Introdução à programação paralela com MPI em agregados de computadores (clusters)

M. T. Rebonatto[1]
[1] Curso de Ciência da Computação - Instituto de Ciências Exatas e Geociências
Universidade de Passo Fundo, CEP: 99072-320, Brasil
rebonatto@upf.br

RESUMO

M. T. Rebonatto. 2004. Introdução a programação paralela com MPI em agregados de computadores (clusters). Congresso Brasileiro de Ciência da Computação, Itajaí, 2004, 938 – 955. Itajaí, SC – Brasil, ISSN 1677-2822

A evolução dos computadores tem propiciado uma gradual evolução das aplicações que podem ser realizadas utilizando sistemas de computação, quase sempre aumentando a complexidade, volume de dados e a necessidade de poder computacional empregado. Esses requisitos exigidos pelas novas aplicações nem sempre são fornecidos através de computadores monoprocessados, despertando o interesse pelas máquinas paralelas que oferecem suporte ao processamento paralelo e distribuído de aplicações. Dentre as arquiteturas de máquinas paralelas, os agregados de computadores (clusters) despontam como uma alternativa com excelente relação custo/benefício, com possibilidade de competir de igual com os supercomputadores. Estas máquinas podem ser constituídas de computadores simples como PCs, facilitando sua disseminação e utilização. Para a programação das aplicações que irão comandar as máquinas paralelas atuais e vindouras, a programação paralela com uso de bibliotecas para a exploração do paralelismo se constitui numa alternativa muito atrativa, pois concilia o conhecimento de programação já dominado em linguagens tradicionais como C e C++ a utilização de recursos para a exploração do paralelismo. Dentre as bibliotecas para exploração do paralelismo disponibilizadas, destaca-se o Messaging Passing Interface – MPI, padrão de fato e direito na escrita de aplicações paralelas para serem executadas em agregados de computadores. Dessa forma, este texto contém informações relativas à programação de aplicações paralelas clusters utilizando MPI em conjunto com a linguagem de programação C na escrita dos códigos fonte.

PALAVRAS DE INDEXAÇÃO ADITIONAIS: *processamento paralelo e distribuído, bibliotecas para exploração do paralelismo, exploração explícita do paralelismo em programas*

INTRODUÇÃO

Desde o primórdio da era da computação a demanda crescente por processamento tem motivado a evolução dos computadores, viabilizando implementações de aplicações que envolvem uma elevada taxa de computações e grandes volumes de dados. Esse crescimento na capacidade de processamento dos computadores podia ser obtido através do aumento de desempenho dos processadores ou a utilização de vários. Como ampliar a capacidade de processamento dos computadores esbarrava no custo e na capacidade tecnológica, algumas soluções tenderam a utilização de vários processadores em conjunto, surgindo o termo Processamento Paralelo.

Pessoas têm pensado sobre a computação paralela, paralelismo e programação paralela por apenas algumas décadas. Entretanto, elas têm pensado sobre sistemas paralelos por várias centenas de anos, uma vez que no mundo real, várias ações estão ocorrendo em paralelo (GOULART et. al. 1999, p. 9). Pode-se pensar no ambiente de uma empresa, que possua um grande número de divisões e setores e, em um determinado momento estão sendo desenvolvidas atividades em todos os seus recintos. Pode-se considerar também que existem outras empresas, em diversas cidades, países, etc. Nos casos descritos, tudo ocorre em paralelo.

O paralelismo é uma técnica utilizada em tarefas grandes e complexas para obter resultados mais rápidos, dividindo-as em tarefas pequenas que serão distribuídas em vários processadores para serem executadas simultaneamente. Estes processadores comunicam-se entre si para que haja coordenação (sincronização) na execução das diversas tarefas em paralelo. Os principais objetivos do paralelismo são:

- Aumentar o desempenho (reduzindo o tempo) no processamento;
- Resolver grandes desafios computacionais; (CENAPAD, 2001, p. 3)

As execuções de aplicações que necessitam de grande poder computacional possuem um elevado tempo de resposta, de forma que, para diminuir este tempo, pode se utilizar o processamento paralelo. No processamento paralelo, diversos processadores dividem a carga computacional, possibilitando uma aceleração na obtenção dos resultados. Além da redução do tempo de resposta, a utilização de vários processadores na execução de uma aplicação, possibilita que novos problemas possam ser investigados,

aumentando também a quantidade de memória disponível. Aplicações onde não era possível a execução com uma grande quantidade de detalhes podem ser contempladas, pois pode-se aumentar o volume dos dados de entrada a serem processados, não aumentando em escala proporcional o tempo de resposta.

O propósito principal do processamento paralelo é realizar computações mais rapidamente do que as feitas com um único processador, através da exploração da concorrência usando mais de um processador simultaneamente. A definição de processamento paralelo mais empregada foi formulada inicialmente por Hwang em 1984. Segundo o autor, processamento paralelo é uma forma eficiente de processar informações, a qual enfatiza a exploração de eventos concorrentes na computação de um processo (HWANG; XU, 1998). Neste contexto, um computador paralelo é uma coleção de computadores, tipicamente do mesmo tipo, interconectados de certa forma que possibilite uma coordenação de suas atividades. Assume-se que os processadores são utilizados em conjunto para resolver cooperativamente um problema (GOULART et. al. 1999, p. 14).

Para a resolução dos problemas em paralelo, deve-se na maioria dos casos escrever programas que explorem o paralelismo (pode-se utilizar programação seqüencial com geração de binários executáveis através de compiladores paralelizantes, porém esta é uma técnica que geralmente produz aplicações menos eficientes), desta forma é necessário o domínio da programação paralela. A escrita de programas com exploração explícita do paralelismo pode ser realizada utilizando o *Messaging Passing Interface* (MPI), biblioteca padrão para a exploração do paralelismo em programas.

Arquiteturas paralelas

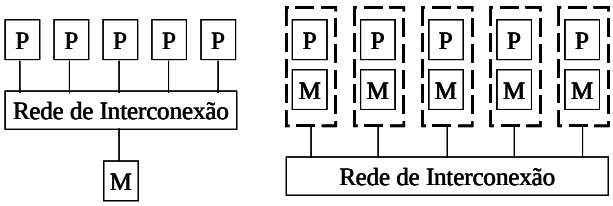
A busca do alto desempenho visando atender à demanda crescente de processamento motivou o surgimento de vários modelos de arquiteturas paralelas. A classificação das máquinas mais utilizada atualmente foi proposta por Michael Flynn em 1972, citada por Silva (SILVA, 1999) e por Zomaya (ZOMAYA, 1996, p. 7). Esta classificação foi elaborada segundo o fluxo de instruções e dados nos processadores e apresenta quatro tipos de máquinas SISD, MISD, SIMD e MIMD. A Figura 1 ilustra a classificação das máquinas paralelas segundo Flynn.

	SD (Single Data)	**MD** (Multiple Data)
SI (Single Instruction)	**SISD** Máquinas monoprocessadas	**SIMD** Máquinas Vetoriais
MI (Multiple Instruction)	**MISD** Sem representante (até agora)	**MIMD** Multiprocessadores/ Multicomputadores

Figura 1 – Máquinas paralelas segundo Flynn

As máquinas SISD possuem um fluxo de instruções atuando sobre um fluxo de dados, sendo as tradicionais máquinas monoprocessadas. A classe MISD possui vários fluxos de instruções atuando sobre um fluxo de dados. Sua classificação é muito contestada por praticamente inexistir aplicações ao seu uso. As divisões SIMD e MIMD representam as máquinas conhecidas como paralelas (NAVAUX et. al., 2001, p.179 e De ROSE, 2001, p. 5), sendo as primeiras com um fluxo de instruções atuando sobre diferentes dados e a segunda possibilitando múltiplos fluxos de instruções sobre dados distintos (KUMAR et. al, 1994, p. 18 e HWANG; XU, 1998, p. 14).

Segundo Navaux (NAVAUX et. al., 2001, p. 184) e Hwang (HWANG; XU, 1998, p. 26), grande parte das máquinas paralelas atuais são da classe MIMD, que possui uma divisão em relação ao espaço de endereçamento de memória. Multiprocessadores são máquinas MIMD que possuem um único espaço para endereçamento da memória, desta forma possibilitando regiões de dados compartilhados. São máquinas geralmente de custo elevado e limitado número de processadores devido a contenção de acesso ao barramento de memória (HWANG; XU, 1998). Os multicomputadores são máquinas MIMD com vários espaços de endereçamento de memória, desta forma não possibilitando memória compartilhada diretamente. Estas máquinas possuem um custo reduzido, uma vez que podem ser empregados componentes de computadores comercias na sua construção e não possuem um limite rígido quanto ao número de computadores. A Figura 2 ilustra os modelos de multicomputador e multiprocessador.



Multiprocessador Multicomputador

Figura 2 – Multiprocessadores e multicomputadores

Interpretando a grafia dos tipos de máquinas MIMD, temos os multiprocessadores (multi + processadores) onde são replicadas as unidades processadoras, com todas acessando a uma mesma memória através da rede de conexão (geralmente um barramento). Por outro lado, nos multicomputadores (multi + computadores) ocorre à replicação das unidades processadoras e também da memória do sistema. Dessa forma, cada processador possui acesso a sua memória, e comunica-se com os demais processadores através da rede de interconexão.

Um dos tipos de multicomputadores que nos últimos anos vem ganhando destaque são as máquinas do tipo *Network of Workstation* (NOW). Segundo Sato, estações de trabalho em rede são uma boa forma de se obter computadores escaláveis, uma vez que podem ser utilizadas até algumas dezenas de estações para um único trabalho (SATO; MIDORIKAWA; SENGER, 1999, p. 5). Os agregados de computadores, também conhecidos como *clusters*, são considerados uma evolução das máquinas NOW (De ROSE, C. A. F., NAVAUX, P. O. A, 2003, p.130). Nestes casos, podem ser utilizados computadores tradicionais como PCs interligados através de uma rede de comunicação. As principais diferenças em relação às máquinas NOW são encontradas em nível de software, onde o Sistema Operacional (SO) pode ser configurado retirando os serviços desnecessários, e na rede de comunicação, sendo utilizadas redes de alto desempenho e baixa latência a fim de obter uma comunicação eficiente (De ROSE, C. A. F., NAVAUX, P. O. A, 2003, p.131).

Atualmente, já são utilizados alguns milhares de computadores interligados em rede, aumentando a capacidade de processamento

dos clusters, deixando-os em condições de competir com os supercomputadores. (HARGROVE; HOFFMAN; STERLING, 2004). Existem hoje vários clusters com grande poder computacional (CLUSTERS@TOP500, 2004), muitos dos quais estão presentes na lista dos supercomputadores mais rápidos do mundo (TOP500 SUPERCOMPUTER SITES, 2004).

Agregados de computadores

Os clusters são as máquinas paralelas mais difundidas atualmente, sendo utilizadas em diversas áreas e para muitas funções distintas. Desta forma, destacam-se os seguintes tipos de agregados de computadores (KINDEL et. al., 2004, p.249):

- *High Performance Computing* – HPC;
- *High Availability* – HA;
- *Horizontal Scaling* – HS.

Os clusters HS são utilizados para o balanceamento de carga eficiente em sistemas onde há um grande número de requisições de serviço a serem executadas (por exemplo servidores da internet). Neste caso, não há processamento paralelo, sendo as requisições individualmente enviadas as máquinas disponíveis. Não possui como objetivo principal à alta disponibilidade (HA), porém pode alcançá-la.

A disponibilidade ininterrupta de um ou mais serviços é o objetivo principal dos clusters HA. Como nos clusters HS não ocorre processamento paralelo (salvo a utilização de software específico). Todos os computadores que compõem o cluster são responsáveis pelos mesmos serviços, podendo ou não ocorrer um balanceamento de carga eficiente.

O processamento paralelo é utilizado nos clusters conhecidos como de alto desempenho (HPC). Neste tipo de cluster, as aplicações especificamente escritas poderão fazer uso de vários processadores na resolução de um único problema, alcançando em determinadas situações um alto desempenho. O desempenho pode ser profundamente influenciado pelo tipo de aplicação, rede de interconexão utilizada além da quantidade e tipo de máquina utilizada na composição do cluster (freqüência de operação, se é multiprocessada, etc). Desta forma, mesmo sem obter um alto desempenho (não depende apenas do cluster), os agregados de computadores utilizados para o processamento paralelo e distribuído são conhecidos como HPC. No restante do texto, as referências a clusters serão todas para este tipo de agregado de computadores.

