2

Programação Concorrente: *Threads*, MPI e PVM

Celso Maciel da Costa *(PUCRS, celso@inf.pucrs.br)*[1]
Denise Stringhini *(UFRGS, ULBRA, string@inf.ufrgs.br)*[2]
Gerson Geraldo Homrich Cavalheiro *(UNISINOS, gersonc@exatas.unisinos.br)*[3]

Resumo:

A programação concorrente, nas suas mais diversas formas, tem sido sido mais e mais utilizada nos meios acadêmico, científico e de produção. Apesar de não ser uma técnica de programação realmente nova, este aumento de interesse se deu em resposta à maior acessibilidade à recursos computacionais de alto desempenho - as arquiteturas multiprocessadoras e os agregados de computadores.

Neste curso são apresentadas três das principais ferramentas de programação sobre agregados de computadores: *threads* POSIX, *Parallel Virtual Machine* (PVM) *Message Passing Interface* (MPI). Enquanto que PVM e MPI são voltadas à exploração de arquiteturas com memória distribuída, *threads* exporam a concorrência em arquiteturas dotadas de diversos processadores compartilhando uma área de memória comum.Para cada uma destas ferramentas, são apresentadas as caracterísitcas principais, as quais refletem a finalidade de sua utilização, e também os principais serviços oferecidos. Um conjunto de exemplos desenvolvido com cada uma destas ferramentas permite ao leitor ter uma visão prática da programação concorrente através de diferentes técnicas.

---

[1] Doutor em Ciência da Computação pela Université Joseph Fourier, França e Mestre em Ciência da Computação junto ao PPGC da UFRGS. Professor e Pesquisador da Faculdade de Informática da PUCRS. Consultor do MEC para avaliação de cursos da área de Computação e Informática. Áreas de Interesse: Ambientes de Execução Paralela e Programação Paralela e Distribuída.

[2] Doutoranda e Mestre em Ciência da Computação junto ao Curso de Pós-Graduação em Ciência da Computação da UFRGS. Professora da Universidade Luterana do Brasil. Áreas de Interesse: Processamento de alto desempenho e Ambientes de programação paralela

[3] Doutor em Ciência da Informática: Sistemas e Comunicações pelo INPG, França, e Mestre em Ciência da Computação junto ao PPGC da UFRGS. Bacharel em Informática pela PUCRS. Professor e Pesquisador do Centro de Ciências Exatas e Tecnológicas da UNISINOS. Orientador de Mestrado no PIPCA da UNISINOS. Áreas de Interesse: Processamento de Alto Desempenho, Escalonamento e Ambientes de Execução.

2.1. Introdução

Obter bons índices de desempenho na execução de suas aplicações tem sido uma preocupação constante do meio de produção de software. Em especial, para programadores de aplicações com alto custo computacional, cujos problemas envolvem elevado tempo de cálculo ou grande consumo de memória. As soluções tem sido buscadas na implementação de programas concorrentes, utilizando os recursos de programação disponíveis nas diferentes arquiteturas para o processamento de alto desempenho disponíveis.

Atualmente, as arquiteturas em voga para o processamento de alto desempenho são os agregados de computadores, arquiteturas compostas de diversos computadores autônomos - não raro dotados de 2 ou 4 processadores, os chamados, *Symmetric Multi-Processors* (SMPs). Estas arquiteturas tem preterido outras opções devido, principalmente, quando avaliados quesitos de versatilidade e custo. No entanto, como este tipo de configuração de hardware oferece recursos para que a concorrência seja explorada a dois níveis, internos aos nodos e entre os nodos do agregado, diferentes técnicas de programação devem ser utilizadas.

Neste capítulo são apresentadas três das principais ferramentas utilizadas para a programação concorrente em agregados de computadores. A primeira a ser vista, as *threads* POSIX, discutem a questão da programação concorrente em arquiteturas dotadas de memória compartilhada. As outras duas consistem em ferramentas consagradas para realização de aplicações sobre arquiteturas com memória distribuida: PVM (*Parallel Virtual Machine*) e MPI (*Message Passing Interface*).

2.2. *Threads* POSIX

Uma primeira proposta de exploração de concorrência em sistemas de computação se deu através da execução simultânea de diversos fluxos de execução sobre os recursos de uma mesma arquitetura monoprocessadora (um processador, um módulo de memória, a arquitetura von Neumann). Neste caso, os fluxos de execução foram concebidos para, além de compartilhar, no tempo, o acesso ao processador (e demais recursos da arquitetura, como sistema de entrada/saída), a cooperar entre si através de dados compartilhados em memória.

Apesar de definida há muito tempo (década de 1960), esta estrutura resta válida, tendo sido implementada de diferentes maneiras, de acordo com os recursos computacionais disponíveis e/ou as necessidades das aplicações.

Em um primeiro momento, este modelo permitiu a execução simultânea de diversos processos, caracterizando fluxos de execução independentes, cooperando entre si através de leituras e escritas em um espaço de memória compartilhado. Este processo consiste na instancia de um programa, sendo em uma entidade ativa, dotada de um fluxo de execução (ou *thread*) interno, onde são executadas, uma após outra, as instruções definidas pelo programa, e um espaço de memória, onde são armazenadas seus dados internos. Para que os processos tenham acesso à região de memória compartilhada, instruções especiais foram disponibilizadas. Um grupo destas instruções especiais foi definido para oferecer a capacidade do processo de alocar/liberar área para variáveis e de ler/escrever de dados sobre estas variáveis. Como os diversos processo encontram-se em execução simultânea, um novo grupo de instruções foi criado para possibilitar o controle do acesso destes às variáveis compartilhadas; as instruções deste grupo

oferecem mecanismos de sincronização no acesso aos dados, tornando possível, por exemplo, evitar o acesso simultâneo por dois (ou mais) processos a uma mesma variável.

2.2.1. A multiprogramação leve

Com o aumento da necessidade de melhores índices de desempenho na execução de programas, a utilização da concorrência através de processos executando de forma concorrente tem sido preterida, abrindo espaço para a multiprogramação leve (*multithreading*). Esta forma de programação, na realidade, explora o mesmo modelo de programação oferecido pela da utilização de processos concorrentes. A diferença reside nos recursos de implementação utilizados, os quais permitem que os programas sejam executados com desempenho bastante superior.

Atualmente consistindo na ferramenta típica de exploração da concorrência intra-nó, a multiprogramação leve permite que vários fluxos de execução sejam instanciados no interior de um processo. Cada um destes fluxos de execução é chamados de processo leve ou *threads*, executando uma a uma instruções pertencentes a um trecho de código definido. A expressão "processo leve" como sinônimo de *thread* é utilizada tendo como base a literatura clássica sobre UNIX, sendo uma contraposição a estrutura "pesada" de um processo convencional: as *threads* são consideradas leves por serem menos onerosas a serem manipuladas (pelo sistema operacional) [OLI 01]. Uma *thread* necessita apenas descrever o fluxo de execução ao qual encontra-se associada (basicamente uma pilha de dados e o conjunto de registradores); os demais recursos de processamento empregados são os disponibilizados no contexto do processo, sendo estes compartilhados por todas suas *threads* ativas [VAH 76] em seu interior.

Um recurso em particular é compartilhado pelas *threads*: o espaço de endereçamento do processo. Em outras palavras, a área de memória do processo é acessível por todas as *threads* ativas, portanto não se faz necessária nenhuma instrução especial de acesso à área compartilhada de memória, sendo apenas necessárias instruções para controle do acesso aos dados compartilhados.

Outra conseqüência do compartilhamento do espaço de endereçamento é que diferentes *threads* podem vir a executar um mesmo trecho de código. De forma semelhante a que vários processos podem ser instancias diferentes de um mesmo programa. Neste caso, é importante ter consciência de que cada uma das *threads* é uma instância diferente (possuindo, por exemplo, seus próprios dados internos).

Uma questão interessante diz respeito a granulosidade que pode ser obtida pela utilização da multiprogramação leve, ou seja, da relação entre número de instruções executadas por um fluxo de execução com as sincronizações entre estes fluxos. *threads*, sendo menos onerosas que os processos convencionais, permitem que o programador descreva sua aplicação com um maior número de atividades concorrentes sem a perda substancial de desempenho que seria verificada caso esta mesma descrição fosse implementada através de processos convencionais. Na multiprogramação leve, a granulosidade é definida em termos de procedimentos (ou funções) concorrentes, enquanto que no caso de processos concorrentes, a granulosidade é definida em termos de módulos de programa.

2.2.2. Implementação de *threads*

A multiprogramação leve encontra-se, nos dias atuais, disponível nos mais diferentes sistemas computacionais, tendo sido implementada tanto apoiada em recursos

de hardware como de software. Exemplos de implementações em hardware para o suporte às *threads* podem ser encontrados nos materiais descrevendo as arquiteturas Tera e Eart-Mana. Esta seção descreve a implementação do suporte de software para *threads*, mais precisamente, aos suportes oferecidos à implementação de *threads* segundo a definição de serviços de *threads* do padrão POSIX.

O suporte em software para *threads*, mesmo sendo disponibilizado nos diferentes sistemas operacionais atuais, é implementado de diferentes formas. No caso de Solaris, AIX e Linux, as *threads* são disponibilizadas por bibliotecas próprias do sistema. Em outros ambientes, como Minix e Windows 95/98, a programação através de *threads* é possível através de bibliotecas não integradas ao núcleo do sistema operacional. O fato de serem ou não disponibilizadas diretamente pelo sistema operacional altera o conjunto de recursos disponíveis para sua implementação, influindo diretamente na forma que os programas são executadosi e, portanto, no seu desempenho.

Os três modelos básicos que seguem as implementações das bibliotecas de *threads* podem ser identificados pelo mecanismo de escalonamento utilizado para alocação de *threads* ao processador. Estes modelos são: 1:1 (*one-to-one*), N:1 (*many-to-one*) e M:N (*many-to-many*).

2.2.3. O modelo 1:1

As *threads* criadas neste modelo são suportadas diretamente pelo sistema operacional, recebendo a denominação de *threads* sistema (ou *threads kernel*). As *threads* sistema possuem, no que diz respeito ao escalonamento ao processador, dos mesmos direitos que os processos convencionais. Assim, um processo composto por n *threads* sistema recebe n vezes mais o processador que um processo composto por apenas uma única *thread*.

