Introdução à Programação Paralela e Distribuída

Prof. Gerson Geraldo Homrich Cavalheiro

Universidade do Vale do Rio dos Sinos
PIP/CA - Centro 6
Av. Unisinos 950 93022-000 São Leopoldo - RS - Brasil
E-mail: gersonc@exatas.unisinos.br
http://www.inf.unisinos.br/~gersonc

1 Introdução

Dispondo de uma arquitetura multiprocessadora, o principal problema que se apresenta é de como explorá-la eficientemente através do uso da programação concorrente. Esta forma de programação, também chamada programação paralela ou distribuída – esta última denominação empregada quando a arquitetura não dispõem de uma área de memória comum –, possibilita que aplicações sejam descritas por programas capazes de serem executados por diversos fluxos de execução independentes. A programação concorrente explora a possibilidade de descrever a aplicação através de um conjunto de tarefas, algumas das quais podendo executar simultaneamente, outras necessitando colaborar entre si para a troca de informações.
O objetivo deste curso é apresentar uma introdução ao paradigma de programação concorrente. A seguir é apresentada uma definição para dois dos principais elementos que compõem um programa concorrente: tarefa e sincronização. Na seqüência, é apresentada uma discussão sobre os diferentes níveis de concorrência que podem ser explorados em um programa aplicativo e no restante deste capítulo ferramentas para programação concorrente.

2 A Programação Concorrente

O termo concorrência é comumente aplicado para definir sistemas onde é possível compartilhar o uso dos recursos de processamento de um computador – tipicamente o processador – para suportar o fluxo de execução de dois ou mais programas independentes [22] [20]. Nestes casos, a concorrência visa obter ganho de desempenho no sistema e/ou prover um ambiente *time-sharing*. Um ambiente *time-sharing* fornece um mecanismo de compartilhamento dos recursos de processamento de um computador entre processos submetidos por diferentes usuários, desta forma, cada usuário tem o sentimento de que possui um computador próprio para execução de seu programa. Já o uso da programação concorrente para ganho de desempenho, vem de uma constatação de que enquanto um programa realiza operações de E/S, o processador permanece inativo. Considerando que os programas são independentes entre

si, a concorrência é empregada para não desperdiçar o tempo de uso da unidade de processamento: assim, quando um programa necessita realizar uma operação de E/S, um novo programa é selecionado para ocupar o processador e avançar na execução de suas instruções.

Seguindo o modelo de von Neumann, a arquitetura dos computadores consiste em uma máquina de cálculo que possui um processador, uma área de memória para dados e código executável e de dispositivos de E/S. Neste modelo, as instruções de um programa são executadas seqüencialmente, uma à uma, na mesma ordem em que se encontram no código programa fonte. Programas em execução concorrente, neste caso, não possuem nenhuma restrição temporal de execução e competem pelo tempo de acesso ao processador, dividindo entre si o espaço de memória disponível e o acesso aos dispositivos de E/S.

Para garantir a correta execução dos programas em ambientes que exploram a concorrência a este nível, o aspecto mais importante é o de prover mecanismos que impossibilitem que a execução de um programa interfira na execução de outro. De uma forma mais clara, não permitir que uma instrução de um programa venha alterar alguma posição de memória pertencente a outro programa (proteção de memória) e garantir que um programa correto tenha respeitada a ordem de execução de suas instruções e que, após ter sido iniciado, tem sua execução completada em um tempo finito (proteção do processador).

O princípio destas considerações é de que os programas são inteiramente dissociados. Cada programa executa uma tarefa específica, não havendo, entre dois programas, qualquer forma de colaboração dinâmica. Portanto, entre os fluxos de execução de cada programa não existe nenhuma dependência temporal, visto que não há troca de informações entre eles. Em outras palavras, o conjunto de fluxos em execução não reflete uma aplicação, e sim aplicações autônomas processando atividades não relacionadas entre si.