Componentes de um HPC

Nos dias atuais há uma ampla quantidade de tipos de computadores que podem ser utilizados na composição de clusters. Na prática, existe uma grande quantidade de clusters composta de PCs, um número menor formado de estações de trabalho RISC e reduzidas ocorrências com máquinas de grande capacidade de processamento compondo os agregados de computadores com alto desempenho.

No caso dos clusters formados por PCs (a maioria disponível, com exceção apenas nos grandes centros de pesquisa e empresas de grande porte) pode-se agregar uma máquina com computadores de capacidade de processamento diferente, misturando máquinas de fabricantes, freqüência de operação, quantidade de memória e números de processadores de propósito geral distintos. Para facilitar a manutenção e o balanceamento da carga na execução, grande parte dos clusters é formado de máquinas homogêneas, com uma tendência atual de se utilizar máquinas biprocessadas na sua composição (COSTA; STRINGHINI; CAVALHEIRO, 2002, p. 32). Como exemplo pode-se citar a máquina Amazônia do Centro de pesquisa em Alto Desempenho CPAD – PUCRS/HP (De ROSE, C. A. F., NAVAUX, P. O. A, 2003, p.135).

Por serem máquinas dedicadas ao processamento paralelo de aplicações, a maior parte das máquinas que compõem o cluster não possuem periféricos para utilização por parte de usuários (teclado, vídeo e mouse), e nem ligação direta a nenhuma rede com livre acesso por parte de usuários. A utilização do cluster ocorre através de uma máquina denominada *front-end* que possui os periféricos de I/O padrão. Esta máquina, por vezes denominada de hospedeira, possui ligação com uma rede externa e também com a rede de interconexão da máquina paralela. A Figura 3 ilustra a composição de um cluster.

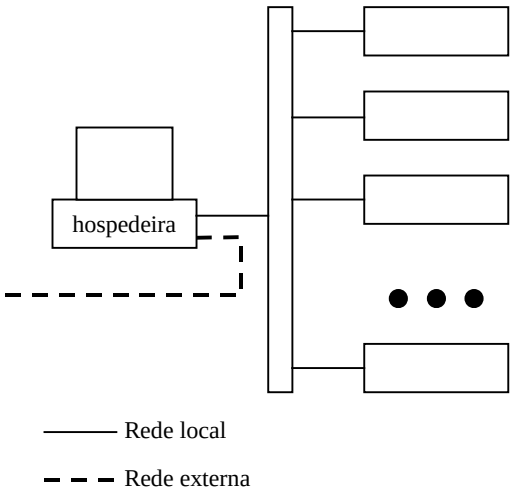


Figura 3 – Estrutura de um HPC

Nos clusters HPC, conforme a Figura 3, são encontradas máquinas que realizam diferentes funções e, portanto, devem possuir hardware e software diferentes. Uma primeira divisão pode ser visualizada entre máquinas que oferecem serviços (servidores) das máquinas que realizam o processamento (computação). As máquinas de computação são as máquinas que ficam localizadas junto a rede local do cluster e não possuem acesso direto a rede externa. Convém salientar que a rede local do cluster em muitos casos pode dividir-se em duas (ou mais): uma para controle e outra para trabalho. Esta divisão proporciona um melhor desempenho e facilidades para o monitoramento das atividades do cluster.

As máquinas de computação são as máquinas responsáveis pelo efetivo processamento das aplicações paralelas no agregado. O poder de processamento de um cluster é resultado da capacidade computacional das máquinas de processamento, portanto elas devem ser configuradas com um mínimo de serviços sendo executados (apenas funções básicas do SO) para poderem se dedicar ao processamento das aplicações. Porém, elas necessitam de diversos serviços fornecidas pelas máquinas servidoras. Dentre as facilidades geralmente disponibilizadas nos servidores dos clusters destacam-se:

- Disco;
- Login;
- Monitoramento;
- Comunicação.

Servidores de arquivos compartilhados são encontrados nos clusters para facilitar a execução das aplicações paralelas além de possibilitar que o conjunto de dados de entrada a ser processado fique disponível a todas as máquinas de computação. Agregados com um grande número de processadores podem dedicar máquinas e dispositivos específicos para servirem disco ao cluster, porém geralmente o servidor de arquivos é compartilhado com algum outro serviço, disponibilizado através de *Network File System* (NFS) na máquina hospedeira.

Os servidores de login são encontrados para possibilitar o acesso ao cluster uma vez que as máquinas de computação não são acessíveis diretamente. Em muitos casos, a máquina hospedeira é o único servidor de login, mas isso não é uma regra, podendo o cluster possuir vários servidores de login aumentando as opções de acesso ao cluster. Uma das funções do servidor de login que muitas vezes passa despercebida é a unificação dos *User Identification* (UID) e a senha dos usuários em todas as máquinas do cluster, uma vez que 'n' máquinas significam várias imagens distintas de SO, podendo assim um usuário possuir UID e senha diferente nas máquinas, o que dificultaria a utilização do cluster. Para resolver este problema pode ser utilizado o *Network Information Service* (NIS) que unifica senha e UID em todas as máquinas, mantendo as informações atualizadas e consistentes. Este serviço deve ser único no sistema e pode ser utilizada uma máquina dedicada para esta função ou não.

A execução das aplicações no agregado pode demorar horas, ou mesmo semanas. Este fato aliado a problemas externos ou internos a aplicação que podem ocorrer durante a execução forçam a necessidade da utilização de estações específicas para o monitoramento das atividades do cluster. Dependendo do número de máquinas e taxa de utilização podem ser destinadas estações exclusivas para este propósito ou pode ser utilizada a máquina hospedeira, fato comum em agregados compostos de poucas máquinas (geralmente menos que 64). A tendência é que as funções de monitoramento e algumas de acesso sejam disponibilizadas através da internet, dedicando uma máquina para a realização desta função (BIAZUS NETO; REBONATTO; BRUSSO, 2004, p. 287).

Menos comum nos agregados atuais são as máquinas dedicadas a comunicação das máquinas de computação. Este fato ocorre em grandes clusters onde a rede de interconexão deve ser dividida em vários segmentos, sendo as máquinas dedicadas a comunicação responsáveis pelo intercâmbio de informações entre os diversos segmentos. Geralmente a comunicação é disponibilizada através de barramento ou chaveamento (*switch*), que pode possuir vários níveis mas sem a necessidade de máquinas específicas para seu controle.

Convém lembrar que os serviços descritos não são obrigatórios. O mais comum é o compartilhamento de disco e a unificação da identificação dos usuários, disponibilizados na máquina hospedeira.

As aplicações paralelas são executadas a partir de uma máquina, geralmente a hospedeira, e a partir deste momento, a execução deve 'espalhar-se' pelas máquinas do cluster. Para isso, um serviço necessário que deve ser disponibilizado por todas as máquinas do agregado de computadores é a execução remota de aplicações. Esta funcionalidade pode ser obtida com o *Remote Shell* (RSH) que deve estar habilitado em todas as máquinas que irão executar as aplicações.

Programação paralela

Na busca por um ganho de desempenho na execução de aplicações (tipicamente redução do tempo de execução), o software tem papel importante, pois é ele que irá comandar o hardware durante o processamento. No caso de grande parte das arquiteturas paralelas, entre elas os agregados de computadores, o desenvolvimento de aplicações paralelas é fundamental, pois caso contrário (se continuarmos a desenvolver aplicações seqüências) o paralelismo disponível em nível de hardware não será explorado.

Para o desenvolvimento de aplicações paralelas, pode-se optar por uma das seguintes alternativas:

- Exploração implícita do paralelismo;
- Exploração explícita do paralelismo.

A exploração implícita do paralelismo é empregada com a utilização de ferramentas automáticas (tipicamente compiladores paralelizantes) que automatizam a criação de binários paralelos. A escrita dos programas pode seguir a mesma metodologia da programação seqüencial, sem a necessidade do programador explicitamente declarar os pontos onde o paralelismo será explorado e nem como isso será realizado. O compilador irá automaticamente detectar os pontos onde é possível uma execução paralela e gerar um código paralelo. Esta técnica é altamente dependente da arquitetura de hardware utilizada (o código paralelo gerado para a máquina A pode não funcionar na máquina B) e da disponibilidade de uma ferramenta para paralelização automática para a linguagem de programação utilizada. A principal vantagem é um possível reaproveitamento de códigos seqüências já existentes. Hoje em dia existem compiladores paralelizantes basicamente para as linguagens Fortran e C, disponibilizadas para algumas arquiteturas de máquinas paralelas.

A exploração implícita do paralelismo não pode ser considerada uma solução definitiva para a programação paralela pelas seguintes razões (CENAPAD, 2001, p. 5).

- Pode não ocorrer à exploração total do paralelismo: as ferramentas tendem a atuar sobre partes restritas do código fonte e a não resolver problemas de dependências de dados, gerando falsas positivas;
- Não são aplicáveis a qualquer tipo de aplicação: nem todas as aplicações são passíveis de paralelização automática;
- Geram código geralmente menos eficientes: no estágio atual, estas ferramentas não conseguem superar a atividade humana (exploração explícita do paralelismo).

Por outro lado, a exploração explícita do paralelismo força que o programador especifique os pontos onde o paralelismo será explorado e como esta tarefa será realizada. O conjunto destas ações é denominada de programação paralela de aplicações (alguns autores preferem o termo programação concorrente, porém para evitar confusões com textos da área de sistemas operacionais, será adotado unicamente o termo programação paralela). Ela possibilita a escrita de programas com vários fluxos de execução isolados, que deverão ser executados de forma independente e concomitantemente (CAVALHEIRO, 2001, p. 35).

Na programação paralela, preocupações que não eram o centro das atenções na programação seqüencial devem merecer uma maior destaque. São elas:

- Fluxos de execução diferentes: na programação seqüencial existe apenas um fluxo de execução ativo em um determinado momento enquanto que na paralela podem existir vários;
- Dependências de dados: com mais de um fluxo de execução, inclui-se uma possibilidade de um dado não estar pronto para utilizá-lo num determinado momento (ele pode estar sendo processado em outro fluxo de execução). Na programação seqüencial isto é facilmente detectável, porém na paralela não;
- Cooperação: um dos princípios do processamento paralelo é a cooperação entre todas as unidades processadoras para a resolução dos problemas. Esta cooperação deve ser explicitada nos programas;
- Comunicações e sincronizações: a cooperação entre os processos depende de comunicações e sincronizações entre eles.

Este acréscimo de cuidados torna a programação paralela ainda mais gratificante quando dominada, pois com a sua compreensão, novos horizontes podem ser abertos em muitas áreas. Para explicitar o paralelismo em programas, são utilizadas diversas formas de anotação, entre elas destacam-se (SATO; MIDORIKAWA; SENGER, 1999, p. 6):

- Diretivas ao compilador;
- Bibliotecas para exploração do paralelismo;
- Linguagens paralelas.

Na utilização de diretivas ao compilador, o trabalho de paralelização é dividido entre o compilador e o programador. O programador diz onde poderá ocorrer execução em paralelo e o compilador gera um código para a execução paralela. Sua principal vantagem é a divisão do trabalho. Ela possui como desvantagem a necessidade do compilador reconhecer as diretivas especiais e ter capacidade de paralelização, desta forma a sua utilização é pouco difundida.

A escrita de programas paralelos com linguagens paralelas é apontada por muitos como o caminho ideal da programação paralela. Sua principal desvantagem é a reduzida utilização das linguagens devido ao baixo número de arquiteturas de hardware disponíveis para sua execução.

O emprego de linguagens de programação tradicionais (C e Fortran) com a utilização de bibliotecas para a exploração do paralelismo é a opção mais difundida atualmente. A grande portabilidade das bibliotecas entre as mais diversas arquiteturas é um grande trunfo para sua elevada utilização. Diversas bibliotecas para a exploração do paralelismo estão disponíveis, merecendo destaque a *Parallel Virtual Machine* (PVM) e o MPI, que será detalhado no decorrer deste texto.