As vantagens do modelo *one-to-one* refletem o fato das *threads* serem manipuladas individualmente pelo sistema operacional. Uma arquitetura multiprocessadora pode, desta forma, ser explorada eficientemente: num instante de tempo, cada processador pode estar executando uma das *threads* da aplicação, provendo o paralelismo real na execução das atividades da aplicação. Pelo mesmo princípio, no momento em que uma *thread* sistema bloqueia suas atividades para executar uma operação de entrada/saída, as demais continuam suas respectivas execuções sem nenhum prejuízo.

Este modelo pode ser explorado através da implementação POSIX de *threads* em Linux.

2.2.4. Modelo N:1

Quando o suporte às *threads* não é oferecido pelo sistema operacional, a solução adotada é utilizar bibliotecas de *threads* usuário. Neste caso as *threads* executam ao mesmo nível da aplicação, sendo seu gerenciamento realizado no interior do processo. Assim sendo, o escalonamento de *threads* é realizado quanto o processo for escalonado ele próprio (pelo sistema operacional) para utilizar o processador. Vem daí a denominação deste modelo, *many-to-one*, pois várias *threads* são executadas sobre uma única unidade de escalonamento do sistema. O fato de *threads* usuário serem escalonadas no interior de um processo implica que arquiteturas multiprocessadoras não sejam exploradas por completo -- ou seja, não é possível explorar o paralelismo real --,

e que todo o conjunto de *threads* usuário de um processo seja bloqueado quando uma destas *thread* iniciar uma operação de entrada/saída.

Em contrapartida, a manipulação de *threads* usuário é ainda menos onerosa que a manipulação de *threads* sistema, não limitando o número de atividades concorrentes de uma aplicação ao número de *threads* que podem suportas de forma eficiente.

A biblioteca de *threads* em Linux oferece igualmente este modelo; existem também implementações para outros ambientes, como para o Minix e para o Windows.

2.2.5. Modelo M:N

A terceira forma de implementação, *many-to-many*, permite que as características de ambos modelos anteriores sejam mescladas. Neste modelo, o interior de cada processo podem comportar N *threads* sistema, sobre cada uma das quais é suportada a execução de um subconjunto das M *threads* usuário definidas na aplicação.

Desta forma, o benefício da estrutura mais leve das *thread* usuário, permite que programador não restrinja o grau de concorrência de seu programa em função dos recursos de hardware disponíveis. Basta encontrar a relação entre *threads* usuário e *threads* sistema que ofereça um bom compromisso de desempenho.

Esta distinção permite diferenciar a concorrência existente entre as atividades de uma aplicação da concorrência real que pode ser obtida em uma determinada arquitetura [BLA 90], pois, em geral, M (a concorrência de uma aplicação) é muito maior que N (a capacidade de execução paralela de uma arquitetura real). Assim, o programador restringe-se a definir as tarefas de sua aplicação, sendo estas associadas a *threads* sistema no momento da execução do programa.

Solaris [POW 91] oferece este modelo de execução de *threads*, utilizando *light weight processes*, as LWPs.

2.2.6. A interface POSIX de *threads*

O padrão POSIX 1003.1c (IEEE) define uma interface de programação de aplicações (API) para o desenvolvimento de ferramentas de suporte à programação utilizando *threads*. Nesta seção encontram-se sumarizados alguns dos principais serviços definidos por este padrão.

2.2.6.1. Manipulação de *threads*

Definição do corpo de uma *thread* Em um programa C/C++, o conjunto de instruções a ser executado por uma *thread* é definido no corpo de uma função (ou método). Esta função, construída pelo programador, pode receber parâmetros através de um ponteiro de memória e igualmente retornar algum dado via uma posição de memória (do processo) alterada, sendo seu cabeçalho:

```
void* func( void * args );
```

Este cabeçalho indica que args é, de fato, um ponteiro para uma região de memória onde encontram-se os dados de entrada para a *thread*. Eventuais dados a serem retornados, são igualmente referenciados através de um ponteiro (o retorno da função) a uma região da memória. No corpo da função, o retorno se dá através de um clássico:

```
return dta;
```

Criação de uma *thread* A criação de uma *thread* se dá através da invocação da primitiva `pthread_create` dentro de um bloco qualquer de comandos. Esta primitiva interage com a biblioteca de manipulação de *threads*, para que um novo fluxo de execução seja criado. A primitiva `pthread_create` possui o seguinte cabeçalho:

```
int pthread_create( pthread_t *thid, const pthread_attr_t *atrib,
                    void *(*funcao) (void *), void *args );
```

Uma invocação a `pthread_create` implica na criação de uma nova *thread* responsável pela execução da função `funcao`. Assim sendo, várias invocações à esta primitiva promovem a criação de várias execuções concorrentes da mesma função. O sucesso na criação da nova *thread* pode ser verificado através do retorno desta primitiva: se o retorno for 0 (zero), a nova *thread* foi criada sem erros.

No momento da criação, a biblioteca de *thread* pode ser instruída para manipular a *thread* com alguns atributos especiais, atributos estes fornecidos especificados pelo programador em `atrib`. Caso o argumento passado seja `NULL`, são utilizados os atributos *default*. Um detalhamento sobre estes atributos, e sobre o tipo `pthread_attr_t`, é apresentado na seqüência.

Os eventuais parâmetros requeridos pela função que será executada pela *thread* são enviados para a nova *thread* através do argumento `args`. No endereço de memória apontado por `args` encontra-se o dado que deverá ser utilizado como entrada da função a ser executada pela *thread*. Note que este argumento trata-se de um ponteiro `void`, podendo portanto endereçar qualquer tipo de dado na memória, cabe a função fazer a devida conversão (através de um `cast`, por exemplo).

O primeiro argumento enviado para `pthread_create` consiste em um endereço de memória para uma variável do tipo `pthread_t`; no processo de criação, variável será atualizada com o identificador da *thread* criada. Através deste identificador é possível identificar as *threads* individualmente. Caso se fizer necessário, uma *thread* em execução pode ter acesso ao seu próprio identificador através da primitiva `pthread_self`:

```
pthread_t pthread_self( void );
```

Término de uma *thread* O término da execução de uma *thread* se dá normalmente ao final da execução da função que está sendo executada. Tipicamente, no momento da execução de um `return`. Observe que, como descrito pelo cabeçalho da função, um valor deve ser retornado pela *thread*, no caso, um endereço de memória para o dado de retorno (ou `NULL`, caso nenhum dado deva ser recebido). Como opção ao `return`, o programador pode vazer uso da primitiva:

```
int pthread_exit( void *ret );
```

Sincronizando execuções de *threads* As *threads* executam uma função e, tal uma função convencional, recebem parâmetros de entrada e podem produzir um resultado que deve ser retornado ao seu final. Esta troca de dados, descrita e executada de forma natural na programação seqüencial, implica, na execução concorrente, na necessidade de uma sincronização explícita entre as *threads* em dois momentos: o primeiro refere-se a passagem dos parâmetros de entrada para uma *thread*; o segundo deve tomar lugar no momento em que os dados produzidos por uma *thread* (seu retorno) precisam ser recuperados por uma outra *thread*.

Conforme visto anteriormente, a passagem de parâmetros de entrada se dá na criação de uma *thread*. A recuperação do retorno de uma *thread* somente pode ser obtido através de um mecanismo de sincronização explício, o qual permite que uma *thread* seja bloqueada até que os dados produzidos por uma outra estejam disponíveis. A *thread* bloqueada permanecerá neste estado até que a condição de sincronização seja satisfeita, ou seja, que a *thread* indicada tenha terminado, estando seus os dados de saída disponíveis. A primitiva `pthread_join` garante este tipo de sincronização:

```
pthread_t pthread_join( pthread_t thid, void **ret );
```

O uso do `pthread_join` permite que uma *thread* reste bloqueada até que a *thread* identificada em `thid` seja terminada. A recuperação do retorno de dados da *thread* `thid` é possível através de `ret`, caso a *thread* não retorne nenhum valor, `NULL` pode ser empregado para este parâmetro.

Atributos de criação de *threads* Dentre os parâmetro necessários a primitiva `pthread_create`, um informa os atributos que devem ser considerados para sua execução e manipulação da nova *thread* que esta sendo criada. Estes atributos são informados através de um tipo de dado fornecido pela biblioteca Pthread: `pthread_attr_t`, uma estrutura opaca definida pela biblioteca onde cada campo corresponde a um dos atributos. Uma estrutura opaca diz respeito a uma estrutura cujos campos não devem ser manipulados diretamente pelo usuário, mas sim através de um conjunto de primitivas. No caso desta estrutura, foi citado um conjunto de primitivas, as quais referem-se aos atributos de *threads* mais utilizados.

```
int pthread_attr_init(pthread_attr_t *attr);
```

Esta função permite a inicialização de um descritor de atributos de *thread* com os valores *default* assumidos pela biblioteca.

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detstate);
```

Um dos atributos da *thread* permite permite informar se ela poderá ou não ser sincronizada com uma outra através de uma operação de *join*. Em caso positivo, o atributo deve ser `PTHREAD_CREATE_JOINABLE`, caso contrário, o atributo deve conter o valor `PTHREAD_CREATE_DETACHED`. A diferença esta na liberação da área alocada para a *thread*, caso o atributo *joinable* seja escolhido, esta área será liberada somente após a conclusão da operação de *join*, caso o atributo da *thread* seja *detached*, a área de memória a ela alocada é liberada imediatamente após seu término. Isto se faz necessário, pois, no momento que uma *thread* termina, sua área de dados é liberada e, caso uma segunda *thread* realize uma operação de *join* sobre esta *thread*, em um instante de tempo posterior ao do seu término, o dado produzido não poderá ser recuperado. O atributo *default* é *joinable*.

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
```

O atributo setado por `policy` permite definir a política a ser utilizada para o escalonamento da *thread*. As possibilidades são: `SCHED_OTHER`, a política *default* da biblioteca, `SCHED_RR`, uma política do tipo *Round-Robin* e `SCHED_FIFO`, uma política do tipo *first-in first-out*. A segunda função permite informar se a política de

escalonamento a ser utilizada deve ser aquela adotada pela *thread* que executou a operação de criação ou se a definida pelos parâmetros informados por `pthread_attr_setschedpolicy`. O *default* é `PTHREAD_EXPLICIT_SCHED`, para utilizar os parâmetros informados explicitamente, a herança pode ser selecionada pelo valor `PTHREAD_INHERIT_SCHED`.

```
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
```

Através da biblioteca Pthreads, é possível explorar tanto o modelo 1:1 como o modelo N:1, de execução de *threads* (fique claro que a funcionalidade ainda resta dependente da implementação realizada). Os respectivos atributos são `PTHREAD_SCOPE_SYSTEM` e `PTHREAD_SCOPE_PROCESS`, sendo o primeiro o valor *default*. Caso a opção seja por *thread* sistema, a *thread* sofrerá o escalonamento do sistema operacional para obter acesso ao processador. No outro caso, ela será escalonada no interior do próprio processo onde foi criada.