Assim temos a primeira definição para concorrência: um conjunto de fluxos de execução independentes que disputam o uso dos recursos de arquitetura para terem suas instruções executadas. Esta disputa reflete o compartilhamento do tempo de uso do processador e de espaço de armazenamento na memória. A abordagem adotada pela programação concorrente [25] possui uma diferença importante: neste caso, busca-se a implementação de uma única aplicação, utilizando-se do recurso de decompo-la em diversos fluxos de execução. Cada um destes fluxos é responsável pela execução de uma parte da solução do problema, o que implica que eles não sejam verdadeiramente independentes. Sendo cada fluxo de execução responsável por uma parte da solução do problema, de alguma forma é necessário que resultados obtidos por um destes fluxo seja comunicado a outro de forma à compor a solução final, ou seja, a solução a ser apresentada pela aplicação. Em [19], este nível de concorrência é identificado como concorrência a nível de unidade.

Desta forma, programar de forma concorrente implica detectar na aplicação as atividades que não possuem restrições temporais e os pontos onde restrições temporais existem. Ou seja, para cada conjunto de instruções de um programa, definir quais produzem resultados necessários ao início da execução de outro conjunto e quais conjuntos são independentes entre si.

O termo concorrência passa assim a agregar um novo significado, aplicando-se a fluxos de execução onde determinados trechos de instruções não possuem restrições temporais de execução – independentemente do fato de compartilharem ou competirem por recursos de execução –, intercalados por trocas de dados entre estes fluxos. Esta forma de programação permite que diferentes fluxos de execução colaborem entre si, empregando mecanismos de sincronização [19], para atingir um objetivo comum: o resultado esperado.

Um programa concorrente é, desta forma, composto por um conjunto de fluxos de execução que, de alguma forma ordenada, trocam informações. Os fluxos de execução consistem no suporte à execução das tarefas definidas na aplicação e as sincronizações oferecem os mecanismos sobre os quais são implementadas as trocas de dados entre estas tarefas. Um programa concorrente será corretamente executado se todas tarefas definidas forem executadas respeitando as sincronizações definidas entre elas.

2.1 Definição de tarefa e sincronização

Uma tarefa, em um programa concorrente, consiste em um conjunto de instruções executadas de forma seqüencial sob um fluxo de execução. Uma tarefa produz um conjunto de dados de saída, e define um conjunto de parâmetros como seus dados de entrada. Assim como os dados de saída de uma tarefa podem ser parâmetros de entrada de outra (ou outras), os dados de entrada de uma tarefa são dados de saída de uma outra tarefa (ou de um conjunto de tarefas). Durante a execução de um programa, duas tarefas são consideradas independentes quando a saída de uma não for a entrada de outra, caso contrário, existe uma comunicação: as duas tarefas devem então ser executadas de forma síncrona, garantindo a correta comunicação entre elas.

A sincronização permite o controlar a evolução da execução de um programa controlando o acesso das tarefas aos dados gerados por outras tarefas. Este mecanismo implícito a coordenação das comunicações entre tarefas.

Tomando em consideração as dependências entre as tarefas, um programa concorrente pode ser representado através de um grafo dirigido, onde cada vértice representa uma tarefa e uma aresta dirigida, uma sincronização. O algoritmo de um programa concorrente e o grafo gerado pela sua execução são apresentados na figura 1. Uma análise deste grafo permite concluir verificar que as tarefas possui um ciclo de vida composto por quatro estados [13]:

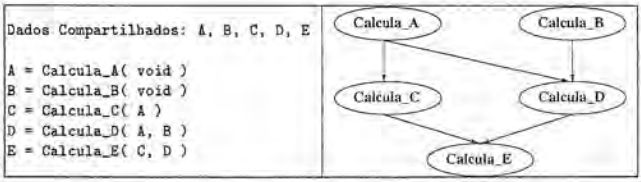


Figura 1: Grafo gerado pela execução de um algoritmo concorrente.