Organização do texto

A programação paralela em agregados de computadores usando o MPI é o tema principal deste texto. Dessa forma o MPI será alvo da próxima seção, incluindo suas versões, a manipulação de sua máquina virtual paralela juntamente com seus principais aplicativos. Na terceira e última seção, será abordada a escrita de programas paralelos com o MPI, descrevendo as principais funções disponibilizadas (será utilizada a interface com a linguagem C, padronizado pela *American National Standards Institute* – ANSI), incluindo programas exemplo.

MPI

Grande parte das primeiras gerações de máquinas paralelas utilizava a arquitetura de trocas de mensagens, uma vez que ela possuía um menor custo se comparada com a arquitetura de memória compartilhada (multiprocessadores). Neste contexto foram desenvolvidas várias bibliotecas distintas para exploração do paralelismo usando trocas de mensagens. Assim, os fabricantes de máquinas paralelas disponibilizavam bibliotecas para trocas de mensagens que funcionavam corretamente em seu hardware, porém eram incompatíveis com as máquinas de outros fabricantes. Muitas das diferenças encontradas nas rotinas para trocas de mensagens eram apenas na sintaxe; entretanto freqüentemente essas diferenças causavam sérios problemas no porte de uma aplicação paralela de uma biblioteca para outra. O MPI foi criado essencialmente para resolver este problema, definindo um conjunto padrão de rotinas para trocas de mensagens que pode ser utilizado para escrever um programa paralelo portável utilizando C, C++ ou Fortran (GRAMA et. al., 2003).

O padrão MPI foi concluído no final do primeiro semestre de 1994 (versão 1.0) e atualizado na metade do ano seguinte (versão 1.1) (CENAPAD, 2001, CÁCERES; SONG, 2004, MESSAGE PASSING INTERFACE FORUM, 2004). A versão 2 teve sua preliminar apresentada no *SuperComputing´96*, e em abril de 1997 o documento MPI-2 foi unanimemente votado e aceito (IGNÁCIO & FERREIRA FILHO, 2002).

O MPI é resultado de um esforço da comunidade para definir e padronizar a sintaxe e semântica de uma biblioteca de rotinas para trocas de mensagens que pudesse ser implementada numa ampla variedade de máquinas paralelas (CÁCERES; SONG, 2004). Alguns autores apontam sua utilização e aceitação atual ser devida a colaboração dos membros que constituíram o MPI Fórum. O MPI Fórum foi aberto, constituído por pesquisadores, acadêmicos, programadores, usuários e fabricantes, representando aproximadamente 40 organizações. Entre os participantes do fórum pode-se destacar:

- Representantes da Convex, Cray, IBM, Intel, Meiko, nCUBE, NEC e Thinking Machines;
- Membros de grupos de desenvolvedores de software tais como: PVM, P4, Zipcode, Chamaleon, PARMACS, TCGMSG e Express;
- Especialistas na área de processamento paralelo (CENPAD, 2001, p.12).

O MPI é uma das bibliotecas para exploração do paralelismo mais difundidas, padrão para a comunicação paralela com trocas de mensagens em agregados de computadores. Ele é largamente utilizado pela disponibilidade em grande número das máquinas paralelas atuais. Segundo Cáceres e Song, o MPI possui várias razões para ser a cada dia mais utilizado. São elas (CÁCERES; SONG, 2004, p. 32):

- O MPI possui mais de uma implementação de boa qualidade disponível gratuitamente;
- Os grupos de comunicação do MPI são sólidos, eficientes e determinísticos;
- O MPI gerencia eficientemente os *buffers* de mensagens;
- O MPI pode ser utilizado eficientemente para programar uma ampla variedade de arquiteturas de máquinas paralelas, incluindo multiprocessadores;
- O MPI é totalmente portável;
- O MPI é fortemente especificado.

Ainda que possua várias implementações, elas conseguem apresentar um bom desempenho uma vez que somente a lógica das operações é especificada. A implementação fica a cargo de desenvolvedores que otimizam o código de acordo com o hardware a ser portado.

As implementações oferecem segurança para o programador, pois a interface de comunicação projetada é confiável, não havendo a necessidade de preocupação em relação a falhas na comunicação. Esta interface também gerencia de maneira transparente os *buffers* de envio e recepção de mensagens, dividindo e recuperando as mesmas quando necessário.

Ao incorporar grupos de comunicação eficientes, seguros e determinísticos o MPI adiciona escalabilidade as suas características. Ela pode ser obtida com a criação de grupos e sub-grupos de processos que resultarão em comunicações coletivas com melhor desempenho.

A facilidade da programação em uma ampla variedade de máquinas é devida em parte a sua interface de programação, não muito diferente dos padrões existentes quando da sua criação como PVM e P4. Esta interface foi complementada com extensões que permitem uma maior flexibilidade.

A disponibilização de diversas implementações que funcionam em arquiteturas distintas, conduz o MPI a um beneficio adicional: a transparência. Uma aplicação desenvolvida num sistema homogêneo pode ser facilmente executado num sistema com diferente arquitetura e até mesmo heterogêneo, contendo máquinas com diferentes arquiteturas (CÁCERES; SONG, 2004, CENAPAD, 2001).

Implementações

Existem implementações especialmente desenvolvidas para famílias de computadores, computadores específicos e versões disponibilizadas por universidades e centros de pesquisa que funcionam em diversos tipos de máquinas. As principais implementações que funcionam em várias plataformas de hardware são:

- MPICH: *Argonne National Laboratory*/*Mississipi State University*;
- CHIMP: *Edinburgh Parallel Computing Centre*;
- PMPIO: *National Aeronautics and Space Administration* – NASA;
- UNIFY: *Mississippi State University;*
- MPI/Pro: MPI *Software Technology;*
- LAM: *Ohio Supercomputer Center* (ARGONNE NATIONAL LABORATORY, 2004, CENAPAD, 2001).

O MPICH é utilizado pela comunidade acadêmica, científica e comercial. Seu código serve de base para inúmeras outras implementações, disponibilizando versões para plataformas de comunicação e sistemas operacionais únicos, incluindo também versões próprias para famílias de computadores e máquinas específicas. Entre as versões disponíveis para plataformas de comunicação, destacam-se o SCI-MPICH e o MPI-BIP, enquanto que o Winmpich, Nt-mpich, WMPI II, WMPI 1.5 e PaTENT MPI funcionam em ambientes Win-NT e win32. As máquinas NEC, tais como SX-3, SX-4, SX-5 e SX-6 dispõem do MPI/SX (RITZDORF, 2004a) implementado a partir do MPICH de acordo com as características de hardware específicas desta família de computadores. Na linha de maquinas específicas, destaca-se o MPI/ES (RITZDORF, 2004b), disponível para o *Earth Simulator*, máquina com melhor índice de desempenho no 22ª edição do ranking das top500 (TOP500 SUPERCOMPUTER SITES, 2003), versão criada a partir do MPI/SX. Convém lembrar a máquina *Earth Simulator* é a primeira colocada desde a 18ª edição do ranking.

Seguindo a linha das versões para famílias de computadores e máquinas específicas, destacam-se as seguintes, não necessariamente produzidas a partir do MPICH.

- IBM MPI: implementações da IBM específicas para clusters, máquinas SP e OS/390;
- CRAY MPI: disponibilizada pela *Cray Research* e o *Edinburgh Parallel Computing Centre*;
- SGI MPI: disponível para máquinas SGI mips3 e mips4;
- Digital MPI: funciona em máquinas da *Digital Equipment Corporation*;
- HP MPI: disponível para a linha de máquinas da HP;
- MPI para Macintosh G3 clusters;
- MPI para *Fujitso* AP1000: produzido pela *Australian National University*.

A versão UNIFY fornece um conjunto de funções MPI dentro do ambiente PVM, sem abrir mão das rotinas do PVM já existentes. O MPI/Pro é uma versão comercial vendido pela MPI *Software Technology, Inc.*, disponível para clusters comercias, máquinas NT e outras implementações de multicomputadores (ARGONNE NATIONAL LABORATORY, 2004).

O *Local Area Multicomputer* (LAM) é uma versão que foi desenvolvida para funcionar tanto em clusters como em estações de trabalho ligadas através de rede (máquinas NOW). Dessa forma, disseminou-se rapidamente mesmo quando o número de máquinas agregadas e dedicadas ao processamento paralelo e distribuído era pequeno. Essa será a versão que será utilizada nas demais referências e exemplos no decorrer do texto, sendo executada através do SO Linux.

Convém salientar que a lista disponibilizada neste texto não abrange todas as versões e implementações do MPI. Apenas serve para dar noção da abrangência deste padrão para comunicação através de trocas de mensagens. Outras empresas, universidades, centros de pesquisa e fabricantes disponibilizam, e estão criando novas versões para as mais diferentes arquiteturas de máquinas. Outra lista versões pode ser obtida em LAM/MPI PARALLEL COMPUTING, 2004.

O mundo do MPI

O MPI define várias padronizações necessárias a programação paralela de aplicações utilizando trocas de mensagens. As rotinas disponibilizadas podem ser agrupadas de acordo com suas finalidades específicas em três grandes grupos (COSTA; STRINGHINI; CAVALHEIRO, 2002, p. 56, GEYER, 2003):

- Gerência de processos;
- Rotinas de comunicação ponto a ponto;
- Rotinas para a comunicação de grupos (incluindo manutenção).

A gerência de processos é realizada através da manipulação do que é conhecido como ‘máquina virtual paralela’, base para a execução de aplicativos paralelos utilizando o MPI.

O conjunto de rotinas disponibilizado possui interface para as linguagens C, C++ e Fortran. A biblioteca de rotinas pode ser igualmente divida nos três grupos acima citados, incluindo no primeiro grupo a identificação dos processos juntamente com rotinas para o início e fim dos processos criados pelo MPI.

Mesmo com uma quantidade de versões implementadas, o MPI mantém um alto grau de portabilidade entre as aplicações desenvolvidas. Segundo CENAPAD, 2001 p. 15 "o MPI pode variar a quantidade de funções, a forma de inicialização de processos, a quantidade de cada sistema de '*buffering*', os códigos de retorno e erro, de acordo com os desenvolvedores de cada implementação".

Máquina virtual paralela

O conceito utilizado pelo MPI para a execução de aplicações paralelas em multicomputadores é a criação de uma máquina virtual paralela composta dos recursos computacionais dos computadores que compõem a máquina física em questão. Por exemplo, no caso de um agregado de computadores, composto de 16 máquinas INTEL monoprocessadas, poderá ser composta uma máquina virtual paralela de 16 processadores. No caso da memória, a organização em forma de multicomputador não possibilita seu compartilhamento. Dessa forma, o fato da memória estar fisicamente distribuída entre as máquinas do cluster não causará problema algum: o conjunto das memórias pertencentes às máquinas físicas que compõem o agregado também poderá ser considerada parte da máquina virtual paralela.

Após a máquina virtual paralela ser configurada e estar operante, deixa de existir um conjunto de máquinas interconectadas através de alguma rede passando a estar disponível uma máquina paralela (virtual), também denominada neste texto de "ambiente MPI". As aplicações paralelas escritas com o MPI são executadas nessa máquina virtual (ambiente MPI).

O LAM/MPI oferece diversos aplicativos para a manipulação de sua máquina virtual paralela. A Tabela 1 ilustra alguns desses aplicativos.

Tabela 1: Aplicativos para a manipulação de máquina virtual paralela em LAM/MPI

Aplicativo	Função
lamboot	Inicializa a máquina virtual
recon	Verifica a disponibilidade das máquinas físicas
lamnodes	Mostra o status atual da máquina virtual
lanclean	Desmonta a máquina virtual
wipe	Elimina a máquina virtual

O *lamboot* é o aplicativo que torna o ambiente MPI ativo. Na prática, ele inicializa nas máquinas componentes da máquina virtual um *daemon* ('*lamd*', um em cada) que aloca uma porta para a comunicação. As informações de porta e máquina são distribuídas as demais máquinas físicas formando uma rede totalmente conectada (CAVALHEIRO, 2001, p. 44).