2.2.7. Compartilhamento de memória

A comunicação entre *threads* não se limita apenas aos parâmetros de entrada e do retorno da função executada por uma *thread*. A própria memória do processo serve de base de comunicação e, como já foi dito, o acesso a memória se dá pela simples execução de instruções de escrita e leitura. Pode ser dito que as *threads* em execução simultânea concorrem pelo acesso a todos os recursos disponíveis pelo processo, mesmo os dados em compartilhados em memória. O problema é garantir que apenas uma das *threads* tenha acesso a um dado compartilhado em determinado instante de tempo, empregando algum mecanismo de sincronização, tais como: exclusão mútua ou coordenação (em [SEB 00] competição e cooperação, respectivamente).

A sincronização no acesso à memória é necessária para evitar que *threads* acessando simultaneamente dados compartilhados tenham informações errôneas ou incompletas. Neste caso, a função da sincronização é controlar a execução de conjuntos de instruções que acessam uma área de dados compartilhada. A este conjunto de instruções é dada a denominação de sessão crítica. Note-se no entanto que, apesar de prover mecanismos para garantir certa coerência nas comunicações entre as *threads*, o indeterminismo de execução dos programas não é eliminado completamente, a utilização de sincronizações através de invocações à primitiva `pthread_join` é, neste caso, mais eficiente.

A seguir são discutidos os recursos mais comuns disponibilizados pelas bibliotecas POSIX para sincronização entre trechos de códigos internos às *threads*.

2.2.7.1. Mutex

Suponha duas *threads* executando ao mesmo tempo e compartilhando um dado armazenado em uma variável inteira, como no exemplo abaixo

```
int x;  // x é uma variável global
```

// Thread A	// Thread B
a = x; // lê dado global	b = x; // lê dado global
a = a + 1; // adiciona	b = b - 1; // subtrai
x = a; // escreve dado global	x = b; // escreve dado global

Este exemplo mostra claramente uma situação onde duas *threads* possuem sessões críticas que trabalham sobre uma área de dados compartilhada, a variável `x`. A execução não controlada de ambos trechos de código por *threads* concorrentes pode resultar em valores não coerentes para `x` no final de ambas execuções. Com um valor inicial 313 para `x`, é de se esperar que o valor final continue sendo 313, afinal, uma das *threads* incrementa `x` de 1 o valor e a outra decrementa `x` também de 1.

Porém pode ocorrer que as *threads* `A` e `B` executem suas instruções de forma intercalada, por exemplo:

Thread	Instrução	x	a	b
A	`a = x;`	313	313	-
A	`a = a + 1;`	313	314	-
B	`b = x;`	313	314	313
A	`x = a;`	314	314	313
B	`b = b - 1;`	313	314	312
B	`x = b;`	312	314	312

O que levaria `x` conter 312, um valor incorreto. Outros entrelaçamentos poderiam resutar em outros valores incorretos, ou mesmo o acaso poderia produzir `x` = 313, o resultado esperado. Em casos como este, é obrigatório garantir a exclusividade de execução às sessões críticas. Esta exclusividade pode ser garantida com a utilização de um **mutex**.

O mutex é um construtor de sincronização que permite que uma *thread* tenha acesso exclusivo a uma área de dados (mutex, do inglês *mutual exclusion*). Tendo exclusividade no acesso, garante-se que uma sessão crítica pode ser executada sem que uma outra *thread* tenha alguma instrução que manipule a mesma área de dados executada, interferindo no resultado.

O funcionamento do mutex baseia-se em operações de *lock* e *unlock*. Ao entrar em uma sessão crítica, é fechada uma "porta" impedindo que outras *threads* avancem pela sessão crítica -- esta é a operação *lock*. Ao sair de uma sessão crítica, abre-se a porta, permitindo que outras *threads* avancem pelas suas respectivas sessões críticas -- a operação *unlock*.

Uma *thread* restará bloqueada aguardando a liberação do mutex caso realize uma operação *lock* enquanto uma outra *thread* esteja executando sua sessão crítica. Neste caso o *lock* já foi pego, na execução da operação *unlock* uma das *threads* bloqueadas no *lock* será selecionada para "pegar" o mutex.

Na biblioteca Pthread, os mutex são disponibilizados através de um tipo de dado: `pthread_mutex_t`. As funções de manipulação de mutex são as seguintes:

A inicialização de um mutex requer que uma variável do tipo `pthread_mutex_t` já exita, sendo realizada através de uma invocação à:

```
pthread_mutex_init( pthread_mutex_t *m, pthread_mutexattr_t *atrib );
```

Onde `m` é o mutex a ser inicializado e `atrib` o atributo que define o valor inicial, aberto ou fechado, para o mutex. A opção *default* (aberto) pode ser selecionada passando `NULL` para `atrib`.

A manipulação permite realizar as operações de "pegar" e "soltar" o mutex, *lock* e *unlock*, respectivamente:

```
int pthread_mutex_lock( pthread_mutex_t *m );
int pthread_mutex_unlock( pthread_mutex_t *m );
```

Com o uso do mutex, o exemplo apresentado anteriormente seria:

```
int x;              // x é uma variável global
pthread_mutex_t m; // m é um mutex associado a variável x
pthread_mutex_init( &x, NULL );
```

`// Thread A`
`pthread_mutex_lock( &m );`
`a = x;  // lê dado global`
`a = a + 1; // adiciona`
`x = a;  // escreve dado global`
`pthread_mutex_unlock( &m );`
	`// Thread B`
	`pthread_mutex_lock( &m );`
	`b = x;  // lê dado global`
	`b = b - 1; // subtrai`
	`x = b;  // escreve dado global`
	`pthread_mutex_unlock( &m );`

Observe que o mutex `m` é uma variável como outra qualquer. Seu uso é de inteira responsabilidade do programador e depende da lógica adotada no algoritmo implementado. Neste exemplo, o mutex `m` está associado à variável compartilhada `x`. Cada sessão crítica que acessará a variável `x` deve explicitar as operações *lock* e *unlock* -- nenhuma garantia é dada sem a interferência explicita do programador no código.

2.2.7.2. Variáveis de condição

Em muitos algoritmos concorrentes, uma *thread* deve entrar em uma sessão crítica somente se ela obtém tanto o direito ao acesso exclusivo, utilizando *lock* em um mutex, como também obter a satisfação de uma determinada condição, por exemplo, o valor variável `x` é `y`). Caso contrário, o código da sessão crítica não deve ser executado. Para não ser necessário empregar um algoritmo que teste a intervalos regulares a variável `x`, as *threads* contam com um segundo mecanismo de sincronização, as variáveis de condição.

Uma variável de condição deve, obrigatoriamente, ser utilizada de forma associada a um mutex. Seu uso permite sincronizar duas (ou mais) *threads* em uma alteração de um estado, tipicamente, de uma posição de memória. Um exemplo característico é a sincronização de *threads* na implementação de um algoritmo do tipo produtor/consumidor (conforme o exemplo apresentado na seqüência).

Na biblioteca Pthread, uma variável de condição pode ser construída a partir do tipo: `pthread_cond_t`. Como no caso do mutex, a inicialização prevê que uma variável de condição tenha sido previamente declarada:

```
pthread_cond_init( pthread_cond_t *c, pthread_condattr_t *atrib );
```

Esta primitiva inicializa a variável de condição `c` com os atributos `atrib`. O valor para `atrib` permite indicar se a condição inicial encontra-se satisfeita (*default*, selecionada com `NULL`) ou não.

A manipulação de variáveis de condição se dá através das seguintes primitivas:

```
pthread_cond_wait( pthread_cond_t *c, pthread_mutex_t *m );
pthread_cond_signal( pthread_cond_t *c );
```

```
pthread_cond_broadcast( pthread_cond_t *c );
```

A primitiva *wait* faz com que a *thread* que a executou seja bloqueada na espera de uma sinalização. As duas outras primitivas permitem sinalizar que uma condição foi satisfeita, sendo que a primitiva *signal* sinaliza apenas uma das *threads* bloqueadas pela variável de condição, enquanto que a primitiva *broadcast* sinaliza todas as *threads* bloqueadas. Com *signal*, portanto, apenas uma *thread* sai do estado bloqueado, enquanto que *broadcast* permite liberar todas as *threads* aguardando uma mesma condição.

Um aspecto importante a considerar é que a condição tratada consiste em uma posição de memória compartilhada com, no mínimo, duas *threads*: a *thread* dependente da condição e a *thread* liberadora. Sendo assim, a presença de um mutex é obrigatória, e todas as primitivas de manipulação de uma variável de condição devem estar inseridas em uma sessão crítica protegida por *lock* e *unlock*. Para evitar uma situação de *deadlock*, no momento em que uma *thread* é bloqueada em *wait*, o mutex associado a condição é liberado automaticamente (por isto o parâmetro `m` na primitiva *wait*) e, no momento em que a *thread* bloqueada recebe uma sinalização, o mutex deve ser recuperado. Para isto, são executados, de forma implícita pelo *wait*, uma chamada a uma primitiva *unlock* no momento em que um *wait* é iniciado e a uma primitiva *lock* quando o *wait* for satisfeito.

Caso seja a sinalização provenha de um *broadcast*, apenas uma das *threads* obtém o mutex e prossegue a execução, as demais aguardam a liberação do mutex, mesmo tendo a condição satisfeita. Esta característica das variáveis de condição faz com que seja necessário um teste extra, para verificar se a condição continua *estando* satisfeita.

Abaixo um exemplo de um algoritmo produtor/consumidor utilizando variáveis de condição. A condição adotada é dados sejam consumidos (lidos de um *buffer*) apenas se existirem dados para serem lidos. Caso o *buffer* esteja vazio, o consumidor deve aguardar que as ao menos um item seja produzido. O algoritmo apresentado utiliza as funções de manipulação de variáveis de condição definidos pela interface POSIX.

```
// Área de memória compartilhada
Buffer b;           // Buffer de armazenamento temporário
int nb_itens = 0;   // Contador de ítens no buffer
pthread_mutex_t mb; // Proteção do buffer e do contador
pthread_cond_t c;   // Sincronização produtor/consumidor
```

```
void Produtor() {
 Item it;
 for( ; ; ) {
   it = ProduzItem();
   pthreads_mutex_lock( &mb );
  ArmazenaBuffer( b, it );
  nb_itens++;
  pthread_cond_signal( &c );
  pthreads_mutex_unlock(&mb);
 }
}
```

```
void Comsumidor() {
 Item it;
for( ; ; ) {
  pthreads_mutex_lock( &mb );
  while( nb_item <= 0 )
  pthread_cond_wait( &mb, &c
);
  it = LeBuffer();
  nb_itens--;
  pthreads_mutex_unlock(&mb);
 }
}
```

Observe no algoritmo, o trecho de código da *thread* consumidora em que a primitiva *wait* encontra-se em um *loop*. O teste é necessário pois nada garante que no momento em que a *thread* conseguir executar sua sessão crítica, a condição continura satisfeita.