- **Aguardando:** a tarefa foi definida, porém os dados de entrada não encontram-se disponíveis. É o caso da tarefa `Calcula_C`, antes do término da execução da tarefa `Calcula_A`.
- **Pronta:** a tarefa pode ser executada, pois os dados que necessita para ser executada encontram-se disponíveis. Será o caso, no exemplo, da tarefa `Calcula_C` após o término da execução da tarefa `Calcula_A`.
- **Executando:** quando uma tarefa é atribuída a um fluxo de execução, tendo suas instruções executadas seqüencialmente.
- **Concluída:** uma tarefa é considerada concluída ao ter produzido seus dados de saída, liberando o fluxo de execução ao qual estava associada.

Os mecanismos de sincronização empregados para realizar as comunicações de dados permite controlar a evolução no ciclo de vida das tarefas. Observe no entanto que uma tarefa não é necessariamente lançada em execução no momento em que estiver num estado **Pronto**. Se esse for o caso, a tarefa aguarda em uma fila de tarefas prontas o momento em que será iniciada sua execução.

2.2 Diferentes níveis de concorrência

Na prática da programação de alto desempenho, o termo concorrência é aplicado a diferentes níveis na execução de uma aplicação e, de acordo com os recursos de hardware disponíveis, ela é expressa de uma forma diferente. Sendo tomada um agregado (*cluster*) como arquitetura de suporte a execução de uma aplicação, os níveis de concorrência que podem ser explorados são: a concorrência intra-nó e a concorrência entre nós, sendo a principal diferença a forma de interação entre os fluxos de execução para comunicação de resultados das atividades independentes. A interação entre os fluxos de execução é um aspecto de relevada importância na programação concorrente, pois ela caracteriza a forma pela qual diferentes atividades da aplicação irão colaborar para atingir o resultado final.

Concorrência intra-nó

A concorrência intra-nó é definida por um conjunto de atividades concorrentes executadas em um mesmo nó de um agregado (ou de um computador independente). A concorrência intra-nó pode ainda ser real ou temporal dependendo do número de fluxos de execução ativos em um determinado momento e do número de recursos de processamento - processadores - disponíveis no nó. Caso haja um número de processadores pelo menos igual ao de fluxos de execução ativos, existe a concorrência real, também denominado paralelismo. Caso contrário, um número de fluxos de execução ativos em um determinado instante de tempo maior que o número de processadores disponíveis, existe a concorrência clássica, onde há o compartilhamento de recursos.

A forma de interação mais natural, e menos onerosa, para o compartilhamento de informações nesta forma de concorrência é através do espaço de armazenamento provido pela memória do próprio nó. O mecanismo de comunicação explora o fato de que os dados manipulados pelas instruções contidas em diferentes fluxos de execução são variáveis armazenadas em memória. O acesso a estas variáveis se dá através de triviais operações de leitura e escrita (tipo *load* e *store*). Sendo a memória um recurso pertencente ao nó e compartilhado pelos fluxos em execução, um dado escrito por uma instrução em um fluxo de execução pode ser lido por uma instrução em outro fluxo.

Mesmo que a comunicação entre fluxos de execução se apoie em instruções de leitura e escrita, portanto idênticas às utilizadas em uma execução seqüencial, um cuidado adicional deve ser tomado na sincronização ao acesso a dados compartilhados. Em uma execução seqüencial, o resultado de uma instrução é comunicado a outra instrução através de uma escrita em memória, que reflete o efeito colateral da execução da instrução: a própria alteração do estado do dado na memória, que no futuro servira de entrada para uma outra instrução. A comunicação entre duas instruções se dá através do compartilhamento de certas posições de memória e a sincronização entre duas instruções é realizado automaticamente, executado uma instrução somente a partir do momento em que a instrução que lhe antecede foi completada (aqui cabe salientar que outros níveis de concorrência, como o provido por arquiteturas super-escalares, não estão sendo considerados). Com isto garante-se que as alterações de estado da memória foram realizados e que todo dado lido pela instrução em execução está atualizado.