Para a utilização do *lamboot*, se deve especificar quais são as máquinas físicas que irão compor o ambiente MPI. Isso pode ser realizado através de um arquivo texto, contendo em cada linha a identificação de uma máquina. A Figura 4 ilustra um arquivo de máquinas físicas (*hosts*) que podem um ambiente MPI.

```
$cat hosts

dunga
taffarel
falcao
valdomiro
figueroa
manga
```

Figura 4 – Arquivo com especificação das máquinas que irão compor uma máquina virtual paralela em LAM/MPI

Pode-se substituir a identificação do arquivo pelo seu endereço IP (nas redes que utilizam TCP/IP). Por padrão, é alocado um processador de cada máquina, porém na utilização de máquinas multiprocessadas para a composição do ambiente MPI, deve-se incluir após a identificação da máquina os argumentos "CPU=x", onde x indica o número de unidades processadoras a serem utilizadas na máquina em questão.

A sintaxe para o *lamboot* é simples, deve-se apenas indicar um arquivo texto contendo as máquinas físicas que irão o ambiente MPI. A Figura 5 ilustra uma execução deste comando.

```
$ lamboot -v hosts

LAM 6.5.9/MPI 2 C++/ROMIO - Indiana University

Executing hboot on n0 (dunga - 1 CPU)...
Executing hboot on n1 (taffarel - 1 CPU)...
Executing hboot on n2 (falcao - 1 CPU)...
Executing hboot on n3 (valdomiro - 1 CPU)...
Executing hboot on n4 (figueroa - 1 CPU)...
Executing hboot on n5 (manga - 1 CPU)...
topology done
```

Figura 5 – Execução do comando *lamboot*

Para uma correta execução, como no caso da Figura 5, todas as máquinas constantes do arquivo '*hosts*' devem estar operantes. Note-se que o parâmetro "-v" (*verbose*) foi adicionado para que maiores informações sobre o comando pudessem ser visualizadas. Outros argumentos são suportados e podem ser obtidos através da leitura das páginas do manual (comando *man*).

O *recon* serve para que seja realizado um teste e verificar se todas as máquinas que farão parte do ambiente MPI estão ativas e operantes. De forma análoga ao *lamboot*, ele utiliza um arquivo contendo a identificação das máquinas que devem ser verificadas. A Figura 6 ilustra a execução desse comando.

```
$ recon -v hosts

recon: -- testing n0 (dunga)
recon: -- testing n1 (taffarel)
recon: -- testing n2 (falcao)
recon: -- testing n3 (valdomiro)
recon: -- testing n4 (figueroa)
recon: -- testing n5 (manga)
-----------------------------------------------
Woo hoo!
. . .
-----------------------------------------------
```

Figura 6 – Execução do comando *recon*

O *recon* é geralmente utilizado antes do *lamboot* para verificar se todas as máquinas físicas estão aptas a integrarem a máquina virtual paralela. Essa verificação evita esperas desnecessárias com a utilização do *lamboot* quando uma das máquinas não está operante (por falha, sem energia, não conectada a rede, ...). Os pontinhos entre as duas linhas indicam a presença de maiores informações relativas a execução do comando. A correta execução do *recon* indica que a máquina virtual paralela esta apta a ser inicializada, mas não garante sua correta montagem.

Uma vez que a montagem do ambiente MPI pode ser realizada passando como argumento as máquinas físicas que irão compô-lo, diferentes configurações de máquinas virtuais paralelas podem coexistir num mesmo agregado de computadores. Pode-se, por exemplo, ter máquinas virtuais com 8 e com 16 processadores num cluster com 16 máquinas monoprocessadas. Dessa forma, mostrar a configuração atual do ambiente MPI é função do comando *lamnodes*. A Figura 7 ilustra sua execução.

```
$ lamnodes

n0      dunga.localdomain:1
n1      taffarel.localdomain:1
n2      falcao.localdomain:1
n3      valdomiro.localdomain:1
n4      figueroa.localdomain:1
n5      manga.localdomain:1
```

Figura 7 – Execução do comando *lamnodes*

O *lamnodes* é geralmente utilizado quando o usuário faz *login* no cluster, e deseja saber se sua máquina virtual paralela está ativa. Sua utilização se deve ao fato de que geralmente após o usuário desconectar-se do *front-end* do cluster, suas configurações do ambiente MPI são mantidas, a menos que o agregado de computadores seja reinicializado.

A máquina paralela virtual é desativada quando desliga-se o cluster ou quando utiliza-se o comando *wipe*. Ele elimina todas as ligações de comunicação do ambiente MPI. Após sua execução, os *daemons* inicializados nas máquinas deixam de funcionar e não é mais possível a execução de aplicações paralelas. A Figura 8 ilustra a execução do comando *wipe*.

```
$ wipe -v hosts

LAM 6.5.9/MPI 2 C++/ROMIO - Indiana University

Executing tkill on n0 (dunga)...
Executing tkill on n1 (taffarel)...
Executing tkill on n2 (falcao)...
Executing tkill on n3 (valdomiro)...
Executing tkill on n4 (figueroa)...
Executing tkill on n5 (manga)...
```

Figura 8 – Execução do comando *wipe*

Conforme demonstrado na Figura 8, a sintaxe do *wipe* é similar a do *lamboot*, sendo necessário indicar o arquivo contendo a identificação das máquinas físicas (Figura 4) que compõem a máquina virtual paralela. Caso no momento da execução do *wipe* alguma aplicação paralela estiver sendo executada, ela também será finalizada (CAVALHEIRO, 2001, p. 45). A execução do *lamboot* ou do *wipe* sem a indicação do arquivo que contém as máquinas físicas leva a uma execução apenas na máquina local, ou seja, ativa ou desativa um ambiente MPI com apenas uma máquina física.

O *lamclean* serve para eliminar apenas as aplicações paralelas que estão sendo executadas sobre o ambiente MPI, mantendo a máquina virtual paralela operante. Sua sintaxe é semelhante a do comando *lamnodes*, não sendo necessário argumentos, porém diferente do *lamnodes* sua execução normal não gera informações na saída. O *lamclean* é geralmente empregado na fase de implementação e validação das aplicações paralelas. Durante este estágio de desenvolvimento, problemas com as aplicações podem ser constantes. Ele possibilita a parada de aplicações que porventura estejam com problemas, possibilitando assim novos testes com as correções realizadas.

Compilação de programas

Grande parte das implementações, disponibiliza facilidades para a compilação e linkedição dos programas fonte escritos com o MPI. Caso a versão não disponibilize ferramentas para automatizar este processo deve-se incluir manualmente as bibliotecas do MPI utilizando um compilador padrão. Um *script* geralmente disponibilizado pelas versões é o *mpicc* (disponível em LAMMPI e MPICH) utilizado para compilação de programas fonte em C/C++. A Figura 9 ilustra o formato deste comando juntamente com exemplos.

```
mpicc {fonte.c} (–o binário) [parâmetros]    (a)

-----------------------------------------------

mpicc primo.c –o primo                       (b)

-----------------------------------------------

mpicc raiz.c –o raiz –lm                     (c)

Legenda
{   } Obrigatório                [   ] Opcional
(   ) Opcional, porém recomendado
```

Figura 9 – Utilização do *mpicc*

Na Figura 9a, o esquema geral de utilização do *script mpicc* é descrito, onde deve-se obrigatoriamente identificar o arquivo fonte a ser compilado. Após, pode-se indicar o binário resultante do processo de compilação e linkedição, colocado na seqüência o argumento '-o'. Este parâmetro não é obrigatório, mas sua utilização é extremamente recomendada, uma vez que a sua omissão irá gerar o arquivo de saída padrão (geralmente o arquivo 'a.out'). Ao final, podem ser passados ao *mpicc* todos os argumentos do compilador C disponível (geralmente o *gcc* é o compilador C disponível em agregados de computadores com SO linux).

Um exemplo de utilização do *mpicc* encontra-se na Figura 9b, onde a partir do arquivo fonte 'primo.c' está sendo gerado o binário 'primo'. Outro exemplo pode ser visualizado na Figura 9c, onde o parâmetro "–lm" indica que a biblioteca matemática deverá ser incorporada no processo de geração do binário 'raiz' a partir do arquivo fonte 'raiz.c'.

Execução de aplicações

Após a criação dos binários, as aplicações paralelas escritas com MPI podem ser executadas, desde que o ambiente MPI (máquina virtual paralela) esteja operacional. As distribuições do MPI oferecem *scripts* que possibilitam a execução das aplicações sobre a máquina virtual paralela, sendo o *mpirun* um dos mais populares.

Caso um binário gerado (escrito + compilado) com as bibliotecas do MPI seja colocado em execução sem o ambiente MPI estar operacional, a aplicação não será executada. Outra inconsistência que pode ocorrer na execução de aplicações escritas com MPI, mesmo com a máquina virtual paralela operante, é a não utilização de um *script* como o *mpirun* ou semelhante. Nesse caso, a aplicação irá ser executada gerando apenas um processo, não havendo execução paralela, mesmo que o programa seja escrito e compilado com essa finalidade. A Figura 10 ilustra a sintaxe e exemplos do comando *mpirun*.

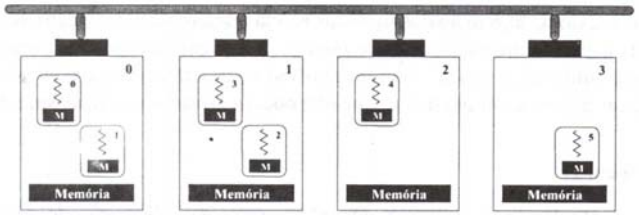
```
mpirun [parâmetros] {-np x binário}
       [parâmetros do binário]                    (a)

------------------------------------------------

mpirun –np 6 primo                                (b)

------------------------------------------------

mpirun –O –np 4 raiz 5 1234567                    (c)

Legenda
{   } Obrigatório                  [   ] Opcional
```

Figura 10 – Utilização do *mpirun*

A Figura 10a ilustra o esquema geral de utilização do *script mpirun* na execução de aplicações paralelas, onde se deve obrigatoriamente indicar o nome do binário a ser executado e o número de processos a serem criados através do parâmetro '-np x', onde 'x' é o número de processos a serem criados. Pode-se opcionalmente indicar parâmetros extras para a execução do MPI como também informar argumentos a aplicação sendo executada através da linha de comando. Diversas são as opções do MPI que podem ser incluídas na execução das aplicações, sendo utilizada a opção '-h' para visualizar as disponíveis ou utilizar o comando *man*.

Um exemplo de utilização do *mpirun* está descrito na Figura 10b, onde o binário 'primo' esta sendo executado com 6 processos. A Figura 10c ilustra outro exemplo, onde o binário raiz é executado com 4 processos em uma máquina paralela formada de máquinas físicas homogêneas, indicada pelo argumento '-O'. Na Figura 10c ainda pode ser visualizado a utilização de argumentos de linha de comando para o binário 'raiz', sendo manipulados dentro da aplicação o `argc` que é igual a 3, e `argv[0]`, `argv[1]` e `argv[2]` que valem, respectivamente, 'raiz', '5' e '1234567'.

O *mpirun* solicita a execução remota dos binários nas máquinas que compõem a máquina virtual paralela. Por exemplo, a utilização do comando ilustrado na Figura 10b, sobre uma máquina virtual paralela composta de 4 máquinas reais resulta no esquema de execução semelhante ao ilustrado na Figura 11.



Fonte: CAVALHEIRO, 2004, p.28.

Figura 11 – Execução de aplicação sobre máquina virtual paralela.

Na Figura 11 se pode identificar as máquinas físicas do agregado, representadas por retângulos, com sua área própria de memória (não compartilhada) e interligadas através de uma rede de interconexão em sua parte superior. Os processos sendo executados podem ser visualizados através de retângulos com as pontas arredondadas, cada um contendo seu próprio fluxo de execução e sua área de memória. Como a aplicação foi executada com 6 processos num agregado composto de quatro máquinas físicas, duas das máquinas possuem dois processos em execução. Isto é possível, pois o *mpirun* cria os processos nas máquinas físicas que compõem o cluster de forma circular, de acordo com a disposição das máquinas no arquivo que foi utilizado para a criação do ambiente MPI (COSTA; STRINGHINI; CAVALHEIRO, 2002, p.56). Um exemplo de arquivo para criação de um ambiente MPI pode ser visualizado na Figura 4, porém nesse exemplo, são utilizadas seis máquinas físicas ao invés de quatro (utilizada no exemplo da Figura 11).