Outro aspecto importante é que, ao contrário do mutex, uma sinalização em uma variável de condição (um *signal* ou um *broadcast*) não é memorizada. Em outras palavras, o sinal enviado garante que, naquele instante de tempo, a condição está satisfeita. Portanto, somente as *threads* em estado de *wait* quando a condição for satisfeita estão aptas à receber o sinal.

2.2.7.3. Semáforo

Muito embora não esteja presente em todas as implementações das bibliotecas *threads* POSIX, outro recurso bastante utilizado para o controle da evolução da execução de *threads* são os semáforos, cuja interface de serviços encontra-se definida pelo padrão IEEE POSIX 1003.1b.

Os semáforos, como o nome indica, permitem controlar o avanço de um grupo de *threads* sobre um trecho de código. Ao contrário das variáveis de condição, os sinais de um semáforo possui memória de estado. Este mecanismo de sincronização é composto de um contador, um valor inteiro que armazena o estado do semáforo. Este contador é acessado unicamente por operações P e V (do holandês *proberen* e *verhogen*, testar e incrementar, respectivamente\footnote{Outros autores atribuem a origem de P e V aos termos alemães *passeren*/*vrijgeven* (passar/liberar) [WIL 99].}. Diversos algoritmos podem ser empregados na manipulação do contador, mas o principio básico é de controlar quantas *threads* tem permissão de avançar e executar instruções sobre os recursos compartilhados.

Uma *thread* deve, antes de entrar em uma sessão crítica, requisitar a permissão de passagem, executando uma operação P. A realização da operação P decrementa o valor do contador do semáforo, caso o valor resultante seja maior ou igual à 0 (zero), a *thread* tem sua passagem liberada; caso contrário, a *thread* é bloqueada. A liberação da passagem é realizada através da operação V; esta operação incrementa o contador do semáforo e, caso hajam *threads* bloqueadas, uma das *threads* terá sua passagem liberada, sendo imediatamente decrementado o contador.

Observe que caso o semáforo assuma apenas valores 1 e 0, o comportamento é o mesmo apresentado pelo uso de mutex. Atingindo valores maiores que 1, o semáforo permite controlar o avanço de *threads*, neste caso o semáforo não está sendo usado para garantir exclusão mútua. Tal mecanismo é bastante útil em certos casos, onde o estado de um sinal necessita ser memorizado para um tratamento futuro. O algoritmo básico para um semáforo pode ser encontrado em [OLI 01] e uma possível implementação em [CAV 01].

2.2.7.4. Programando com *threads* em Linux

A interface para *threads* encontra-se disponível em Linux através da biblioteca Pthreads. Esta biblioteca é normalmente distribuída junto ao sistema operacional Linux, podendo ser utilizada em programas escritos em C/C++, com o compilador GNU C/C++. Para poder utilizar a Pthread em um programa, o programador deve fazer uso de dois arquivos oferecidos pela biblioteca: `pthread.h`, o arquivo de *header* e a biblioteca de funções `libpthread.a`. Apesar Pthread não oferecer semáforos, estes podem ser utilizados através dos serviços especificados pelo arquivo de *header* `semaphore.h`.

Os passos necessários para obter um executável a partir de um programa fonte são descritos abaixo.

Compilação Na compilação de cada módulo deve-se garantir que o arquivo *header* (o arquivo `.h`) da biblioteca Pthread possa ser encontrado pelo compilador; em geral este arquivo encontra-se no diretório `/usr/include`, que é o diretório *default* de *headers*. O comando abaixo, executado no diretório onde se encontra o fonte, gera o arquivo `OiMundo.o`, que corresponde ao código objeto do módulo `OiMundo.c`.

```
$ gcc -c OiMundo.c
```

Linkedição Todos os arquivos objetos gerados pela compilação, mais a biblioteca Pthread propriamente dita, o arquivo `libpthread.a`, devem ser linkeditados para formar o executável. O diretório *default* para busca de bibliotecas pelo linkeditor é o diretório `/usr/lib`, onde normalmente se encontra o arquivo `libpthread.a`. A inclusão da biblioteca pthread deve ser explicitada pelo parâmetro `-l` de linkedição. Assim, a linha de comando para a linkedição seria a seguinte:

```
$ gcc OiMundo.o -lpthread -o OiMundo
```

Neste caso, o programa executável é gerado no arquivo chamado `OiMundo`. Caso o programa tenha sido escrito em diversos módulos, estes devem ser compilados separadamente - observe a função `main` esteja definida uma única vez – e linkeditados em conjunto, gerando um único executável.

Execução Uma vez compilado e linkeditado, o executável pode ser lançado a partir da linha de comando, simplesmente executando :

```
$ OiMundo
```

Observe que o diretório corrente, ou seja, o diretório `.` (ponto), esteja no caminho de busca dos arquivos executáveis (variável de ambiente `$PATH`). Caso não esteja, digite opcionalmente `./OiMundo`.

2.2.8. Exemplo

Nesta seção são apresentadas duas soluções ao problema de busca do maior elemento de um vetor de números inteiros: uma primeira utilizando mutex para compartilhar dados entre tarefas e uma segunda, utilizando apenas o mecanismo de criação e junção (*create* e *join*) de *threads*.

O algoritmo básico é bastante simples, consistindo em percorrer todo o vetor na busca do maior elemento. Nas implementações realizadas, assume-se duas entradas *n* e *m*, onde *n* indica o tamanho do vetor e *m* o número de *threads* que devem ser criadas (na primeira implementação) ou o tamanho a partir do corte do vetor a partir do qual o cálculo deva ser realizado seqüencialmente (na segunda implementação). Por uma questão de espaço, as funções de busca seqüencial do maior elemento em um vetor e de geração randomica foram suprimidas.

A primeira solução implementa um algoritmo que cria um número fixo de *threads*, determinado por *m*, onde cada uma é responsável pela busca do maior elemento em um intervalo, de tamanho *n/m*, do vetor original. Uma variável global recebe o maior elemento do vetor - cada *thread*, ao encontrar o maior elemento no seu intervalo, verifica se o valor encontrado é maior que o encontrado até o momento, caso seja,

atualiza com o novo valor. Ao final de todas as *threads*, a variável compartilhada possui o maior valor encontrado.

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
int *vet;          // vetor inicial
int  maior;        // maior valor, variável compartilhada
pthread_mutex_t m; // acesso a variável maior
void* procuraSeq( void *in ) {
 int i = ((Intervalo *)in)->i,    // cast, obtendo os parâmetros
     f = ((Intervalo *)in)->f;     // enviados a thread
 int aux = achaMaior( vet, i, f ); // busca sequencial
pthread_mutex_lock(&m);
 if( maior < aux ) maior = aux; // sessão crítica
 pthread_mutex_unlock(&m);
 return NULL;
}
void main( int argc, char **argv ) {
 int i, n = atoi(argv[1]), t = atoi(argv[2]);
 Intervalo *inter = (Intervalo *) malloc(t * sizeof(Intervalo));
 pthread_t *id = (pthread_t *) malloc(t * sizeof(pthread_t));
 pthread_mutex_init( &m, NULL );
 vet = geraVetor(n);
 maior = vet[0];  // inicializa variável compartilhada
for( i = 0 ; i < (t-1) ; i++ ) { // criação das threads
  inter[i].i = i * n/t; inter[i].f = (i+1) * n/t;
  pthread_create(&(id[i]),NULL,procuraSeq, (void*)&(inter[i]) );
 }
 inter[i].i = i * n/t; inter[i].f = t;
 pthread_create(&(id[i]),NULL,procuraSeq,(void *)&(inter[i]) );
for( i = 0 ; i < t ; i++ ) // aguarda término das threads
  pthread_join( id[i], NULL );
  printf("Maior = %d\n", maior );
  free(id); free(vet);
}
```

Na segunda implementação não foi utilizada memória compartilhada nem sessões críticas, sendo a comunicação entre as *threads* realizada através da passagem de parâmetros e retorno de resultados. A técnica adotada cria *threads* recursivamente, cada uma recebendo um vetor de tamanho *n'* e criando outras duas *threads*, cada uma recebendo um vetor de tamanho *n'/2*, enquanto que *n'/2 > m*: neste limite é realizada uma busca seqüencial do maior elemento do vetor. Ao final, a *thread* que realizou a busca seqÜencial retornao o maior elemento encontrado no seu intervalo, cabendo a *thread* que a criou decidir qual o maior valor dentre os retornados pelas duas *threads* criadas recursivamente.

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
int *vet; // vetor inicial
void* buscaRec( void *in ) {
 int i = ((Intervalo *)in)->i, // cast, obtendo os parâmetros
     f = ((Intervalo *)in)->f, // enviados a thread
     s = ((Intervalo *)in)->s; //se (f-i)<s,calcula seqüencial
 int *maior, *aux;
 if( (f - i) < s ) { // atingiu tamanho mínimo ?
  maior = (int *) malloc(sizeof(int));
```

```
  *maior = achaMaior( vet, i, f ); // Sim, calcula seqüencial
 } else { // Não, mais um nível de recursão
  Intervalo inter[2];
  pthread_t id[2];  // id[0] : busca no intervalo [0,(f-1)/2)
                    // id[1] : busca no intervalo [(f-1)/2,f)
  inter[0].i = i; inter[0].f = i+(f-i)/2; inter[0].s = s;
  pthread_create(&(id[0]),NULL, buscaRec, (void *)&(inter[0]) );
  inter[1].i = i+(f-i)/2; inter[1].f = f; inter[1].s = s;
  pthread_create(&(id[1]), NULL, buscaRec,(void *)&(inter[1]) );
  pthread_join(id[0], (void *)&maior ); // Aguarda primeiro
                                        // intervalo
  pthread_join(id[1], (void *)&aux );   // Aguarda segundo
                                        // intervalo
  if( *aux > *maior ) *maior = *aux;
  free(aux);
 }
 return (void *)maior; // O maior encontrado neste nível
}
void main( int argc, char **argv ) {
 int *maior, n = atoi(argv[1]), s = atoi(argv[2]);
 Intervalo inter;
 pthread_t id;
 geraVetor(n); // cria um vetor com elementos aleatórios
inter.i = 0; inter.f = n; inter.s = s;
 pthread_create(&id, NULL, buscaRec, (void *)&inter );
 pthread_join( id, (void *)&maior );
printf("Maior = %d\n", *maior );
 free(maior); free(vet);
}
```

2.2.9. *Threads* e outras bibliotecas

Um comentário final sobre o uso de *threads* diz respeito ao cuidado que o programador deve ter na seleção das outras bibliotecas que deverão compor seu programa. Pelo simples fato de que diversas *threads* encontram-se executando de forma concorrente, é possível que, em determinado instante de tempo, duas (ou mais) destas *threads* invoquem serviços de uma mesma biblioteca. Para que os serviços sejam executados de forma correta, é impressindivel que esta biblioteca tenha sido desenvolvida para ser *thread-safe*, ou seja, suportar a execução de serviços de forma simultânea sem que um erro seja produzido. Uma técnica comum na solução deste problema faz uso de sessões críticas: cada um dos serviços da biblioteca é considerado uma sessão crítica. Assim, uma vez que um de seus serviços esteja ativo, o simples uso de um mutex permite o bloqueio de qualquer outro fluxo de execução em que for realizado outra invocação à esta mesma biblioteca.