Em uma execução concorrente, o sincronismo implícito é inexistente e toda instrução em um fluxo de execução que acesse um dado compartilhado com outro fluxo consiste em uma instrução crítica. Mesmo que uma instrução crítica possa acessar um dado compartilhado da mesma forma que uma instrução executada em um fluxo de execução de execução seqüencial, um mecanismo externo deve prover a sincronização entre a instrução que produz o valor para o dado compartilhado e a instrução que necessita deste dado como entrada para sua própria execução. O

correto emprego da sincronização é a única garantia de que a comunicação entre as tarefas vai ser realizada como esperado.
Dentre os diversos mecanismos de sincronização, os mais utilizados são os que garantem exclusão mutua no acesso a memória (mutexes) e controle no avanço de execução (como semáforos e *joins*).
As ferramentas que possibilitam a exploração da concorrência intra-nós mais clássicas baseiam-se no padrão POSIX [1] para processos leves (ou *threads*). *Threads* POSIX são disponíveis em diferentes sistemas, tais como Linux, AIX, Minix, Solaris e mesmo na família Windows.

Concorrência entre nós

Em um agregado, cada nó possui seu próprio conjunto de recursos de processamento. Diferentes fluxos de execução suportado por diferentes nós não competem por processador ou memória, assim sendo, tarefas que não necessitem ser sincronizadas podem ser executadas ao mesmo tempo, uma em cada nó, explorando o paralelismo real da arquitetura. Eventuais sincronizações e trocas de dados entre as partes ainda se faz necessário, mas como a memória da arquitetura do agregado não prevê uma área de endereçamento comum a todos os nós, o único recurso é utilizar o meio físico previsto pela interconexão entre os nós. Costuma-se denominar *aplicação distribuída* uma aplicação que executa sobre uma arquitetura não dotada de memória comum que explora a rede de interconexão entre os nós para permitir a cooperação entre as tarefas. O termo *distribuída* reflete a configuração da memória, distribuída entre os nós.
Utilizar uma rede para prover a comunicação entre fluxos de execução implica no uso de primitivas do tipo *Envia* e *Recebe* (*Send* e *Receive*) ou suas variantes (algumas das quais detalhadas nas próximas seções). Essa forma de interação é consideravelmente mais complexa (e mais onerosa em tempo de processamento) que o mecanismo empregado quando todos processadores dispõem do acesso à uma memória comum.
A utilização das primitivas de *Envia* e *Recebe* permite a colaboração e a sincronização entre as tarefas definidas pela aplicação através de troca de mensagens. O princípio desta colaboração é empregar uma primitiva *Recebe* para receber e *Envia* para enviar um dado. Durante a execução de um programa, uma chamada a uma destas duas primitivas pode ser interpretado como o fim de uma tarefa e início de uma nova. No caso de uma chamada a uma primitiva *Envia*, a tarefa *enviadora* é considerada terminada e o resultado produzido enviado a tarefa *receptora*. Na chamada a uma primitiva *Recebe*, a tarefa em execução é considerada terminada e a seqüência de instruções após o *Recebe* define a próxima tarefa, a qual estará **Pronta** para ser executada após o recebimento do respectivo dado.
Apesar de produzir algoritmos com lógicas mais complexas, o uso do compartilhamento de dados via troca de mensagens é bastante popular e são encontradas na bibliografia diversas ferramentas disponibilizando estes recursos de programação.

Entre estas ferramentas, podemos citar PVM (*Portable Virtual Machine*) e MPI (*Message Passing Interface*), ambas permitem a criação de uma máquina virtual, composta de nós virtuais, cada nó executando um único fluxo de execução (uma componente da aplicação).

Concorrência intra e entre nós
Para uma completa exploração do paralelismo em um agregado, é necessário empregar tanto a concorrência intra-nó como a entre nós. Porém o casamento entre estes dois mecanismos não é uma tarefa simples [7]. O ganho que pode ser obtido na exploração desses dois níveis de concorrência é o recobrimento de parte do tempo gasto em comunicações pela execução de cálculo útil [24], princípio equivalente ao utilizado para compartilhar o uso do processador quando um processo realiza uma operação de E/S.
Diversas ferramentas propõem uma solução disponibilizando os recursos de exploração de ambos níveis de concorrência, como por exemplo as bibliotecas Athapascan-0 [7] e Nexus [12] e a linguagem de programação Java [10].

