constituída pelos dois microcomputadores (sempre superior a 9%).

Há que se considerar que em todos os casos as simulações impuseram rigoroso controle sobre o balanceamento de carga, obtido através de temporizadores que simulavam o processamento nas diferentes tarefas e que possuiam valores bem próximos uns dos outros, de maneira que as tarefas quase que eram encerradas ao mesmo tempo, fato que seguramente contribui positivamente para a melhoria do desempenho em processamentos paralelos / distribuídos, porém nem sempre é possível de se obter na prática.

Entretanto, tomados os devidos cuidados na distribuição das tarefas, o sistema, no caso real deve propiciar desempenho similar a este obtido nas simulações.

O mecanismo desenvolvido se mostrou portanto eficiente para ser empregado em aplicações desse tipo, demonstrando ser capaz de auferir significativos ganhos de desempenho aos processos assim tipificados.

9 CONSIDERAÇÕES FINAIS E PERSPECTIVAS PARA TRABALHOS FUTUROS

Avanços significativos foram alcançados durante o desenvolvimento deste trabalho, sobretudo no que se refere à capacitação desenvolvida na área de processamento paralelo sobre plataformas distribuídas utilizando o modelo de passagem de mensagem.

O estudo realizado sobre a especificação MPI – Message Passing Interface, com ênfase na implementação LAM MPI acrescentou novos conhecimentos suficientes para a realização deste trabalho e para a proposição de outros, constituindo-se num marco inicial de extrema importância para o desenvolvimento de novos projetos utilizando as técnicas estudadas.

Quanto a trabalhos futuros, sugere-se que o primeiro deles utilize o mecanismo aqui desenvolvido sobre o exemplo de aplicação que motivou este trabalho, aplicando a ferramenta a um caso real.

Uma segunda proposição é de se utilizar distribuições diferentes para problemas semelhantes ao do exemplo de aplicação, uma delas pode ser em árvore, onde o processo Mestre coordena apenas a primeira fase do processo global e, à medida que os processos Operários desta fase obtêm respostas positivas, eles passam a se comportar como processos Mestres dos Operários da fase seguinte e assim sucessivamente até que todas as fases sejam cumpridas. Espera-se que um procedimento assim venha a melhorar o desempenho global de processos similares.

Outra melhoria que pode ser implementada, e deve pelo menos ser estudada, diz respeito a possibilidade do processo Mestre gerenciar não apenas um único processo global por vez, mas vários deles. Isto é possível mediante a criação de grupos de processos e o emprego de diferentes manipuladores de comunicação associados a cada grupo, conforme previsto no padrão MPI.

É desejável também a evolução do presente trabalho no sentido de que os processos possam ser abrigados em máquinas dotadas de diferentes sistemas operacionais, sobretudo o

Windows, tendo em vista a grande quantidade de computadores assim configurados, para tanto aconselha-se o estudo de implementações diferentes do MPI que admitam tal heterogeneidade, como, por exemplo, o MPICH.

A evolução final desejável para um trabalho deste tipo é a uso do tempo ocioso de máquinas componentes de uma determinada rede para a execução de processos distribuídos, similar ao *grid*. Uma configuração assim poderia abrigar cópias de processos Operários em diversas máquinas e, quando solicitado, um processo Mestre procuraria por máquinas ociosas e atribuiria trabalho a elas. Um procedimento assim deve se aproximar do caso ideal no que se refere ao aproveitamento de recursos e melhoria de desempenho, objetivos almejados pelos processamentos distribuído e paralelo respectivamente e cuja convergência parece ser irreversível.

Referências

ALVIM, A. C. F. . **Bases de Programação MPI**. Rio de Janeiro : PUC-RJ, S.d. (tutorial), Disponível em: http://www.inf.puc-rio.br/~alvim/MPI/MPI.html .Acesso em: 06 jul 2006.

ENDLER, M.; GUBITOSO, M. D.; KOCK, G.; SONG, S. W. Towards Transparent Parallelization of Connectionist Systems. **PDCS 96**, Dijon – France, Sept. 1996.

FIALLOS, M.; MELCIADES, W.; PIMENTEL, C. Paralelização do Algorítmo "Backpropagation" em Clusters de Estações de Trabalho. In: BRAZILIAN CONFERENCE ON NEURAL NETWORKS, 4., 1999, São José dos Campos. **Proceedings…**[S.l: s.n.], 1999.

FLYNN, M. J. Some computer organizations and their effectiveness. **IEEE Transactions on Computers**, New York, v.21, n.9, p.948-960, Sept. 1972.

GUBITOSO, M. D. **MAC431 - Introdução ao processamento paralelo e distribuído.** São Paulo: IME/USP, 2005 (notas de aula). Disponível em: <http://mairinque.ime.usp.br/~gubi/cursos/431/apostila/>. Acesso em: 04 mar. 2006.

LINHALIS, F.; FIATS, M. I. **PVM e MPI**. São Carlos, USP: EESC, 1998. Disponível em: <http://black.rc.unesp.br/gpacp/bibliografia/D-01%20-%20PVM%20e%20MPI%20-%20ICMC%20-%20USP%20-%20S%C3%A3o%20Carlos.doc>. Acesso em: 04 mar. 2006.