Esta técnica, embora garanta a funcionalidade do programa, implica em limitar a concorrência na execução do programa, em uma restrição indesejada quando o objetivo é o aumento de desempenho. Mecanismos mais elaborados permitem o desenvolvimento de bibliotecas *thread aware*, onde além de garantir o correto funcionamento dos serviços, possibilitam que diversos serviços estejam executando de forma concorrente.

2.3. PVM

A biblioteca PVM [GEI 94] é uma infra-estrutura de *software* que emula um sistema com memória distribuída num ambiente de rede heterogêneo, que pode ser composto por uma grande variedade de máquinas diferentes (de PC's a supercomputadores). O PVM permite que se crie uma ***máquina virtual*** composta de um número praticamente ilimitado de *hosts* heterogêneos.

Diversos usuários podem configurar diferentes máquinas virtuais ao mesmo tempo (sobrepostas) e cada usuário pode executar diversas aplicações PVM simultaneamente. O PVM fornece funções para a criação de processos (*tasks*) na máquina virtual e permite que estes se comuniquem e se sincronizem uns com os outros. Um processo ou *task* é definido como uma unidade computacional em PVM e é análogo a um processo Unix (em geral **é** um processo Unix).

As aplicações, escritas em C ou Fortran, podem ser paralelizadas através das primitivas para troca de mensagens que são comuns na maioria dos sistemas com memória distribuída. Com a troca de mensagens implementada através de primitivas do tipo *send*/*receive*, os vários processos que compõem a aplicação podem cooperar para a resolução do problema em paralelo.

O modelo computacional do PVM assume que qualquer processo pode enviar uma mensagem para qualquer outro processo PVM e que não há limite para o tamanho ou para o número destas mensagens. O modelo de comunicação do PVM fornece envio (*send*) e recepção (*receive*) de mensagens assíncronas e ainda recepção síncrona. Note-se que o envio de mensagens é de uma certa forma bloqueante, já que o processo emissor espera até que o *buffer* de envio esteja livre para reutilização. Observa-se ainda que a recepção síncrona (ou bloqueante) difere da assíncrona (não-bloqueante) por colocar o processo receptor em estado de espera até que a mensagem que lhe é destinada chegue ao *buffer* de recepção.

Além das primitivas de comunicação mencionadas, a biblioteca PVM ainda fornece funções para:

- Inicialização e término de processos PVM;
- Adição e remoção de *hosts* da máquina virtual;
- Sincronização e envio de sinais entre processos PVM;
- Obtenção de informações sobre a configuração da máquina e processos;
- Empacotamento e desempacotamento de dados;
- Difusão de mensagens (*multicast* e *broadcast*);
- Criação dinâmica de grupos de processos.

2.3.1. Manipulação da máquina virtual

O sistema PVM é composto de duas partes. Uma é a biblioteca de funções paralelas, abordada mais adiante. A outra é o *daemon*, chamado pvmd3 (ou apenas pvmd), que reside em todos os computadores que fazem parte da máquina virtual. São processos que executam em segundo plano e que são responsáveis pelo gerenciamento das tarefas paralelas oferecidas pelo PVM. Pode-se dizer que o conjunto de *daemons* do PVM, executando em diferentes máquinas, forma a máquina virtual do PVM. O PVM pode ser obtido a partir de sua *home page* em [PVM 01].

O PVM oferece um console, através de um *prompt* próprio, para que seja possível interagir com a máquina virtual. Este console pode ser acessado a partir de

qualquer máquina que tenha um *daemon* executando (e que conseqüentemente faça parte da máquina virtual do PVM). Para iniciar o PVM e acessar seu *prompt*, deve-se digitar, em qualquer máquina que tenha o PVM instalado:

```
% pvm
```

A partir daí, o *prompt* estará ativo, indicando que o PVM está executando nesta máquina.

```
pvm>
```

Os principais comandos do PVM para configuração e manipulação da máquina virtual são os seguintes:

- pvm> add *hostname*

adiciona uma máquina à máquina virtual do PVM

- pvm> delete *hostname*

remove uma máquina da máquina virtual

- pvm> conf

mostra a configuração da máquina virtual

- pvm> ps –a

mostra quais tarefas estão executando em cada máquina

- pvm> quit
  devolve o *prompt* ao sistema operacional, mas continua a executar o PVM
- pvm> halt
  termina (mata) todas as tarefas PVM, destrói a máquina virtual e sai do console
- pvm> help

ajuda que contém todas as funcionalidades do console PVM

Não é necessário adicionar manualmente as máquinas a cada seção. Ao invés disso, pode-se fornecer um arquivo com os nomes das máquinas no momento da inicialização do PVM. Este arquivo deve listar os nomes das máquinas de forma a que fique um nome em cada linha. A inicialização é feita através do seguinte comando:

```
% pvm hostfile
```

Assim, o PVM irá adicionar as máquinas contidas no *hostfile* simultaneamente, antes do aparecimento do *prompt*.

2.3.2. Interface de programação

Abaixo são descritas as principais primitivas da biblioteca PVM.

```
int tid = pvm_mytid (void);
```

Retorna o identificador da tarefa que faz a chamada a essa função. Este identificador é gerado no momento em que esta foi criada pela aplicação (**tid** - *task identifier*).

```
ptid = pvm_parent();
```

Retorna o identificador (**ptid**) do pai (processo criador da tarefa).

```
int info  = pvm_config(    int *nhost, int *narch, struct pvmhostinfo **hostp );
```

Retorna informações sobre a máquina virtual, inclusive o número de máquinas (**nhost**) e o número de arquiteturas diferentes (**narch**). **hostp** é um ponteiro para um vetor de estruturas do tipo *pvmhostinfo*. Este vetor terá o tamanho de no mínimo **nhost** posições. No retorno, cada uma delas conterá o TID do *pvmd*, o nome da máquina, e a relativa velocidade da máquina na configuração.

```
int numt = pvm_spawn(char *task,char **argv,int flag,char *where,int ntask,int *tids);
```

Inicia **ntask** cópias do executável **task** na máquina virtual. **argv** é um ponteiro para um array de argumentos para **task**, com seu final em NULL. Se não há argumentos para **task**, então **argv** é NULL. O argumento **flag** é usado para especificar as opções, e é uma soma entre:

Value	Option	Meaning
0	PvmTaskDefault	PVM escolhe a máquina onde executará os processos.
1	PvmTaskHost	**where** é uma string que indica a máquina para execução.
2	PvmTaskArch	**where** indica a arquitetura (PVM_ARCH) para execução.
4	PvmTaskDebug	inicia as tarefas sob o controle de um depurador.
8	PvmTaskTrace	gera dados de monitoração.
16	PvmMppFront	inicia as tarefas num *front-end* MPP.
32	PvmHostCompl	complementa o argumento **where**

```
int info = pvm_kill (int tid);
```

Termina (mata) outro processo PVM identificado por **tid**.

```
int bufid = pvm_initsend (int encoding);
```

O PVM utiliza um *buffer* de saída para envio de mensagens (as mensagens são empacotadas e colocadas neste *buffer* antes do envio). O *buffer* é inicializado com esta rotina, que limpa o *buffer* de envio e cria um novo para o empacotamento de uma nova mensagem. O esquema de codificação é definido pelo **encoding** (a opção **PvmDataDefault** é a mais utilizada). A rotina retorna um identificador para o buffer em **bufid**.

```
int info = pvm_pkbyte (char  *cp, int nitem, int stride );
int info = pvm_pkcplx (float  *xp, int nitem, int stride );
int info = pvm_pkdcplx (double *zp, int nitem, int stride );
int info = pvm_pkdouble (double *dp, int nitem, int stride );
int info = pvm_pkfloat ( float  *fp, int nitem, int stride );
int info = pvm_pkint (int  *np, int nitem, int stride );
int info = pvm_pklong (long   *np, int nitem, int stride );
int info = pvm_pkshort (short  *np, int nitem, int stride );
int info = pvm_pkstr (char  *cp );
```

São as rotinas de empacotamento de dados (o tipo do dado está indicado após as letras **pk**). O primeiro parâmetro corresponde a um ponteiro para o primeiro item a ser

empacotado. **nitem** é o número total de itens a serem empacotados. **stride** igual a **1** (valor comumente utilizado) indica que os itens estão contíguos na memória.

```
int info = pvm_send (int tid, int msgtag);
```

Envia a mensagem identificada por **msgtag** para o processo **tid**. O argumento **msgtag** funciona como um qualificador para a mensagem, que deverá ser o mesmo na recepção da mensagem pelo processo destino, caso contrário, a mensagem não será recebida.

```
int info = pvm_mcast (int *tids, int ntask, int msgtag);
```

Envia a mensagem identificada por **msgtag** para todos os processos presentes no vetor **tids** (exceto para ele mesmo). O tamanho de **tids** é igual a **ntask**.

```
int info = pvm_psend (int tid, int msgtag, void *vp, int cnt, int type);
```

Empacota e envia um *array* do tipo **type**, que pode ser:

PVM_STR	PVM_FLOAT	PVM_BYTE	PVM_CPLX
PVM_SHORT	PVM_DOUBLE	PVM_INT	PVM_DCPLX
PVM_LONG	PVM_UINT	PVM_USHORT	PVM_ULONG

```
int bufid = pvm_recv (int tid, int msgtag);
```