3 Processos Comunicantes

Um grande número das atuais arquiteturas paralelas de computadores é composta por com um conjunto de nós, cada um destes dispondo de um (ou um grupo de) processador(es) e um módulo de memória privado, estando os nós interconectados através de uma rede de comunicação de alta velocidade [2]: são as ditas arquiteturas com memória distribuída, tais os agregados de computadores. Neste tipo de arquitetura, a malha de interconexão consiste no único recurso que pode ser explorado para a cooperação entre os processadores: placas de rede conectadas a cada nó são responsáveis por (*i*) ao comando de um nó, enviar um conjunto de dados que se encontram em uma determinada região da memória através da rede e (*ii*) receber dados da rede e disponibiliza-los em alguma região da memória para futuramente serem recuperados por algum programa em execução.
A nível de software esta estrutura reaparece ao se tratar de processos: um processo, em um sistema de computação, consiste em uma unidade de execução autônoma que possui uma área de memória própria para armazenamento de dados e um conjunto de instruções. Este conjunto de instruções define o serviço a ser realizado, e é executado sobre um fluxo de execução próprio ao processo[1] acessando unicamente os dados que possui na sua memória privada. Dois ou mais processos podem vir a cooperar empregando mecanismos explícitos de troca de mensagens, primitivas do tipo *Envia* e *Recebe*, para envio dos dados, independentemente de estarem ou não sendo executados sobre um mesmo nó físico.

[1] Na sessão 4será mostrado que, na realidade, um processo pode possuir vários fluxos de execução, cada um suportando a execução de diferentes seqüências de instruções.

Se a nível de hardware o problema da troca de mensagens é de compreensão relativamente simples, a nível de software estão envolvidas questões relativas a sincronização e ao compartilhamento de dados entre as diferentes partes da aplicação. Uma tarefa, em tal estrutura de aplicação paralela, pode ser definida por uma seqüência de instruções entre duas primitivas de comunicação, as quais permitem a sincronização entre tarefas. Assim, os dados recebidos através de uma primitiva *Recebe* consistem nos parâmetros de entrada da tarefa (uma vez os dados recebidos a tarefa esta pronta para ser executada) e os dados enviados consistem nos parâmetros de saída da tarefa,

Uma aplicação para construir uma estrutura de processos que executem em computadores de uma rede, colaborando entre si através de mensagens, deve dispor de recursos para criação remota de processos e de comunicação [25]. Estes recursos são discutidos nesta sessão, fazendo uso de ferramentas que seguem o padrão MPI [15] - *Message Passing Interface* -. Os exemplos apresentados foram desenvolvidos sobre Linux e os serviços MPI oferecidos pela biblioteca LAM - *Local Area Machine* -.

3.1 Máquina virtual

A solução proposta por MPI para a criação remota de processos é a construção de uma *máquina virtual*, composta de *nós virtuais*, sobre uma arquitetura real. Cada nó virtual consiste em um processo, responsável por executar a aplicação. Segundo o padrão MPI, a configuração da máquina virtual, número de nós, permanece inalterada durante toda a execução da aplicação[2]. A princípio, cada nó real suporta a execução de um nó virtual, porém nós reais são autorizados a suportar a execução de mais de um nó virtual. Esta característica é útil pois permite codificar a aplicação sem que o número de nós da máquina real seja conhecido.

Na figura 2 é apresentada uma configuração de uma máquina virtual contando com 6 nós; cada nó virtual representado por um retângulo com bordas arredondadas. As formas quadradas na figura representam os 4 nós da máquina real sobre a qual a máquina virtual esta sendo executada. Nos nós virtuais estão representados o fluxo de execução suportado e sua área de memória privada.

O processo que executa o nó virtual consiste em uma unidade de execução independente, podendo receber/enviar mensagens de/a outro nó virtual.