PROGRAMAÇÃO PARALELA COM MPI

A escrita de aplicações com MPI segue o modelo de trocas de mensagens, onde processos são executados em espaços de endereçamento de memória distintos (multicomputadores). A comunicação ocorre quando uma região de memória de um processo é copiada para outro. Esta operação é realizada quando um processo executa uma operação de envio (*send*) e o outro executa uma de recebe (*receive*) o conteúdo de memória a ser comunicado (GROPP; LUSK; SKJELLUM, 1999, p. 13).

Nesse contexto, uma aplicação paralela escrita com MPI é constituída de um número fixo de processos que se comunicam através de rotinas do tipo envia/recebe para o compartilhamento

dos dados. Cada um destes processos pode executar um fluxo de execução diferente do outro, caracterizando um modelo de execução com *Multiple Program Multiple Data* – MPMD que significa múltiplos programas executando sobre múltiplos dados (IGNÁCIO; FERREIRA FILHO, 2002, p. 107). Na prática, utiliza-se disparar um único programa gerando múltiplos processos onde cada processo irá executar uma porção delimitada do código. Esse modelo é conhecido como *Sample Program Multiple Data* – SPMD (um programa sobre múltiplos dados) e segundo COSTA; STRINGHINI; CAVALHEIRO é "uma forma natural de programar" aplicações MPI (2002, p. 56).

O MPI disponibiliza um conjunto de funções e estruturas de dados para proporcionar a programação de aplicações paralelas. Com o objetivo de facilitar a visualização, o MPI utiliza um prefixo em cada um dos seus componentes de software. Eles são pré-fixados com "MPI_", tendo após o *underscore* a primeira letra em maiúscula. Dessa forma, na leitura e interpretação dos programas fonte são salientados os pontos onde alguma funcionalidade do MPI é empregada.

O MPI possui aproximadamente 120 funções (pode variar de acordo com a implementação), porém o número de seus conceitos chaves é muito menor. Para compreender o funcionamento da escrita de aplicações em MPI deve-se compreender:

- Processos MPI;
- Comunicadores;
- Mensagens.

A execução das aplicações escritas com MPI ocorre com o disparo em paralelo de seus processos. Segundo TANEMBAUN; WOODHULL, "um processo é basicamente um programa em execução" (2000, p.26), e no caso de aplicações que usam o MPI, vários processos podem estar em execução ao mesmo tempo. Geralmente, os processos MPI são disparados com o auxílio do script *mpirun* (Figura 10).

O S.O. é encarregado de atribuir uma identificação aos processos em execução. Num cluster onde existem vários S.O.s (podem até mesmo ser homogêneos, porém não é um único S.O.) a identificação dos processos MPI em execução seria dificultada. O *rank* é a solução implementada pelo MPI para fornecer uma identificação direta e simples de cada processo. Ele é único e individual em cada processo MPI, tendo seu valor atribuído no inicio da execução da aplicação paralela. Os *ranks* de uma aplicação paralela variam de 0 até N-1, onde N é o número de processos com que a aplicação foi executada (COSTA; STRINGHINI; CAVALHEIRO, 2002, p. 56), indicado pelo parâmetro "–np" do *mpirun*, conforme ilustrado na Figura 10.

Os processos MPI organizam-se em grupos ordenados para possibilitar uma comunicação eficiente. Para possibilitar a comunicação entre os processos do grupo, o MPI introduz o conceito de comunicadores, que nada mais são do que "um objeto local que representa o domínio (contexto) de uma comunicação (conjunto de processos que podem ser conectados)" (CENAPAD, 2001, p. 16). Por padrão, todos os processos fazem parte de um grupo único associado ao comunicador pré-definido identificado por MPI_COMM_WORLD.

Para a comunicação dos processos através dos comunicadores o MPI definiu um formato de mensagens. Elas podem ser enviadas ou recebidas e são compostas de duas partes: dados e envelope. A parte dos dados corresponde ao endereço de memória a ser enviado/recebido, o tipo do dado sendo comunicado e a quantidade de elementos do dado na mensagem. O envelope deve conter a identificação do processo (a receber ou para quem enviar, utilizando para isso o *rank* do processo), o assunto da mensagem (em MPI definido por *tag*) e o comunicador associado (PACHECO, 1997). A Figura 12 ilustra a composição das mensagens em MPI.

<table>
<tr><td colspan="6">Mensagem</td></tr>
<tr><td colspan="3">Envelope</td><td colspan="3">dado</td></tr>
<tr><td>origem
destino</td><td>tag</td><td>comunicador</td><td>endereço</td><td>quantidade</td><td>tipo</td></tr>
</table>

Figura 12 – Composição das mensagens em MPI

Um atributo interessante do envelope das mensagens é a utilização do *tag* (assunto). Com ele, pode-se rotular as mensagens, viabilizando a implementação de filtros na recepção (CAVALHEIRO, 2001, p. 47). No dado, a opção de indicar uma quantidade a ser enviada/recebida facilita a movimentação de grandes áreas de memória, como vetores e matrizes, além de possibilitar o envio de uma variável isolada, indicando na quantidade o valor 1 (um).

Os tipos de dados utilizados nas mensagens MPI não são tipos padrão da linguagem de programação utilizada. Ao invés, o MPI proporciona tipos próprios de dados para permitir a utilização de máquinas com arquiteturas diferentes na composição de sua máquina virtual paralela. Esse fato é devido a uma implementação não homogênea dos tipos de dados na grande variedade de máquina as quais o MPI é portado. A Tabela 2 ilustra os tipos de dados utilizados no MPI e sua relação com os disponíveis na linguagem C.

Tabela 2: Tipos de dados do MPI relacionados aos fornecidos pela linguagem C

Tipos de dados do MPI	Tipos de dados em 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	unisigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Fonte: PACHECO, 1997, p. 48

A tabela 2 relaciona os tipos MPI relacionados com os tipos padrão da linguagem C. Os dois últimos, MPI_BYTE e MPI_PACKED não possuem correspondente direto na linguagem

C. O primeiro, poderia ser relacionado com um *unsigned char*, onde seriam manipulados apenas valores numéricos. O segundo tipo é útil quando se faz necessária a comunicação de dados não continuamente agrupados na memória, uma vez que quando informa-se uma quantidade de elementos, eles por padrão, devem estar em área contínua na memória.

Os tipos ilustrados na tabela 2 não são os únicos suportados, devida a utilização do MPI com outras linguagens e em diferentes hardware. A linguagem Fortran, por exemplo, possui um tipo de dados COMPLEX, dessa forma se o MPI for utilizado com Fortran, existe o tipo de dados MPI_COMPLEX (CENAPAD, 2001, p.16). Por outro lado, se um sistema computacional fornece um tipo de dados *long long int*, pode ser disponível para uma implementação do MPI nesse sistema computacional o tipo MPI_LONG_LONG_INT (PACHECO, 1997, p.48).

MPI Básico

A grande quantidade de funções disponibilizadas pelo MPI pode transparecer uma idéia de que o entendimento da programação de aplicações MPI é uma tarefa de difícil execução, pela necessidade do domínio de um conjunto extenso de funções. Essa idéia não é verdadeira, uma vez que com um conjunto mínimo de funções pode-se construir uma grande quantidade de aplicações paralelas para diversos casos. Este conjunto mínimo é freqüentemente relacionado na literatura de área como MPI básico (GROPP; LUSK; SKJELLUM, 1999, p. 21, CENAPAD, 2001, p.24-30, COSTA; STRINGHINI; CAVALHEIRO, 2002, p. 58, IGNÁCIO; FERREIRA FILHO, 2002, p. 108, GRAMA et. al., 2003, GEYER, 2003). A Tabela 3 relaciona as funções que compõem o MPI básico.

Tabela 3: Funções componentes do MPI básico

Função	Significado
MPI_Init	Inicializa o MPI
MPI_Comm_rank	Determina o *rank* do processo
MPI_Comm_size	Determina a quantidade de processos sendo executados.
MPI_Finalize	Finaliza o MPI
MPI_Send	Envia mensagem
MPI_Recv	Recebe mensagem

As quatro primeiras funções da tabela são praticamente obrigatórias em todas as aplicações com MPI, pois realizam funções como início e término das aplicações e identificação dos processos sendo executados. As duas últimas, proporcionam uma comunicação ponto a ponto, portanto são muito utilizadas. Por serem consideradas rotinas básicas, praticamente todas elas são indispensáveis na construção de aplicações MPI, mesmo as consideradas com maior grau de dificuldade.

Para poder fazer uso das funções relacionadas na Tabela 3 e das demais funcionalidades disponibilizadas pelo MPI, o programador deve incluir no início de seu programa a diretiva de inclusão da biblioteca do MPI, através do comando "`#include <mpi.h>`". Obviamente, que para não ocorrerem erros de compilação, pelo menos uma implementação do MPI deve estar corretamente instalada.

Manipulando processos MPI

A primeira função do MPI a ser utilizada é o MPI_Init. Ela inicia o ambiente de execução da aplicação, sincronizando todos os processos em seu início. Não podem existir chamadas a funções do MPI antes do MPI_Init e após a sua execução, uma nova execução causará um erro. Uma descrição dos argumentos da função pode ser visualizada na Figura 12a.

Ao término de uma aplicação MPI, deve-se utilizar a função MPI_Finalize e após nenhuma função do MPI poderá ser executada. De forma análoga ao MPI_Init, ele sincroniza os processos no final da aplicação. O MPI_Finalize não possui parâmetros, retornando apenas um valor indicando a correta execução ou não da função (Figura 12b).

Após os processos MPI serem inicializados (MPI_Init), pode-se obter informações relativas a identificação de cada processo e o número de processos em execução. O MPI_Comm_size (Figura 12c) determina o número de processos sendo executados e o MPI_Comm_rank (Figura 12d) determina a identificação (*rank*) de cada processo (CENPAD, 2001, p.24-30).

```
int MPI_Init(
        int*   argc   /* entrada/saída */
        char** argv[] /* entrada/saída */)   (a)

-----------------------------------------------

int MPI_Finalize(void)                       (b)

-----------------------------------------------

int MPI_Comm_size(
        MPI_Comm comm         /* entrada */
        int*     num_de_procs /* saída */)   (c)

-----------------------------------------------

int MPI_Comm_rank(
        MPI_Comm comm          /* entrada */
        int*     rank_processo /* saída */)  (d)
```

Fonte: adaptado de PACHECO, 1997, p.50-51

Figura 12 – Funções para manipulação de processos em MPI

Os argumentos passados para a função MPI_Init (Figura 12a) são equivalentes aos argumentos de linha de comando na linguagem C. O primeiro (argc) identifica o número de parâmetros passados enquanto o segundo é um vetor dos argumentos propriamente ditos. Ambos são de entrada e saída, porém não é comum o uso dos valores retornados. As implementações do MPI devem fazer uso dos argumentos passados na linha de comando (por exemplo, o número de processos a serem criados através do *mpirun*) e proceder as ações necessárias na sua interpretação (GRAMA et. al, 2003).

Na determinação do número de processos sendo executados (Figura 12c), são utilizados dois argumentos: um comunicador como entrada e um inteiro contendo o número de processos obtidos, passado por referência como saída. A identificação do processo (determinação de seu *rank*) ocorre com a função descrita na Figura 12d, contendo como argumentos um comunicador e uma referência a um inteiro como saída. Este inteiro irá conter o *rank* do processo e será diferente para cada processo em execução (valores de 0 a N-1, onde N é o número de processos).

Nas funções que utilizam um comunicador é geralmente utilizado o MPI_COMM_WORLD, que por padrão congrega todos os processos em execução. No retorno de todas as funções apresentadas está um número inteiro que seguindo o padrão da linguagem C retorna 0 (ou uma constante MPI_SUCESS) caso a função seja executada com sucesso ou um código de erro, padronizado pelas implementações.

De posse do funcionamento destas quatro funções para manipulação de processos, já é possível constituir um primeiro programa utilizando MPI. Como padrão, o primeiro programa será um tradicional "Ola mundo" (Hello Word!), descrito na Figura 13.