Rotina de recepção de mensagens (*receive*) que fará com que a tarefa fique bloqueada até que a mensagem **msgtag** chegue do processo **tid**. O valor **–1** para **msgtag** e/ou **tid** faz com que o *receive* funcione para qualquer valor definido no envio da mensagem (*wildcard*). A mensagem é armazenada no *buffer* ativo e deverá ser desempacotada.

```
int bufid = pvm_nrecv (int tid, int msgtag);
```

Rotina de recepção de mensagens (*receive*) que não bloqueia a tarefa, mas que verifica se uma mensagem chegou ou não (esta rotina pode ser invocada várias vezes). Se a mensagem não chegou, o **bufid** retornará com valor **0**. Caso contrário, a rotina cria um novo *buffer* e retorna seu identificador em **bufid**.

```
int bufid = pvm_probe (int tid, int msgtag);
```

Apenas verifica se uma mensagem chegou e retorna um *buffer* para ela, mas não a recebe efetivamente.

```
int bufid = pvm_trecv (int tid, int msgtag, struct timeval *tmout);
```

Rotina de recepção de mensagens (*receive*) que bloqueará a tarefa até a chegada de uma mensagem qualificada ou até que o tempo em **tmout** se esgote.

```
int info = pvm_bufinfo (int bufid, int *bytes, int *msgtag, int *tid);
```

Retorna o **msgtag**, o **tid** do emissor e o tamanho em **bytes** da mensagem identificada pelo **bufid**. Pode ser usado para recuperar informações de mensagens recebidas com *wildcards*.

```
int info = pvm_upkbyte (char  *cp, int nitem, int stride);
int info = pvm_upkcplx (float  *xp, int nitem, int stride);
int info = pvm_upkdcplx (double *zp, int nitem, int stride);
int info = pvm_upkdouble (double *dp, int nitem, int stride);
int info = pvm_upkfloat (float  *fp, int nitem, int stride);
int info = pvm_upkint (int  *np, int nitem, int stride);
int info = pvm_upklong (long  *np, int nitem, int stride);
int info = pvm_upkshort (short  *np, int nitem, int stride);
int info = pvm_upkstr(char   *cp );
```

Rotinas de desempacotamento mensagens recebidas no *buffer* ativo. O tipo de dado a ser desempacotado é indicado após as letras **upk** e devem corresponder aos argumentos usados pelo emissor no momento do empacotamento.

```
int info=pvm_precv (int tid,int msgtag,void *vp,int cnt,int type,int *rtid,int *rtag, int*rcnt);
```

Combina as funções de recepção bloqueante e desempacotamento de mensagens. O tipos de dados possíveis correspondem aos da rotina **pvm_psend**.

```
int info = pvm_exit();
```

Avisa o *daemon* que este processo está deixando o PVM (o processo pode continuar a executar, desde que não faça chamadas às funções do PVM).

2.3.3. Desenvolvimento de aplicações

Como mencionado anteriormente, o PVM oferece um conjunto de primitivas ou funções que podem ser utilizadas por programas escritos em C ou Fortran para expressar o paralelismo em aplicações paralelas.

O desenvolvimento de uma aplicação normalmente é realizado a partir do desenvolvimento de um conjunto de programas seqüenciais, onde cada um é responsável por uma parte do processamento. O PVM é utilizado basicamente para a inicialização de novas tarefas e troca de informações (mensagens) entre elas.

Entre os modelos de programação paralela que são possibilitados pelo PVM está o mestre-escravo, que será brevemente descrito aqui a título de ilustração. Neste modelo, tem-se um programa principal, o mestre (*master*), responsável pela inicialização das demais tarefas escravas (*slaves*) e pela coleta de resultados. As tarefas escravas, por sua vez, devem receber os dados, processa-los, e enviar uma resposta (este ciclo acontece quantas vezes forem necessárias). Outros modelos existem, mas a descrição detalhada dos mesmos foge ao escopo deste texto. A tab. 2.1 demonstra o algoritmo básico de uma aplicação mestre-escravo [SEI 01]:

Tabela 2.1: interação do tipo mestre-escravo

Mestre	Escravos
- dispara os escravos » parâmetros *(arg*v) - transmite perguntas » aloca *buffer* » empacota dados » qualifica mensagem » envia *buffer*	- esperam perguntas
- espera respostas	- recebem perguntas » testam o qualificador » recebem *buffer* » desempacotam dados - processam os dados - transmitem respostas » aloca *buffer* » empacotam dados » qualificam mensagem » enviam
– recebe respostas » recebe *buffer* » desempacota dados	

2.3.4. Programa exemplo em PVM: hello e hello_other

O programa abaixo é constituído de dois processos: **hello.c** e **hello_other.c**. O primeiro, após imprimir sua identificação, cria o outro através da função *spawn*. Feito isso, ele espera por uma mensagem de seu processo filho recém criado, desempacota a mensagem e imprime na tela. O segundo, por sua vez, assim que é criado, envia uma mensagem ao pai incluindo nela qual é a máquina em que está executando.

```
//hello.c
main()
{
   int cc, tid, msgtag;
   char buf[100];
   /* imprime o identificador do processo atual, obtido por
   pvm_mytid()*/
   printf("eu sou t%x\n", pvm_mytid());
  /*inicia outro processo cujo executável tem o nome hello_other
   o identificador desta tarefa é retornado em tid
   os demais parâmetros indicam que não há argumento de entrada,
   a máquina será escolhida pelo pvm e será iniciada apenas uma
   tarefa */
   cc = pvm_spawn("hello_other", (char**)0, PvmTaskDefault, "",
   1,      &tid);
   if (cc == 1) {
      msgtag = 1;
      /* bloqueia esperando a resposta que deve ser originária de
      tid e deve ter msgtag = 1 */
      pvm_recv(tid, msgtag);
    //após recebimento, desempacota a mensagem, que é uma string
      pvm_upkstr(buf);
      /* imprime mensagem contendo o tid do emissor e a string
      recebida */
```

```
      printf("recebido de t%x: %s\n", tid, buf);
   } else
      printf("não consegui iniciar hello_other\n");

 pvm_exit();            //sai do pvm
}

//hello_other.c
#include "pvm3.h"
main()
{
   int ptid, msgtag;
   char buf[100];
   ptid = pvm_parent();            //obtém o tid do pai

   strcpy(buf, "hello, world from ");    //monta mensagem

   gethostname(buf + strlen(buf), 64);
   msgtag = 1;

   //inicia o buffer de envio
   pvm_initsend(PvmDataDefault);

   //empacota a mensagem numa string
   pvm_pkstr(buf);

   //envia a mensagem para o processo pai
   pvm_send(ptid, msgtag);

   //sai do pvm
   pvm_exit();
}
```

2.3.5. Compilação

O PVM traz consigo um aplicativo para compilação que é independente de arquitetura, o *aimk* (correspondente ao *make*). O *aimk* se baseia no arquivo chamado Makefile, que contém as regras e as dependências necessárias para a compilação, gerar o executável. Além disso, o *aimk* tem mecanismos que permitem a compilação em diferentes arquiteturas ao mesmo tempo. O aplicativo *make* também pode ser usado para a compilação. A listagem abaixo contém um exemplo de arquivo Makefile. Este arquivo deve estar presente no diretório onde estão os arquivos fonte e os comandos *make* ou *aimk* (deve estar no *path*) podem ser utilizados. Este Makefile considera a compilação em arquitetura Linux.

```
SHELL       =     /bin/sh
#localização do pvm
PVMDIR      =     /usr/local/pvm3

#opção que permite depuração
CFLOPTS     =     -g

#opções de compilação
CFLAGS      =     $(CFLOPTS) -I$(PVMDIR)/include $(ARCHCFLAGS)
PVMLIB      =     -lpvm3

#############################################################
```

```
LIBS        =     $(PVMLIB) $(ARCHLIB)
GLIBS       =     -lgpvm3
LDFLAGS     =     $(LOPT) -L$(PVMDIR)/lib/$(PVM_ARCH)
CPROGS      =     hello hello_other

default:    hello hello_other

all: $(CPROGS)

clean:
      rm -f *.o $(CPROGS)

hello:  hello.c
      $(CC) $(CFLAGS) -o $@ hello.c $(LDFLAGS) $(LIBS)

hello_other:  hello_other.c
      $(CC) $(CFLAGS) -o $@ hello_other.c $(LDFLAGS) $(LIBS)
```

2.3.6. Execução

Para a execução, primeiro deve-se iniciar a máquina virtual do PVM (por exemplo, digitando-se "pvm", adicionando-se uma ou mais máquinas a partir da console e saindo-se da console através do comando "quit", que deixa o PVM ativo). Feito isto, basta digitar o nome do executável Linux (por exemplo, "hello").