Programação SPMD

Antes de iniciar o estudo do MPI propriamente dito, é necessário conhecer o modelo de programação empregado, neste caso, o modelo SPMD - *Single Program, Multiple Data* -. Neste modelo, a aplicação define um conjunto de processos que deverão executar o mesmo código de forma concorrente. Note que, devido ao fato de diferentes

[2] Em uma nova edição do padrão MPI, o MPI-2, é possível variar a configuração da máquina virtual dinamicamente.

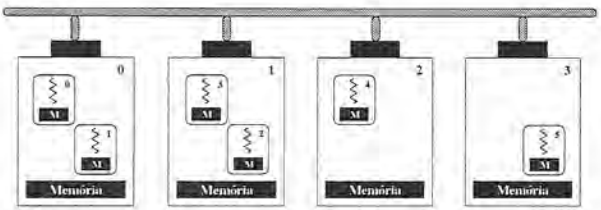


Figura 2: Máquina virtual de 6 nós sobre um agregado de 4 nós.

processos manipularem diferentes conjuntos de dados, a porção de código sendo executada por um processo não é necessariamente a mesma que a executada por um outro processo.

```
main() {
 int id;

 código comum
 id = fork();
 if( id != 0 ) { código pai }
 else { código filho }
}
```

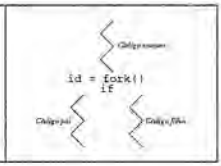


Figura 3: Uso de primitiva `fork` em ambientes Unix.

Este modelo lembra a utilização de primitivas do tipo `fork` em Unix (exemplo na figura 3 para o código e fluxos de execução gerados). A chamada de uma primitiva *fork* por um processo força sua duplicação em dois processos independentes: um processo *pai* e um processo *filho*. Apesar de cada processo conter uma cópia completa do código da aplicação, instruções de controle de fluxo, como o `if` no exemplo, decidem qual o trecho a ser executado em cada um.

Na programação SPMD, não apenas dois, mas um grupo de n processos executam um mesmo programa. De forma semelhante ao `id` no `fork`, cada processo tem acesso a identificação de sua posição no grupo, ou seja, saber que ele é o i-ésimo nó de uma máquina virtual de n nós; em função da posição de seu nó, o processo pode selecionar a porção do código a ser executado.

Durante a execução dos processos, não existe nenhuma forma de sincronização implícita entre os processos; a introdução de pontos de sincronização entre é de responsabilidade da aplicação: o programador deve, explicitamente, utilizar mecanismos de troca de mensagens para possibilitar a cooperação entre as tarefas.

Outra diferença fundamental entre o `fork` e a programação SPMD é que, no momento da execução do serviço `fork` o processo filho consiste em uma cópia do processo original, implicando que a área de dados seja igualmente duplicada, enquanto processos SPMD consistem em instâncias totalmente autônomas desde o momento em que a execução é iniciada. No caso do `fork`, o processo *filho* executa-se a partir da instrução seguinte à chamada ao `fork` no *mesmo nó* do processo original e contém na sua área de dados uma imagem do estado da memória do processo pai. Os processos na programação SPMD são iniciados todos a partir do mesmo ponto, o início do programa, cada um responsável por inicializar sua própria área de dados.

Processo de inicialização

Utilizando LAM, a manipulação da arquitetura real, sobre a qual a aplicação será executada, é realizada através de três aplicativos: `lamboot`, `wipe` e `lamclean`. Observe que a variável de ambiente `LAMHOME` deve estar definida com o diretório em que a biblioteca LAM encontra-se instalada (por exemplo `/usr/local/lam-6.4`).
A primeira preocupação que o usuário deve ter antes de iniciar a execução de seu programa é de definir sobre quais nós reais da arquitetura real a máquina virtual deverá ser construída. Isto é feito da seguinte forma:

```
lamboot <hosts>
```