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

03  int main(int argc, char **argv){
04  int procs;
05  int meurank;

06  MPI_Init(&argc, &argv);
07  MPI_Comm_size(MPI_COMM_WORLD, &procs);
08  MPI_Comm_rank(MPI_COMM_WORLD, &meurank);

09  printf("Oi Mundo!! Sou o processo %d de %d
            sendo executado\n", meurank, procs);

10  MPI_Finalize();

11  return(0);
12  }                                              (a)

  ------------------------------------------------

  $ mpirun -np 4 primeiro
  Oi Mundo!! Sou o processo 0 de 4 sendo executado
  Oi Mundo!! Sou o processo 2 de 4 sendo executado
  Oi Mundo!! Sou o processo 1 de 4 sendo executado
  Oi Mundo!! Sou o processo 3 de 4 sendo executado

                                                   (b)
```

Figura 13 – Programa "Ola Mundo" em MPI

Conforme mostrado na Figura 13a, com exceção das linhas 2, 6, 7, 8 e 10, as demais são semelhantes a qualquer outro programa fonte em C. Na linha 2 é incluída a biblioteca do MPI para a utilização das suas funcionalidades. As linhas 6 e 10 realizam o início e término da aplicação MPI, utilizando para isso as funções descritas nas figuras 12a e 12b respectivamente. A linha 7 utiliza a função descrita na Figura 12c, onde é passada por referência a variável 'procs'. Ao final da execução desta linha de código, a referida variável irá armazenar o número de processos sendo executados nesta aplicação. A linha 8 realiza a obtenção da identificação (*rank*) do processo sendo executado através da função descrita na Figura 12d. De forma análoga à linha 7, está sendo passada por referência a variável 'meurank', e ao término da execução desta linha esta variável irá conter o *rank* de cada processo.

Os valores resultantes nas variáveis manipuladas nas linhas 7 e 8 podem ser visualizados através da linha 9, onde os mesmos são mostrados através da função *printf*. Esses valores são dependentes da execução. Um exemplo de execução é demonstrado na Figura 13b, onde a aplicação resultante do código fonte da Figura 13a é executada com quatro processos.

É importante salientar que o programa fonte da Figura 13a possui apenas uma chamada da função *printf* gerando apenas uma linha de texto na saída, porém em sua execução (Figura 13b) pode ser visto quatro linhas de texto como saída. Isso ocorre em virtude da execução da aplicação em quatro processos, onde em cada processo a linha 9 (*printf*) é executada. Mesmo que sejam quatro linhas resultantes, todas possuem valores distintos para a identificação do processo (meurank), enquanto que o número de processos sendo executados (procs) é o mesmo.

A execução da aplicação demonstrada na Figura 13b foi realizada num cluster, sendo a execução disparada a partir de seu *front-end*, com um processo executado nesta máquina e os demais em máquinas responsáveis pelo processamento. Dessa forma, a função *printf* foi executada em diferentes máquinas, porém sua saída acontece na máquina que efetuou o disparo. O MPI redireciona a saída de vídeo para a máquina que realiza o disparo das aplicações (PACHECO, 1997, p.137). Esse fato é interessante, pois se as funções *printf* fossem direcionadas nas máquinas que as executam não seriam visualizadas pelo usuário, uma vez que na grande maioria dos agregados de computadores as máquinas de processamento não possuem dispositivos de I/O padrão (monitor e teclado). Mesmo que possuíssem monitor, ainda assim a saída seria direcionada para a máquina que disparou a aplicação, pois sob o ponto de vista do MPI não existem máquinas físicas isoladas e sim uma máquina paralela virtual.

A saída gerada em outras máquinas é comunicada pela rede até a máquina que realizou o disparo da aplicação e dessa forma não é garantida a chegada em ordem das saídas geradas. A Figura 13b ilustra este fato, pois a saída da máquina de *rank* = 2 aparece antes da gerada com *rank* = 1.

Enviando e recebendo mensagens

O programa ilustrado na Figura 13 serve para introduzir a programação com o MPI, porém a comunicação entre os processos não é contemplada. A comunicação entre processos MPI ocorre através de trocas de mensagens e o conjunto de funções conhecido com MPI básico possui funções específicas para o envio e recebimento de mensagens. A Figura 14 ilustra as funções para trocas de mensagens em MPI.

```
int MPI_Send(
        void*        mensagem    /* entrada */
        int          elementos   /* entrada */
        MPI_Datatype tipo_MPI    /* entrada */
        int          destino     /* entrada */
        int          tag         /* entrada */
        MPI_Comm     comunicador /* entrada */
                                                 (a)

------------------------------------------------

int MPI_Recv(
        void*        mensagem    /* saída */
        int          elementos   /* entrada */
        MPI_Datatype tipo_MPI    /* entrada */
        int          origem      /* entrada */
        int          tag         /* entrada */
        MPI_Comm     comunicador /* entrada */
        MPI_Status*  status      /* saída */
                                                 (b)
```

Fonte: adaptado de PACHECO, 1997, p. 51
Figura 14 – Funções para trocas de mensagens em MPI

A composição das mensagens enviadas ou recebidas segue o modelo de mensagens demonstrado na Figura 12, sendo que a função de envio de mensagens (Figura 14a) possui seis argumentos e a de recebimento (Figura 14b) segue a mesma lógica, apenas adicionando um argumento de status no recebimento das mensagens.

O primeiro argumento da função de envio de mensagens (Figura 14a) é uma referência a uma posição de memória onde os dados a serem enviados se encontram. Logo em seguida, deve-se indicar a quantidade de elementos e o tipo dos dados a serem enviados. Os tipos de dados devem ser tipos de dados definidos pelo MPI (Tabela 2). O próximo argumento do envio é a identificação (*rank*) do processo que deverá receber a mensagem, seguido do *tag* (assunto) e finalizado pelo comunicador a ser utilizado, geralmente o MPI_COMM_WORLD.

A função de recebimento possui praticamente os mesmos argumentos com apenas a adição de um para status e uma modificação no significado do primeiro e do quarto argumentos. No primeiro argumento ao invés de indicar uma área de memória de onde os dados irão ser copiados, deve-se indicar uma área de memória onde os dados devem ser recebidos. Diferente da função de envio, o primeiro argumento retém um valor (saída) após a execução da função. O quarto argumento ao invés de ser indicado o *rank* do destino (envio) da mensagem deve-se indicar de quem (origem) a mensagem deverá ser recebida. Ao final da especificação da função de recebimento, deve-se indicar uma variável para armazenar, após sua execução, o status do recebimento, onde poderão ser encontradas informações de controle relativas a mensagem recebida. No recebimento, tanto a origem da mensagem quanto o *tag* podem ter seus valores associados a coringas.

Tanto a função de envio quanto à de recebimento de mensagens retornam por padrão o valor 0 (MPI_SUCESS) se foram corretamente executadas ou um código de erro. A Figura 15 ilustra um código fonte onde é estabelecida uma troca de mensagens entre processos.

No código fonte descrito na Figura 15a, pode ser notado a utilização de um tipo de dados disponibilizado pelo MPI: o MPI_Status (linha 7), onde é declarada uma variável de nome ‘status’ deste tipo. A inicialização e identificação dos processos que ocorre nas linhas 8, 9 e 10 é análoga à utilizada no exemplo da Figura 13a.

Na linha 11 começam a ser identificadas diferenças no código a ser executado, através de uma expressão de condição onde a variável de identificação de cada processo ‘meurank’ é verificada. Neste ponto é que entra em cena de forma prática o modelo de programação SPMD. A expressão verificada na linha 11 somente será verdadeira para o processo com *rank* = 0, dessa forma, somente um processo irá executar a porção de código compreendida entre as linhas 12 e 16 (inclusive). Os demais processos irão ter a condição da linha 11 com o resultado falso, assim, irão executar as linhas 17 a 19 e não irão executar as linhas 12 a 16, mesmo que o código binário correspondente dessas linhas esteja neles inseridos (no modelo SPMD todos os processos possuem o mesmo código binário, mas somente alguns irão executar trechos específicos). A partir da linha 20, todos os processos irão executar todas as linhas restantes do programa.

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

03  int main(int argc, char **argv){
04  int i, envia, recebe;
05  int procs, meurank;
06  int tag=10;
07  MPI_Status status;

08  MPI_Init(&argc, &argv);
09  MPI_Comm_size(MPI_COMM_WORLD, &procs);
10  MPI_Comm_rank(MPI_COMM_WORLD, &meurank);

11  if (meurank == 0){
12     for(i=1; i< procs; i++){
13        MPI_Recv(&recebe, 1, MPI_INT, i, tag,
                    MPI_COMM_WORLD, &status);
14        printf("Recebido o valor %d do
                  processo %d\n", recebe, i);
15     }
16  }
17  else {
18     envia = meurank * 2;
19     MPI_Send(&envia, 1, MPI_INT, 0, tag,
               MPI_COMM_WORLD);
20  }

21  MPI_Finalize();

22  return(0);
23  }                                          (a)

-----------------------------------------------

$ mpirun -np 4 segundo
Recebido o valor 2 do processo 1
Recebido o valor 4 do processo 2
Recebido o valor 6 do processo 3
                                             (b)
```

Figura 15 – Programa envolvendo trocas de mensagens em MPI

A troca de mensagens está localizada no trecho de programa não executado de forma igual por todos os processos. O processo de envio das mensagens está compreendido entre as linhas 17 a 19 enquanto o processo de recebimento esta localizado nas linhas 12 a 16. Na prática, os processos que possuem o *rank* diferente de zero enviam uma mensagem ao processo que possui *rank* igual a zero. A mensagem é composta de uma região de memória com dado tipo MPI_INT, tendo seu valor associado ao *rank* do processo multiplicado de dois (linha 18). Para completar a troca de mensagens, o processo com *rank* = 0 realiza o recebimento das mensagens enviadas (linha 13) mostrando o conteúdo recebido (linha 14).

A garantia de que todas as mensagens serão recebidas é .o *loop* (comando *for*) iniciado na linha 12, tendo seu início em 1 e repetindo enquanto a variável de controle do laço ‘i’ seja menor que o número de processos sendo executados ‘procs’. Esta variável de controle do laço é utilizada para definir a origem das mensagens recebidas na função de recebimento MPI_Recv (linha 13, quarto argumento), dessa forma, primeiro será recebida a mensagem do processo 1, para após receber a mensagem do processo 2 e assim por diante. A Figura 15b ilustra a execução do programa fonte da Figura 15a, sendo executada com quatro processos, sendo a primeira linha de saída a mensagem do com *rank* = 1, seguidas pelas linhas oriundas dos demais processos, em ordem crescente de valor do *rank* de cada processo.

Comunicando conjuntos de dados

No processamento paralelo e distribuído, uma das características geralmente presentes nas aplicações é a manipulação de grandes conjuntos de dados. Estes conjuntos precisam ser enviados/recebidos pelos processos MPI a fim de que sejam processados. Por exemplo, caso exista a necessidade de processar um vetor de 1000 elementos em quatro processos MPI, o ideal é que cada um destes processos execute a computação sobre 250 elementos do vetor. No exemplo da Figura 15, onde valores são enviados e recebidos, teríamos de gerar 1000 envios/recebimentos para a distribuição dos dados do vetor a ser processado, o que não seria nada produtivo uma vez que o custo da comunicação (tempo envolvido) de uma mensagem é muito alto se comparado ao tempo de processamento normalmente empregado.

O MPI fornece três mecanismos para a comunicação de conjuntos de dados: o argumento de quantidade de elementos, tipos derivados e o empacotamento de mensagens. No caso do exemplo do vetor de 1000 posições, o argumento de quantidade de elementos a serem enviados/recebidos resolve perfeitamente o problema. Ele pressupõe que os dados devem estar continuamente alocados na memória (vetores em C são alocados de forma contínua na memória) e pode-se facilmente enviar/receber um conjunto de dados (PACHECO, 1997, p.89). A Figura 16 ilustra um exemplo onde são comunicados conjuntos de dados. É importante salientar que as demais alternativas do MPI também podem ser utilizadas para comunicação de vetores, porém são mais indicadas para aplicações onde os dados não estão continuamente alocados na memória.