2.3.7. Programa exemplo em PVM: master e slave

O programa a seguir ilustra dois tipos de comunicação: *multicast* (da tarefa mestre para as escravas) e ponto a ponto (entre as escravas), além de outras primitivas do PVM. A compilação e a execução são realizadas de forma análoga à do exemplo anterior.

```
//master.c
#include <stdio.h>
#include "pvm3.h"
#define SLAVENAME "slave1"
main()
{
    int mytid;                /* meu id */
    int tids[32];       /* ids das tarefas escravas */
    int n, nproc, numt, i, who, msgtype, nhost, narch;
    float data[100], result[32];
    struct pvmhostinfo *hostp;

    /* verifica tid da tarefa atual */
    mytid = pvm_mytid();

    /* seta numero de tarefas escravas de acordo com num. de
    máquinas */
    pvm_config( &nhost, &narch, &hostp );
    nproc = nhost * 3;
    if( nproc > 32 ) nproc = 32 ;
      printf("Spawning %d worker tasks ... " , nproc);
/* inicia tarefas escravas */
    numt=pvm_spawn(SLAVENAME, (char**)0, 0, "", nproc, tids);
    if( numt < nproc ){
    //Erro ao iniciar as escravas - imprime mensagem e finaliza
```

```
         printf("\n Trouble spawning slaves. Aborting. Error codes
         are:\n");
         for( i=numt ; i<nproc ; i++ ) {
            printf("TID %d %d\n",i,tids[i]);
         }
         for( i=0 ; i<numt ; i++ ){
            pvm_kill( tids[i] );
         }
         pvm_exit();
         exit(1);
      }
        printf("SUCCESSFUL\n");

      /* início do programa */
      n = 100;
      /* initialize_data( data, n ); */
      for( i=0 ; i<n ; i++ ){
         data[i] = 1.0;
      }
/* distribui valores iniciais para todas as tarefas escravas */
        pvm_initsend(PvmDataDefault); //inicializa buffer
        pvm_pkint(&nproc, 1, 1);      //empacota numero escravos
        pvm_pkint(tids, nproc, 1);    //vetor com tids dos escravos
        pvm_pkint(&n, 1, 1);    //valor de n (tamanho de data[])
        pvm_pkfloat(data, n, 1);      //vetor data[]
        pvm_mcast(tids, nproc, 0);    //envia

      /* espera resultados dos escravos */
      msgtype = 5;  //qualificador das mensagens esperadas
      for( i=0 ; i<nproc ; i++ ){ //recebe, desempacota e imprime
         pvm_recv( -1, msgtype );
         pvm_upkint( &who, 1, 1 );
         pvm_upkfloat( &result[who], 1, 1 );
         printf("I got %f from %d; ",result[who],who);
         if (who == 0)
              printf("(expecting %f)\n", (nproc - 1) * 100.0);
         else
              printf("(expecting %f)\n", (2 * who - 1) * 100.0);

      }
      /* Program Finished exit PVM before stopping */
      pvm_exit();
}

//slave.c
#include <stdio.h>
#include "pvm3.h"

main()
{
      int mytid;       /* meu identificador pvm */
      int tids[32];    /* identificadores das tarefas escravas */
        int n, me, i, nproc, master, msgtype;
        float data[100], result;
      float work();

      /* verifica identificador da tarefa atual */
        mytid = pvm_mytid();

      /*recebe dados do mestre - os dados e a ordem de recebimento
      devem ser compatíveis com o envio*/
```

```
    msgtype = 0;
    pvm_recv( -1, msgtype );
    pvm_upkint(&nproc, 1, 1);
    pvm_upkint(tids, nproc, 1);
    pvm_upkint(&n, 1, 1);
    pvm_upkfloat(data, n, 1);

    /* determina a posição da tarefa (0 -- nproc-1) */
    for( i=0; i<nproc ; i++ )
       if( mytid == tids[i] ){ me = i; break; }

    /* realiza cálculos com os dados recebidos */
    result = work( me, n, data, tids, nproc );

    /* envia resultado ao mestre */
    pvm_initsend( PvmDataDefault );//inicia buffer
    pvm_pkint( &me, 1, 1 );        //empacota posição
    pvm_pkfloat( &result, 1, 1 );  //empacota resultado
    msgtype = 5;                   //define qualificador
    master = pvm_parent();         //verifica id do mestre
    pvm_send( master, msgtype );   //envia a mensagem

    /* Program finished. Exit PVM before stopping */
    pvm_exit();
}
//função que realiza os cálculos
float work(me, n, data, tids, nproc )
//exemplo simples: escravos trocam dados com vizinho da esquerda
    int me, n, *tids, nproc;
    float *data;
{
    int i, dest;
    float psum = 0.0;
    float sum = 0.0;
    for( i=0 ; i<n ; i++ ){
       sum += me * data[i];
    }
    /* comunicação ponto a ponto */
    pvm_initsend( PvmDataDefault );//inicia buffer
    pvm_pkfloat( &sum, 1, 1 );           //empacota dado
    dest = me+1;              //define destino
    if( dest == nproc ) dest = 0;
    pvm_send( tids[dest], 22 );    //envia mensagem
    pvm_recv( -1, 22 );       //espera dado do vizinho
    pvm_upkfloat( &psum, 1, 1 );   //desempacota

    return( sum+psum );       //retorna resultado
}
```

2.4. MPI

MPI é uma biblioteca de comunicação que permite a programação paralela baseada em troca de mensagens. Foi definida pelo MPI Fórum (www.mnpi-forum.org), com a participação de Universidades, empresas, laboratórios de pesquisa. A versão 1.0, definida no MPI Fórum, se tornou disponível em maio de 1994, e contém as especificações técnicas da interface de programação.

Em MPI uma execução compreende um ou mais processos que se comunicam chamando rotinas da biblioteca para enviar e receber mensagens. Este conjunto de rotinas pode ser utilizado a partir de programas escritos em ANSIC ou Fortran

Um programa MPI é formado por um conjunto fixo de processos, criados no momento da inicialização, sendo que é criado um processo por processador. Cada um desses processos pode executar um programa diferente, o que caracteriza o modelo de programação MPMD. No entanto, uma forma natural de programar é utilizando o modelo SPMD, no qual um mesmo programa é disparado em cada um dos processadores participantes da execução e em cada processador é selecionado para execução um trecho do programa.

O padrão MPI define funções para:

- Comunicação ponto a ponto;
- Operações coletivas;
- Grupos de processos;
- Contextos de comunicação;
- Ligação para programas ANSI C e Fortran 77;
- Topologia de processos.

A versão 1.1 de MPI possui um enorme conjunto de funções (129). No entanto, com um número reduzido (apenas 6) é possível resolver uma grande variedade de problemas. Neste documento serão apresentadas as funções básicas de comunicação ponto a ponto que, juntamente com mais um numero reduzido de funções permite o desenvolvimento de programas. Serão também apresentados exemplos de programas escritos em MPI e as principais características de MPI-2, versão definida em 1997 que incorpora novas características de programação paralela à biblioteca MPI.

2.4.1. Conceitos básicos

A seguir serão apresentados alguns conceitos básicos MPI.

Processo

Cada programa em execução se constitui um processo. Desta forma, o número de processadores especificado pelo usuário quando dispara a execução do programa indica o número de processos (programas) em execução. Se o número de processadores físicos é menor que o número especificado, os processos são criados, circularmente, de acordo com a lista de processadores especificada na configuração do ambiente.

Mensagem

É o conteúdo de uma comunicação, formada por duas partes:

- **Envelope**

  Endereço (origem ou destino). É composto por três parâmetros:

  - Identificação dos processos (transmissor e receptor);
  - Rótulo da mensagem;
  - Comunicator.

- **Dado**

  Informação que se deseja enviar ou receber. É representado por três argumentos:
  - Endereço onde o dado se localiza;
  - Número de elementos do dado na mensagem;
  - Tipo do dado. Os tipos de dados na linguagem C são:

Tipos de Dados Básicos no C	
Definição no MPI	Definição no C
MPI_CHAR	signed char
MPI_INT	signed int
MPI_FLOAT	float
MPI_DOUBLE	Double
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_LONG_DOUBLE	long double
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_SHORT	Signed short int
MPI_BYTE	-
MPI_PACKED	-

Rank

Todo o processo tem uma identificação única atribuída pelo sistema quando o processo é inicializado. Essa identificação é contínua representada por um número inteiro, começando de zero até N-1, onde N é o número de processos. É utilizado para identificar o processo destinatário de uma mensagem, na operação send, e o processo remetente de uma mensagem, na operação receive.

Group

Group é um conjunto ordenado de N processos. Todo e qualquer group é associado a um communicator muitas vezes já predefinido como "MPI_COMM_WORLD". Inicialmente, todos os processos são membros de um group com um communicator.

Communicator

O communicator é um objeto local que representa o domínio (contexto) de uma comunicação (conjunto de processos que podem ser endereçados).

2.4.2. Primitivas Básicas MPI

A seguir serão apresentadas as primitivas básicas de MPI.

MPI_Init(&argc , &argv)

Inicializa uma execução em MPI, é responsável por copiar o código do programa em todos os processadores que participam da execução. Nenhuma outra função MPI pode aparecer antes de MPI_INIT. argc, argv são variáveis utilizadas em C para recebimento de parâmetros.

MPI_Finalize()

Termina uma execução MPI. Deve ser a última função em um programa MPI.

MPI_Comm_Size(*communicator* , *&size*)

Determina o número de processos em uma execução. *communicator* indica o grupo de comunicação e &size contém, ao término da execução da primitiva, o número de processos no grupo.

MPI_Comm_Rank(*communicator , &pid*)

Determina o identificador do processo corrente. *communicator* indica o grupo de comunicação e *&pid* identifica o processo no grupo.

MPI_Send (*&buf, count, datatype, dest, tag, comm*)

Permite a um processo enviar uma mensagem para um outro. É uma operação não bloqueante. O processo que a realiza continua sua execução. Os parâmetros são:

&buf: endereço do buffer de envio
count: número de elementos a enviar
datatype: tipo dos elementos a serem enviados
dest: identificador do processo destino da mensagem
tag: tipo da mensagem
comm: grupo de comunicação

MPI_Recv (*&buf, count, datatype, dest, tag, comm*)

Função responsável pelo recebimento de mensagens. É uma operação bloqueante. O processo que a executa fica bloqueado até o recebimento da mensagem. Os parâmetros são:

&buf: endereço do buffer de recebimento
count: número de elementos a enviar
datatype: tipo dos elementos a serem enviados
dest: identificador do processo remetente da mensagem
tag: tipo da mensagem
comm: grupo de comunicação
status: status de operação

Exemplo de programa

A seguir será apresentado um programa simples, no qual um processo envia uma mensagem para um outro, que a imprime.

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv) {
  int my_rank;  /* Identificador do processo */
  int n;     /* Número de processos */
  char c[5] ;
  MPI_Status status ;

   MPI_Init(&argc , & argv);
   MPI_Comm_Rank(MPI_COMM_WORLD, &my_rank) ;
   MPI_Comm_Size(MPI_COMM_WORLD, &n) ;
   if (n != 2) exit() ;
   if (my_rank == 0) {
         strcpy (c, "alo") ;
         MPI_Send (c, strlen (c),  MPI_CHAR, 1, 99, MPI_COMM_WORLD) ;
   else {
         MPI_Recv (c, 5,  MPI_CHAR, 0, 99, MPI_COMM_WORLD, status) ;
         Printf("%s\n", c) ;
         }
   MPI_Finalize();
  }
```

O programa acima é formado por dois processos. Cada processo obtém sua identificação (my_rank) e se o numero de processos for diferente de 2 o programa termina (é para ser executado por dois processos). O processo 0 envia uma mensagem contendo "alo" para o processo 1 com a primitiva Send. O processo 1 recebe a mensagem, primitiva Recv, e a imprime.

2.4.3. Modelos de programas paralelos

Um programa paralelo é composto por processos comunicantes que executam em processadores diferentes e que cooperam para a resolução de um cálculo. Os modelos que podem ser utilizados pelo programador para o desenvolvimento de suas aplicações são:

Divisão e conquista

Esta técnica consiste na criação de processos filhos para executar partes menores de uma tarefa. Os filhos executam e devolvem os resultados ao processo pai.

Pipeline

Os processos cooperam por troca de mensagens. Um conjunto de processos formam um pipeline, com a troca de informações em um fluxo contínuo. Para enviar dados a um processo sobre um outro processador, o processo remetente deve agrupar os

dados e remeter. O processo recebedor extrai os dados de uma mensagem recebida, processa e envia para o processo seguinte no pipeline.

Mestre/escravo

O programa é organizado como sendo formado por um processo (Mestre) que executa parte da tarefa e divide o restante entre os demais processos (Escravos). Cada escravo executa sua tarefa e envia os resultados ao mestre, que envia uma nova tarefa para o escravo.

Pool de trabalho

Um conjunto de tarefas é depositado em uma área acessível aos processos componentes do programa paralelo. Cada processo retira uma parte de uma tarefa e executa. Esta fase se repete até que o conjunto de tarefas seja executado.

Fases paralelas

O programa paralelo é formado por fases, sendo necessário que todos os processos terminem uma fase para passar a fase seguinte. Mecanismos de sincronização (ex. barreiras) são usados para que um processo espere pelos demais para passar à fase seguinte.

Com MPI é possível elaborar programas utilizando-se dos modelos acima apresentados. A seguir serão apresentados dois programas, um com o modelo Mestre/Escravo e outro um Pipeline.

2.4.3.1. Programa mestre/escravo

No programa a seguir um processo (o mestre) obtém valores, envia para os escravos calcularem a fatorial, recebe os cálculos e os imprime.