onde `host` é um nome de um arquivo contendo a identificação dos nós reais que serão utilizados.
Por exemplo, para utilizar quatro nós de um agregado composto por 16 nós, (`clu0`, `clu1`, ..., `clu15`) o usuário deve criar um arquivo contendo a identificação dos nós que deseja utilizar, identificando um nó por linha do arquivo. Esta situação é apresentada abaixo (o parâmetro `-v` para *verbose*, permitindo visualizar informações sobre a execução do comando):

```
                    $ lamboot -v quatronos

$ more quatronos    LAM 6.4/IMPI/MPI 2 C++ - University of Notre Dame
clu0
clu2                Executing hboot n0 (clu0)...
clu3                Executing hboot n0 (clu2)...
clu5                Executing hboot n0 (clu3)...
$                   Executing hboot n0 (clu5)...
                    topology done
                    $
```

Neste momento, em cada um dos nós que configura a máquina real, é criado um processo *daemon* (o processo `lamd`) que alocada uma porta para comunicação, a identificação desta porta é enviada para o `lamboot` que se encarrega de distribuir todos pares porta/nó a todos *daemons*, construindo desta forma uma topologia de comunicação inteiramente interconectada.

A operação inversa, *destruir* a configuração LAM montada, é realizada através do aplicativo `wipe`. A execução de um `wipe` implica em um envio de um sinal de término para todos os *daemons* que estejam ativos. Caso alguma aplicação esteja sendo executada, esta será abortada. O mesmo arquivo contendo a identificação dos nós usados pelo `lamboot` é necessário no `wipe`; abaixo um exemplo de uso:

```
$ wipe -v quatronos
LAM 6.4/IMPI/MPI 2 C++ - University of Notre Dame

Executing tkill n0 (clu0)...
Executing tkill n0 (clu2)...
Executing tkill n0 (clu3)...
Executing tkill n0 (clu5)...
$
```

Tanto no caso do `lamboot` como do `wipe`, a omissão do arquivo de máquinas faz com que apenas a máquina que executou o comando seja ativada.

Caso o usuário deseje apenas *derrubar* a aplicação que esteja executando, mas não destruir a configuração LAM montada, ele pode fazer uso do `lamclean`. Ao contrário do `wipe`, `lamclean` desmonta a máquina virtual, enviando um sinal de término apenas aos processos que suportam a execução dos nós virtuais, mantendo os *daemons* LAM ativos. `lamclean` é útil quando deseja-se fazer uma série de execuções sobre uma mesma configuração ou durante a fase de teste de um programa. O `lamclean` não necessita do arquivo descrevendo a máquina real utilizada.

A criação da máquina virtual propriamente dita, ou seja, dos processos que vão suportar a execução dos nós virtuais, se dá no momento em que a aplicação é lançada. Este lançamento se dá através do uso do aplicativo `mpirun`. `mpirun` pode receber diversos parâmetros, sendo principal deles o nome do programa a ser executado; outro parâmetro é o número de nós que deve ter a máquina virtual. Com estes dois parâmetros, o uso de `mpirun` possui a seguinte sintaxe:

```
mpirun -c <n> <prog>
```

Este comando faz com que seja criada uma arquitetura virtual de `n` nós. Cada nó consistindo em um processo lançado sobre um dos nós da máquina real suportando a execução de uma cópia do programa de aplicação. No exemplo abaixo, é aprsentada uma sessão completa utilizando LAM, criando uma máquina virtual de seis nós executando o programa `prog1`. O esquema dos processos em execução já foi visto na figura 2.

```
$ lamboot quatronos

LAM 6.4/IMPI/MPI 2 C++ - University of Notre Dame

$ mpirun -c 6 prog1
   execução do programa
$ wipe quatronos
```

Na seqüência são apresentadas algumas das principais primitivas MPI que permitem a colaboração entre os processos.

3.2 Primeiro programa

Nesta sessão é apresentado um primeiro exemplo de programa em MPI. Este programa, apresentado abaixo, faz com que o nó virtual 0 imprima uma mensagem informando o número de nós da máquina virtual.