Na Figura 16a, é definido na linha 6, um vetor de números inteiros ‘vet’ com 12 elementos (o tamanho é ajustado pelo *define* da linha 3), dessa forma, todos os processos MPI que serão executados possuem definido este vetor. A linha 12, executada por todos os processos, calcula a parte do vetor a ser comunicada tendo como variáveis o tamanho do vetor e o número de processos ativos menos um. Esta subtração se faz necessária uma vez que no algoritmo projetado, o processo com *rank* = 0 envia o vetor ‘aos pedaços’ enquanto os demais processos recebem a porção equivalente do vetor e a processam. Convém lembrar que dependendo do número de elementos do vetor e de processos gerados o cálculo da linha 12 pode gerar um número fracionário. Esta situação é prevenida pela modificação do valor de cálculo para um inteiro (int), evitando erros, porém podem ocorrer situações onde nem todo o vetor é processado. Para uma garantia de que o vetor seja integralmente divido e processado deve-se utilizar algoritmos para a divisão (particionamento) que extrapolam o escopo deste texto.

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

03  #define TAM 12

04  int main(int argc, char **argv){
05  int i, meurank, procs, parte, ind=0;
06  int vet[TAM];
07  int tag=10;
08  MPI_Status status;

09  MPI_Init(&argc, &argv);
10  MPI_Comm_size(MPI_COMM_WORLD, &procs);
11  MPI_Comm_rank(MPI_COMM_WORLD, &meurank);

12  parte = (int)(TAM / (procs-1));

13  if (meurank == 0){
14     for(i=0; i<TAM; i++)
15         vet[i] = i;

16     for(i=1; i < procs; i++){
17        MPI_Send(&vet[ind], parte, MPI_INT,
                    i, tag, MPI_COMM_WORLD);
18        ind = ind + parte;
19     }
20  }
21  else {
22     MPI_Recv(vet, parte, MPI_INT, 0, tag,
                MPI_COMM_WORLD, &status);
23     printf("Proc. %d recebeu [ ", meurank);
24     for(i=0; i<parte; i++)
25        printf("%d ", vet[i]);

26  printf("] Final do proc. %d\n",meurank);
27  }

28  MPI_Finalize();
29  return(0);
30  }                                          (a)
```

```
$ mpirun -np 4 terceiro
Proc. 1 recebeu [ 0 1 2 3 ] Final do proc. 1
Proc. 2 recebeu [ 4 5 6 7 ] Final do proc. 2
Proc. 3 recebeu [ 8 9 10 11 ] Final do proc. 3
                                              (b)
```

```
$ mpirun -np 5 terceiro
Proc. 1 recebeu [ 0 1 2 ] Final do proc. 1
Proc. 2 recebeu [ 3 4 5 ] Final do proc. 2
Proc. 3 recebeu [ 6 7 8 ] Final do proc. 3
Proc. 4 recebeu [ 9 10 11 ] Final do proc. 4
                                              (c)
```

Figura 16 – Programa de envio/recebimento de conjuntos de dados

As linhas 14 e 15 realizam a população do vetor com dados, ação executada apenas no processo com *rank* = 0. O envio de pedaços do vetor ocorre da linha 16 a linha 19, sendo enviadas mensagens em ordem para os processos, ou seja, primeiro a mensagem para o processo com *rank* = 1, após para o com *rank* = 2 e assim sucessivamente. Na linha 17 encontra-se a função de envio da mensagem, tendo como destaques o primeiro e segundo argumentos. O primeiro indica a partir de qual posição da memória o vetor começara a ser enviado enquanto o segundo indica quantos elementos serão enviados (calculado na linha 12).

O endereço de envio das mensagens é controlado pela variável 'ind', inicializada com 0 (linha 5), ou seja, para o primeiro processo serão enviadas 'parte' posições a partir do endereço da posição 0 do vetor. Logo após, a variável 'ind' é atualizada (linha 18) com seu valor acrescido da quantidade de elementos enviados para que as posições subseqüentes possam ser referenciadas num próximo envio.

Entre as linhas 21 e 27 são realizadas as ações de recebimento das mensagens contendo um conjunto de dados, sendo executadas por todos os processos com *rank* diferente de 0. A função de recebimento está colocada na linha 22, sendo que o primeiro e segundo argumentos são os que merecem maior atenção. O primeiro argumento identifica a posição de memória que os dados devem ser copiados quando forem recebidos. Nele é colocado o vetor 'vet', que sem a identificação de índice indica que devem ser colocados a partir da primeira posição do vetor (índice 0). Serão colocadas 'parte' (segundo argumento do MPI_Recv) elementos neste vetor, provavelmente não sendo preenchidas todas suas posições. Conforme já discutido, o vetor existe em todos os processos com seu tamanho completo (12). Esta é uma técnica que facilita a programação, porém ocupa mais memória que o necessário uma vez que os dados serão divididos entre os processos. O ideal neste caso seria a alocação dinâmica de uma área de memória para o vetor em cada um dos processos, alocando apenas a quantidade de memória a ser utilizada. Após o recebimento cada um dos processos mostra os valores recebidos (linhas 23 a 26).

Nas Figuras 16b e 16c são mostradas execuções do código fonte da Figura 16a com 4 e 5 processos respectivamente. Pode-se notar que a quantidade de elementos do vetor nos processos muda de acordo com o número de processos inicializados (4 com 4 processos e 3 com 5 processos). Isso ocorre devido cálculo efetuado na linha 12.

Recebendo mensagens com coringas

No exemplo de envio/recebimento de mensagens da Figura 15, onde o processo com *rank* = 0 recebe mensagens dos demais, ocorre uma especificação de ordem do recebimento das mensagens. Isso é definido na linha 13, pelo quarto argumento onde é especificada a variável 'i', controlada pelo laço, forçando primeiro o recebimento da mensagem do processo com *rank* = 1, após a do *rank* = 2 e assim sucessivamente. Porém, nem sempre esta situação de especificação de quem receber a mensagem primeiro é útil, podendo ocasionar espera desnecessária dos processos (bloqueio temporário). Como exemplo, pode-se citar um processo que distribua tarefas aos demais e após o recebimento de um resultado ele deva enviar outra tarefa a quem lhe enviou o resultado (típico caso de algoritmo mestre/escravo).

Outro ponto mantido com especificação explícita nos exemplos de envio/recebimento de mensagens das Figuras 15 e 16 é o assunto (*tag*) das mensagens. Em ambos os casos, o *tag* é fixo (10), sendo neste caso todas as mensagens comunicadas com o mesmo assunto. Podem ocorrer casos onde o assunto de uma mensagem deve ser diferente das demais. Por exemplo, caso um processo atue como modificador de dados, sempre ficando num *loop* onde recebe um valor, transforma–o de acordo com algum cálculo e após devolve-o a quem lhe enviou. Neste caso, deve haver uma possibilidade de enviar uma mensagem para finalizar os processos transformadores de dados. Para isto, pode ser utilizado um valor pré-definido em mensagem, o que causaria o problema de nunca processar este valor, ou utilizar uma mensagem com o assunto (*tag*) de final de processamento.

Para resolver os problemas supra-citados o MPI oferece no recebimento das mensagens a possibilidade da não especificação de quem enviou a mensagem e nem o assunto da mensagem. Ao invés, utiliza-se coringas no lugar da especificação destes campos. O coringa 'MPI_ANY_SOURCE' pode ser utilizado no lugar de quem enviou a mensagem, causando o recebimento da mensagem de qualquer processo. De forma análoga, o coringa 'MPI_ANY_TAG' pode ser utilizado para especificar que a mensagem a ser recebida pode ser de qualquer assunto (*tag*). A programação de aplicações com uso de coringas proporciona uma maior flexibilidade no recebimento das mensagens.

Porém, de pouco adiantaria o recebimento destas mensagens com os coringas sem poder identificar, após o seu recebimento, quem a enviou e qual o seu assunto. Para isso, o MPI disponibiliza uma estrutura associada ao recebimento das mensagens onde estas informações podem ser recuperadas. Esta estrutura é ligada a variável de status no recebimento das mensagens (último argumento). A Figura 17 ilustra a composição da estrutura de dados MPI_Status.

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
};
```

Fonte: GRAMA et. al., 2003
Figura 17 – Estrutura de dados do MPI_Status

Após o recebimento das mensagens, pode-se utilizar os campos da estrutura da Figura 17 para identificar a origem da mensagem recebida (MPI_SOURCE) e o assunto (MPI_TAG). Tanto a origem como o assunto podem ser tratados como números inteiros, dessa forma, pode-se manipulá-los nos programas. A referência é realizada como normalmente se faz acesso a campos de estruturas de dados, com o nome da variável que contém a estrutura, seguida de ponto e após o nome do campo desejado. Por exemplo, caso a variável utilizada no recebimento seja 'status', pode-se utilizar 'status.MPI_SOURCE' para identificar quem enviou a mensagem. A Figura 18 ilustra um programa fonte com utilização dos coringas no recebimento das mensagens.

O programa descrito na Figura 18a, ilustra um código fonte onde são realizadas duas operações de trocas de mensagens, sendo que em cada uma delas, os processos estão divididos entre o de *rank* = 0 (linhas 11 a 17) e os demais (linhas 19 e 22). O primeiro processo de trocas de mensagens é iniciado pelo processo com *rank* = 0, enviando uma mensagem aos demais (linhas 11 e 12) . Esta é recebida (linha 19), finalizando o primeiro ciclo de envio/recebimento, tendo após um novo valor calculado (linha 20). O segundo processo de envio/recebimento de mensagens é originado pelos processos com *rank* diferente de 0, que enviam uma mensagem ao processo com *rank* = 0 (linha 21). O processo com *rank* = 0, recebe as mensagens (linha 14), finalizando o segundo processo de trocas de mensagens, mostrando após informações aos usuários (linha 15).

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

03  int main(int argc, char **argv){
04  int i, envia, recebe;
05  int procs, meurank;
06  MPI_Status st;

07  MPI_Init(&argc, &argv);
08  MPI_Comm_size(MPI_COMM_WORLD, &procs);
09  MPI_Comm_rank(MPI_COMM_WORLD, &meurank);

10  if (meurank == 0){
11     for(i=1; i< procs; i++)
12        MPI_Send(&i, 1, MPI_INT, i, i,
                   MPI_COMM_WORLD);
13     for(i=1; i< procs; i++){
14        MPI_Recv(&recebe, 1, MPI_INT,
                   MPI_ANY_SOURCE, MPI_ANY_TAG,
                   MPI_COMM_WOLD, &st);
15        printf("Recebido %d do proc. %d com
                  tag %d\n", recebe,
                  st.MPI_SOURCE, st.MPI_TAG);
16     }
17  }
18  else {
19     MPI_Recv(&recebe, 1, MPI_INT, 0,
               MPI_ANY_TAG, MPI_COMM_WORLD,
               &st);
20     envia = recebe * st.MPI_TAG;
21     MPI_Send(&envia, 1, MPI_INT, 0, meurank,
               MPI_COMM_WORLD);
22  }
23  MPI_Finalize();