```
include "mpi.h"
main(argc,argv)
int argc;
char **argv;
{
int numero, i, fat=1  ;
int myrank, size;
MPI_Status status;
MPI_Init (&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&size);

if (myrank==0){
      printf("Sou o processo 0 \n");
      for(i=1; i<4; i++){
            scanf("%d", &numero);
            printf("Numero: %d \n",numero);
            MPI_Send(&numero,1,MPI_INT,i,99,MPI_COMM_WORLD);
            }
      for(i=1; i<4; i++){
            MPI_Recv&numero,1,MPI_INT,MPI_ANY_SOURCE,99,MPI_COMM_WORLD,
            s&tatus);
            printf("resultado: %d \n",numero);
            }
      }
```

```
else
      {
      if (myrank==1)
            {
            printf("Eu sou o processo 1 \n");
            MPI_Recv(&numero,sizeof(int),MPI_INT,0,99,MPI_COMM_WORLD,
            &status);
            for(i=1; i<numero; i++)
                  fat = fat*i ;
            MPI_Send(&fat,1,MPI_INT,0,99,MPI_COMM_WORLD);
            }
      else {
            if (myrank==2)
            {
            printf("Eu sou o processo 2 \n");
            MPI_Recv(&numero,sizeof(int),MPI_INT,0,99,MPI_COMM_WORLD,
            &status);
            for(i=1; i<numero; i++)
                  fat = fat*i ;
            MPI_Send(&fat,1,MPI_INT,0,99,MPI_COMM_WORLD);
            }
            else{
             printf("Eu sou o processo 3 \n");
               MPI_Recv&numero,sizeof(int),MPI_INT,0,99,MPI_COMM_WORLD,
            &status);
                  for(i=1; i<numero; i++)
                        fat = fat*i ;
                  MPI_Send(&fat,1,MPI_INT,0,99,MPI_COMM_WORLD);
                  }
            }
      }
MPI_Finalize();
}
```

O programa acima é formado por quatro processos: o mestre e três escravos. O mestre faz a leitura de três valores e envia um para cada escravo, com a primitiva Send, a seguir fica em um laço esperando pelos cálculos dos escravos. Para o recebimento dos resultados, primitiva Recv, a identificação do remetente é feita com MPI_ANY_SOURCE, que permite o recebimento de mensagens de qualquer processo. Cada escravo recebe o valor do mestre (primitiva Recv), calcula a fatorial do número recebido e envia este resultado para o mestre.

2.4.3.2. Programa Pipeline

O programa a seguir apresenta o exemplo de um pipeline.

```
#include "mpi.h"
main(argc,argv)
int argc;
char **argv;
{
int numero;
int myrank, size;
MPI_Status status;
MPI_Init (&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

```
printf("Ola... \n");
if (myrank==0)
      {
      printf("Sou o processo 0 \n");
      scanf("%d", &numero);
      MPI_Send(&numero,1,MPI_INT,1,99,MPI_COMM_WORLD);
      MPI_Recv(&numero,1,MPI_INT,2,99,MPI_COMM_WORLD,&status);
      printf("Sou processo 0,recebi do processo 2 o valor%d \n",
      numero);
      }
else
      {
      if (myrank==1)
            {
            printf("Eu sou o processo 1 \n");
            MPI_Recv(&numero,sizeof(int),MPI_INT,0,99,MPI_COMM_WORLD,
            &status);
            numero = numero + 10 ;
            MPI_Send(&numero,1,MPI_INT,2,99,MPI_COMM_WORLD);
            }
      else
            {
            printf("Eu sou o processo 2 \n");

            MPI_Recv(&numero,sizeof(int),MPI_INT,1,99,MPI_COMM_WORLD,
            &status);
            numero = numero + 10 ;
            MPI_Send(&numero,1,MPI_INT,0,99,MPI_COMM_WORLD);
            }
      }
MPI_Finalize();
}
```

No programa acima são criados três processos. O processo 0 obtém um valor e o envia para o processo 1. O processo 1 adiciona um outro valor ao recebido e envia ao processo 2, que adiciona um valor ao recebido e envia para o processo seguinte no pipeline, que no caso é o processo inicial (0).

2.4.4. Compilação e execução de programas

A compilação de programas escritos na linguagem C (C++) é feita com o comando

mpicc [fonte.c] -o [executável] [parâmetros]

onde o comando mpicc aceita todos os argumentos de compilação do compilador C. A execução é feita com o comando

mpirun -[argumentos] [executável]

Os argumentos são:

h	- Mostra todas as opções disponíveis
arch	- Especifica a arquitetura da(s) máquina(s)
machine	- Especifica a(s) máquina(s)
machinefile	- Especifica o arquivo que contém o nome das máquinas

np - Especifica o número de processadores
leave_pg - Registra onde os processos estão sendo executados
nolocal - Não executa na máquina local
t - Testa sem executar o programa, apenas imprime o que será executado
dbx - Inicializa o primeiro processo sobre o dbx

Exemplos:
mpirun -np 5 teste
Executa o programa teste em 5 processadores
mpirun -arch sun4 -np 5 teste
Executa o programa teste em 5 processadores de arquitetura sun4.

2.4.5. MPI-2

Desde 1995 o Forum MPI começou a considerar correções e extensões ao MPI padrão. Em julho de 1997 foi publicado um documento que, além de descrever MPI 1.2 e apresentar seus padrões, define MPI-2. MPI-2 adiciona novas funcionalidades à MPI, permitindo uma extensão aos modelos de computação. Estas novas funcionalidades permitem:

- **Criação e gerência de processos:** são definidas primitivas que permitem a criação dinâmica de processos.
- **Comunicação com um único participante:** são definidas operações de comunicação que são realizadas por um único processo**.** Estas incluem operações de memória compartilhada (get/put) e operações remotas de acumulação.
- **Extensões às operações coletivas:** novos métodos de construir intercomunicators e novas operações coletivas.
- **Interfaces Externas:** é definido um novo nível, acima de MPI, para tornar mais transparente objetos MPI.
- **I/O Paralelo:** são definidos mecanismos de suporte para I/O paralelo no MPI.
- **Linguagens**: são definidas novas características de ligação de MPI com C++ e com Fortran-90.

Um estudo destas novas funcionalidades foge ao escopo deste trabalho. O leitor interessado encontra a descrição completa de MPI-2 em www.mpi-forum.org.

2.4.6. Conclusão

O estudo de MPI apresentado tem como objetivo iniciar o leitor no desenvolvimento de programas paralelos com o uso desta biblioteca de comunicação. Foram apresentados conceitos básicos utilizados em MPI, as principais funções utilizadas para troca de mensagens e para inicializar e finalizar o sistema. Foram apresentados exemplos de programas em dois modelos de programação (mestre/escravo e pipeline) de maneira a permitir uma fácil compreensão. Espera-se que, a partir deste documento, o leitor esteja apto a começar a escrever suas aplicações, utilizando como base os programas apresentados. Para um estudo mais aprofundado desta ferramenta de programação, uma descrição completa pode ser encontrada nas referências relacionadas, especialmente nos documentos do Fórum MPI.

2.5. Bibliografia

[BLA90] BLACK, D. L. **Scheduling support for concurrency and parallelism in the Mach operating system**. Computer. V. 5(23). May, 1990.

[CAV01] CAVALHEIRO, G. G. H. **Introdução à programação paralela e distribuída**. In: Anais *I* Escola Regional de Alto Desempenho. Gramado. 2001.

[FOS95] FOSTER, YAN. Designing and Building Parallel Programs. Addison Wesley, 1995.

[GEI94] GEIST, Al et al. **PVM**: **Parallel and Virtual Machine- A User s Guide and Tutorial for Networked Parallel Computing**. London: MIT, 1994

[MPI94] MPI-10 disponível em http://www.mpi-forum.org , May 1994.

[MPI97] MPI-2 disponível em http://www.mpi-forum.org , May 1994.

[OLI01] OLIVEIRA, R. S de; CARISSIMI, A. S e TOSCANI, S. S. **Sistemas Operacionais**. Porto Aleger: Sagra-Luzzatto. 2001

[PAC97] PACHECO, PETER. Paralllel Programming with MPI. Morgan Kaufmann Publischers, 1997.

[POW91] POWELL, M. L. et al. **SunOS multi-thread architecture**. In. **Proc.** of the Winter 1991 USENIX Technical Conference and Exhibition. Springer Verlag, LNCS 980. 1991.

[PVM 01] PVM Home Page disponível em http://www.epm.ornl.gov/pvm/pvm_home.html , Nov. 2001

[SEB00] SEBESTA, R. W. **Conceitos de linguagens de programação**. Porto Alegre: Bookman. 2000.

[SEI01] SEIXAS, Roberto de Beauclair. **Parallel Virtual Machine**. Apresentação disponível em http://www.impa.br/~tron/pdf/pvm.pdf , Nov. 2001

[VAH76] VAHALIA, U. **UNIX Internals**. Englewood Cliffs: Prentice Hall. 1976

[WIL99] WILKINSON, B and ALLEN, M. **Parallel programming**: techniques and applications using networked workstations and parallel computers. Upper Saddle River: Prentice-Hall. 1999.