ROMIG III, P. R.; SAMAL, A. **DeVious:** A Distributed Environment For Computer Vision; Software – Practice and Experience, Lincoln - NE, v.25, n.1, p.23-45, Jan. 1995.

MARTINS, F. P. R.; QUIRÓZ, L. H. C.; LIMA, P.S. P.; ALMEIDA, R. Z. H.; BRUNA, W. C.; ROCHA, W.C.M.; **Identificação do Código de Licença de Veículos Automotores Usando Visão Computacional**. São Paulo: IPT-FAPESP, 2003. (Relatório Técnico IPT / DME / ASC n^{0} 68.109) [1]

MARTINS, F. P. R.; LIMA, P. S. P.; QUIRÓZ, L. H. C.; BRUNA, W.; ALMEIDA, R. Z. H.; **Sistema de Leitura Automática de Códigos de Licença**. Proceedings on Brazilian Symposium on Neural Networks, Maranhão – 2004 – Paper #2943

PACHECO, P.S. **A user's guide to MPI**. California: University of San Francisco, 1998. Disponível em: ftp://math.usfca.edu/pub/MPI/mpi.guide.ps.Z . Acesso em: 08 mai 2006.

Referências Consultadas

ALCÂNTARA, D. M.; LOPES, G. P; ALVES, S.G.C. **Análise de desempenho em passagem de mensagem.** 2003. Trabalho de Conclusão de Curso (Graduação) - Ciência da Computação, Universidade Católica de Brasília, Brasília, 2003.

BUENO A. D **Introdução ao processamento paralelo e ao uso de clusters de workstacions em sistemas GNU/LINUX parte I:** filosofia. Florianópolis: LMTP/UFSC, 2003. Disponível em: <http://www.lenep.uenf.br/~bueno/Artigos/117-IntroducaoProgramacaoParalelaECluster-P1.pdf>. Acesso em: 04 mar. 2006.

BUENO, A. D. **Introdução ao processamento paralelo e ao uso de clusters de workstations em sistemas GNU/LINUX parte II:** processos e threads. Florianópolis: LMTP/UFSC, 2003. Disponível em: <http://www.lenep.uenf.br/~bueno/Artigos/118-IntroducaoProgramacaoParalelaECluster-threads-P2.pdf>. Acesso em: 04 mar. 2006

CABAÇO, S. **Sistemas de processamento paralelo e distribuído :** modelos de concorrência, distribuição e paralelismo em Java estudo de modelos de comunicação e plataformas oferecidas pela linguagem. Disponível em: <http://clientes.netvisao.pt/sucabaco/trabalhos/SPPD/SPPD_TS.pdf>. Acesso em: 04 mar. 2006.

DIETZ, H. **Linux parallel processing HOWTO**. West Lafayette: Purdue University, 1998. Disponível em: <http://yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto.html>. Acesso em: 04 mar. 2006.

FORTUNA, A. O. F.. **Introdução ao MPI**. Rio Claro: UNESP, 2002?. Disponível em: <http://black.rc.unesp.br/gpacp/bibliografia/B-01%20-%20Introdu%C3%A7%C3%A3o%20ao%20MPI%20-%20ICMC%20-%20USP%20-%20S%C3%A3o%20Carlos.ppt>. Acesso em: 04 mar. 2006.

FOSTER, I.; KESSELMAN, C; TUECKE, S. The anatomy of the grid: enabling scalable virtual organizations**. International Journal Supercomputer Applications**, v.15, n.3, p.200-222, 2001. Disponível em: <http://www.globus.org/alliance/publications/papers/anatomy.pdf>. Acesso em: 04 Mar. 2006.

GEIST, A. L. et al. **PVM:** parallel virtual machine: a users' guide and tutorial for networked parallel computing . London: MIT Press, 1994. 176p.

GROPP, W. **Tutorial on MPI:** the message-passing interface. Chicago: Mathematics and Computer Science Division/ Argonne National Laboratory, 2001

Disponível em: <http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html>. Acesso em: 04 mar. 2006.

SANTOS, E.B..; SECKLER, M.M. **Guia para elaboração de dissertação de mestrado**. 2. ed. São Paulo: IPT, 2005. 32p.

SNIR, M. et al. **MPI:** the complete reference. 2. ed. London: MIT Press, 1998. 445p. v.1

TEIXEIRA, H. E. G; BARROS, M. J. A. S.; COELHO, P. J. M. **Clusters Beowulf**. 2000. Trabalho de Conclusão de Curso (Graduação) – Faculdade de Engenharia, Universidade do Porto, Porto, 2000. Disponível em: <ftp://ftp.fe.up.pt/pub/Pessoal/Deec/jmcruz/Beowulf.pdf>. Acesso em: 04 mar. 2006.

THAKUR, R. **Parallel I/O in MPI-2**. Chicago: Mathematics and Computer Science Division/ Argonne National Laboratory, 2002. Disponível em: <http://www.npaci.edu/ahm2002/ahm_ppt/482,1,Parallel I/O in MPI-2>. Acesso em: 04 mar. 2006.

UNIVERSITY OF ILLINOIS. **Introduction to MPI. EUA**:PACS Training Group, 2001. Disponível em: http://foxtrot.ncsa.uiuc.edu:8900/-_Course.pdf Acesso em: 05 maio 2006.