24  return(0);
25  }                                          (a)
```

```
$ mpirun -np 6 quarto
Recebido 1 do proc. 1 com tag 1
Recebido 4 do proc. 2 com tag 2
Recebido 25 do proc. 5 com tag 5;
Recebido 16 do proc. 4 com tag 4
Recebido 9 do proc. 3 com tag 3
                                              (b)
```

Figura 18 – Programa com uso de coringas

No primeiro processo de envio/recebimento, são enviadas mensagens contendo o valor da variável 'i', com *tag* valendo a própria variável 'i' (linha 12). No seu recebimento (linha 19), a mensagem é recebida sendo utilizado o coringa MPI_ANY_TAG, representando que seriam aceitas mensagens com qualquer valor de *tag*. O *tag* recebido é utilizado no cálculo do valor da linha 20, sendo manipulado através da associação da variável de status com o campo desejado (`st.MPI_TAG`). Como o valor recebido na mensagem é mesmo do valor do *tag*, o cálculo da linha 20 sempre irá calcular o quadrado do próprio *rank*.

O valor calculado na linha 20 é o que vai ser enviado, incializando o segundo processo de trocas de mensagens. A linha 21 é responsável pelo envio deste valor, informando no campo *tag* o valor do *rank* de cada processo. Estas mensagens são recebidas sem ordem específica e com qualquer *tag*, uma vez que na linha 14, onde o MPI_Recv está colocado, estão sendo utilizados os coringas MPI_ANY_SOURCE e MPI_ANY_TAG nos locais de origem e *tag* respectivamente. Após este recebimento, são mostrados os valores recebidos, de quem foram recebidos e com qual *tag* (linha 15). A primeira informação é o próprio valor recebido enquanto que a segunda e a terceira são campos da estrutura de status do recebimento, respectivamente `st.MPI_SOURCE` e `st.MPI_TAG`.

A Figura 18b ilustra uma execução do programa fonte descrito na Figura 18a com a execução de seis processos. Pode-se notar que as mensagens não aparecem em ordem crescente da identificação dos processos. Isso ocorre, pois não foi forçado o recebimento das mensagens em ordem, utilizando o coringa MPI_ANY_SOURCE no recebimento. Convém salientar que a ordem de recebimento apresentada não é sempre a mesma, podendo inclusive as mensagens serem recebidas em ordem crescente de *rank*. Essa é uma possibilidade, o que é certo é que não há primeiro o recebimento da mensagem do processo com *rank* = 1 para após o do com *rank* = 2: isso até pode acontecer mas não é regra.

Comunicações de grupo e assíncronas

As funções de comunicação (trocas de mensagens) ilustradas na Figura 14 realizam o que é conhecido como comunicação síncrona ponto a ponto, ou seja, as chamadas das rotinas causam um bloqueio no envio e no recebimento das mensagens. No caso do envio, o bloqueio ocorre até que os dados possam ser copiados para o *buffer* de envio, o que é uma operação extremamente rápida e dá a impressão ao usuário que não ocorreu um bloqueio. Por outro lado, o bloqueio no recebimento ocorre até que uma mensagem com as características esperadas seja recebida. Essa comunicação é ponto-a-ponto, pois são sempre entre dois processos.

Além destas funções, o MPI disponibiliza outras que fornecem uma comunicação assíncrona (sem bloqueio) e também comunicação entre grupos de processos. Na relação das funções de envio/recebimento de mensagens sem bloqueio, destacam-se o MPI_Isend e o MPI_Irecv que realizam envio e recebimento assíncrono respectivamente. Estas funções possuem muitas semelhanças às correspondes bloqueantes, porém, extrapolam o escopo deste texto. Uma completa referência pode ser encontrada em PACHECO, 1997, CENAPAD, 2001 e GEYER 2003.

Além das funções para comunicação assíncrona, o MPI fornece uma extensa lista de operações para comunicação de grupo. Dentre as principais pode-se relacionar:

- MPI_Bcast: envia dados para todos os processos, sendo eles recebidos com a mesma função;
- MPI_Scatter: segmenta e envia dados a um conjunto de processos;
- MPI_Gather: coleta dados distribuídos em diferentes processos;
- MPI_Reduce: realiza uma computação global em vários processos, resultando um valor único.

De forma análoga as funções de comunicação assíncrona, as funções que proporcionam comunicações de grupo extrapolam o conteúdo deste texto, sendo encontradas maiores informações em PACHECO, 1997, CENAPAD, 2001, IGNÁCIO; FERREIRA FILHO, 2002 e GEYER 2003.

CONSIDERAÇÕES FINAIS

A utilização do MPI na programação de aplicações paralelas que podem ser executadas em agregados de computadores foi o principal tema abordado por este mini-curso. Neste contexto, foram abordadas as arquiteturas paralelas, com ênfase nos clusters e as formas de exploração e anotação do paralelismo em programas. Também a biblioteca para exploração do paralelismo MPI foi trabalhada, incluindo sua grande portabilidade, devida a inúmeras versões disponibilizadas, e seu mecanismo para criação de máquinas virtuais paralelas. Ainda foram detalhados algumas das principais rotinas do MPI para a escrita de aplicações paralelas.

Pode-se concluir que o domínio da programação paralela através do MPI é uma tarefa não trivial que requer algum conhecimento. Porém, não é uma tarefa de alta complexidade, pois para a escrita de uma grande quantidade de aplicações paralelas, o programador necessita apenas dominar poucas funções do MPI e compreender o funcionamento das trocas de mensagens entre os processos, uma vez que os mesmos cooperam na execução das aplicações.

É extremamente recomendado que os leitores desenvolvam seus próprios programas, complementando e até mesma invertendo a lógica dos diversos exemplos disponibilizados, principalmente os com trocas de mensagens com uso de coringas. Aos que desejam aprofundar-se no tema, sugere-se a análise e interpretação de programas paralelos completos, tais como aplicações que utilizam algoritmos paralelos do tipo divisão-e-conquista e mestre-escravo.

REFERENCIAS BIBLIOGRÁFICAS

ARGONNE NATIONAL LABORATORY. **Message Passing Interface Implementations**. Disponível em <http://www-unix.mcs.anl.gov/mpi/implementations.html>. Acesso em 1 mar. 2004.

BIAZUS NETO, Luiz D.; REBONATTO, Marcelo T.; BRUSSO, Marcos J. **Uma ferramenta para acesso à agregados de computadores utilizando HTTP**. In: Escola Regional de Alto Desempenho, 4, 2004, Pelotas. Anais ... Pelotas: SBC/UFPel/UCPel/UFSM, 2004. p.285-288.

CÁCERES, Edson N.; SONG, Siang W. **Algoritmos Paralelos usando CGM/PVM/MPI: Uma Introdução**. Texto preparado para o XXI Congresso da Sociedade Brasileira da Computação, Jornada de Atualização em Informática. Disponível em <http://www.ime.usp.br/~song/papers/jai01.ps.gz>. Acesso em 14 Fev. 2004.

CAVALHEIRO, Gerson Geraldo Homrich. **Introdução à Programação Paralela e Distribuída**. In: Escola Regional de Alto Desempenho ERAD, 2001. Anais ... Gramado: SBC/Instituto de Informática da UFRGS/Faculdade de Informática da PUCRS/UNISINOS. 2001. p.35-74.

_____ **Princípios da Programação Concorrente**. In: Escola Regional de Alto Desempenho ERAD, 2004. Anais ...Pelotas: SBC/UFPel/UCPel/UFSM. 2004. p.3-40. ISBN: 85-88442-74-4.

CENAPAD. **Curso de MPI**. Disponível em <ftp://ftp.cenapadne.br/pub/cenapad/mpi>. Acesso em 16 ago. 2001.

CLUSTERS@TOP500. **clusters@top500**. Disponível em <http://clusters.top500.org>. Acesso em 25 jan. 2004.

COSTA, Celso Maciel; STRINGHINI, Denise; CAVALHEIRO, Gerson Geraldo Homrich. **Programação concorrente: Threads, MPI e PVM**. In: Escola Regional de Alto Desempenho ERAD, 2002. Anais ... São Leopoldo: SBC/Instituto de Informática da UFRGS/UNISINOS/ULBRA. 2002. p.31-65. ISBN: 85-88442-16-7.

De ROSE, César A. F. **Arquiteturas Paralelas**. In: ERAD 2001, Escola Regional de Alto Desempenho. Anais … Porto Alegre: SBC/Instituto de Informática da UFRGS. 2001. p.3-33.

De ROSE, César A. F.; NAVAUX, Philippe O. A. **Arquiteturas Paralelas**. Porto Alegre: Editora Sagra Luzzatto, 2003. 152p.

GEYER, Claudio F. R. **Programação Paralela: Uma introdução**. Disponível em <http://www.inf.ufrgs.br/procpar/disc/cmp167/mapoio/NotasDeAula/AmbPDP/tecnicas-MPI-Andre-2003.zip>. Acesso em 15 dez. 2003.

GOULART, Peter Carvalhal et. al. **PARALELISMO: Algoritmos e complexidade**. Porto Alegre: UFRGS, Instituto de Informática, 1999. 106p. RP-306.

GRAMA, Ananth et. al.. **Introduction to Parallel Computing**. Second Edition, 2003. Redwood City: Addison Wesley. 2003, 656p.

GROPP, Wiliam; LUSK, Ewing; SKJELLUM, Anthony. **Using MPI: portable parallel programming with the message-passing interface**. 2nd ed. Cambridge: The MIT Press, 1999. 371p. ISBN: 0-262-57134-X.

HARGROVE ,William W.; HOFFMAN, Forrest M.; STERLING, Thomas. **The Do-It-Yourself Supercomputer**. Disponível em de <http://portal.ncsa.uiuc.edu/SJK/appearances/cluster/sciam.html>. Acesso em 27 abr. 2004.

HWANG, Kai; XU, Zhiwei. **Scalable Parallel Computing**: Technology, Architecture, Programming. Boston: McGraw-Hill, 1998. p.3-90, p.453-564 e p.318-326.

IGNÁCIO, Aníbal Alberto Vilcapona; FERREIRA FILHO, Virgílio José Martins. **MPI: Uma ferramenta para implementação paralela**. Pesquisa Operacional. Rio de Janeiro, v.22, n.1, p.105-116, janeiro a junho de 2002. ISSN: 0101-7438.

KINDEL, Cândice C. et. al. **Projeto e implementação de um cluster Beowulf**. In: Escola Regional de Alto Desempenho, 4, 2004, Pelotas. Anais ... Pelotas: SBC/UFPel/UCPel/UFSM, 2004. p.249-252.

KUMAR, Vipin et. al. **Models of Parallel Computers**. In: Introduction to Parallel Computing: Design and Analysis of Algorithms. Redwood City: The Benjamin/Cummings Publishing Company, 1994. 597p. chap. 2, p.15-64.

LAM/MPI PARALLEL COMPUTING. **MPI Implementation List**. Disponível em <http://www.lam-mpi.org/mpi/implementations/fulllist.php>. Acesso em 2 mar. 2004.

MESSAGE PASSING INTERFACE FORUM. **MPI-2: Extensions to the Message-Passing Interface**. Disponível em <http://www.mpi-forum.org/docs/mpi-20.ps.Z>. Acesso em 1 Mar. 2004.

NAVAUX, Philippe O. A. et. al. **Execução de aplicações em ambientes concorrentes**. In: ERAD 2001, Escola Regional de Alto Desempenho. Anais … Porto Alegre: SBC/Instituto de Informática da UFRGS. 2001. p.179-193.

PACHECO, Peter S. **Parallel programming with mpi**. San Francisco: Morgan Kaufmann, 1997. 418 p.

RITZDORF, Hubert. **MPI/SX**. Disponível em <http://www.ccrl-nece.de/~ritzdorf/mpisx.shtml>. Acesso em 1 mar. 2004 (2004a).

_____. **MPI/ES for the Earth Simulator**. Disponível em <http://www.ccrl-nece.de/~ritzdorf/mpies.shtml>. Acesso em 1 mar. 2004b (2004b).

SATO, Liria Matsumoto; MIDORIKAWA, Edson Toshimi; SENGER, Hermes. **Introdução a Programação Paralela e Distribuída**. Disponível em <http://www.lsi.usp.br/~liria/jai96/apost.ps>. Acesso em 7 maio 1999.

SILVA, Márcio Gonçalves da. **Influência de Parâmetros Arquiteturais em Sistemas Paralelos de Programação em Lógica**. Rio de Janeiro, 1999, 101p. Dissertação de Mestrado - COPPE/UJRJ.

TANEMBAUN, Andrews S.; WOODHULL, Albert S. **Sistemas operacionais: projeto e implementação**. 2 ed. Porto Alegre: Bookman, 2000. 758p. ISBN: 85-7307-530-9.

TOP500 SUPERCOMPUTER SITES. **TOP500 List 11/2003**. Disponível em <http://www.top500.org/list/2003/11/>. Acesso em 13 dez. 2003

_____ **TOP500 Supercomputer sites**. Disponível em <http://www.top500.org>. Acesso em 25 jan. 2004.

ZOMAYA, Albert Y. H. **Parallel and Distributed Computing: The Scene, the Props, the Players**. In: Parallel & Distributed Computing Handbook. New York: McGraw-Hill, 1996. p.5-23.