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

int main( int argc, char **argv ) {
 int myrank, size;

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

 if( myrank == 0 ) printf( "Foram criados %d nos\n", size );

 MPI_Finalize();
}
```

Observando que todas primitivas MPI iniciam pelo prefixo `MPI_`, identificamos no exemplo acima a chamada a quatro serviços MPI. A primeira invocação, realizada através de `MPI_Init`, permite inicializar o nó virtual, os parâmetros passados são o `argc` e o `argv` recebidos pelo programa principal. Obrigatoriamente este é o primeiro serviço MPI a ser invocado.
A seguir, as primitivas `MPI_Comm_size` e `MPI_Comm_rank` permite que o processo saiba quantos nós existem na máquina virtual e qual sua posição neste grupo. No programa apresentado, estas informações são utilizadas para imprimir uma simples mensagem; no caso do nó número 0, é impressa também uma mensagem informando o número total de nós virtuais criados. O primeiro parâmetro destas duas primitifas identificam o grupo de nós virtuais que deseja-se manipular, no caso `MPI_COMM_WORLD`, todos os nós da máquina virtual.
Finalmente, o uso de MPI é encerrado através de uma invocação à `MPI_Finalize`. Observe que, ao executar uma invocação a `MPI_Finalize`, o processo é bloqueado e permanece neste estado enquanto aguarda que todos os outros processos executem este serviço, sendo então a máquina virtual destruída.

3.3 Compartilhamento de dados

MPI oferece diversas primitivas de comunicação, as quais permitem a troca de mensagens entre dois (ou mais) processos. Uma mensagem em MPI pode ser representada como no esquema da figura 4 onde é possível observar que ela é composta por quatro

campos: a identificação da **origem** e do **destino** da mensagem, os **dados** transmitidos e um **tag**. Este tag permite que as mensagens sejam rotuladas, viabilizando sua filtragem na recepção.

Figura 4: Diferentes componentes de uma mensagem MPI.

Dentre as primitivas de comunicações disponibilizadas por MPI, as mais básicas são `MPI_Send` e `MPI_Recv`, respectivamente para envio e recebimento de mensagens. Note que mensagens recebidas em um nó virtual são armazenadas em uma fila enquanto aguardam a operação de recebimento correspondente; nesta fila, as mensagens são armazenadas na ordem em que foram recebidas, não sendo garantida a ordem temporal de envio das mensagens provenientes de diferentes nós. Abaixo a sintaxe destas duas primitivas. Observe que, como na maioria das funções MPI, `MPI_Send` e `MPI_Recv` retornam um valor inteiro indicando um código de erro decorrente da execução do serviço, sendo 0 (zero) uma execução com sucesso.

```
int MPI_Send( void *buff, int cont, MPI_Datatype tipo,
              int dest, int tag, MPI_Comm grupo );
int MPI_Recv( void *buff, int cont, MPI_Datatype tipo,
              int origem, int tag, MPI_Comm grupo, MPI_Status *status );
```

onde:

- `buff` corresponde à área de dados a ser transmitida ou onde os dados recebidos devem ser armazenados. A gerência desta área de memória, alocação e liberação, é responsabilidade do programador.
- `tipo` descreve o tipo do dado que a mensagem contém. Alguns tipos primitivos são definidos por MPI, conforme apresentado na tabela 1. Outros tipos podem ser introduzidos pelo usuário, maiores informações em [15].
- `cont` o número de elementos a serem enviados ou recebidos, sendo cada elemento do tipo `tipo`. Em outras palavras, `buff` é um *array* contendo `cont` elementos do tipo `tipo`.
- `origem` e `dest` identificam, respectivamente o nó origem e destino da mensagem. Na recepção, o parâmetro `origem` permite que as mensagens presentes na fila de recepção sejam filtradas, sendo *recebida* procurada a mensagem mais antiga na fila originada de um determinado nó; caso seja informado o valor `MPI_ANY_SOURCE` para este parâmetro, não ocorrerá este filtro, sendo recebida a mensagem mais antiga na fila.
- `tag` aplica um novo nível de filtro, possibilitando a classificação de mensagens através de um rótulo: somente uma mensagem que possua o *tag* informado será recebida. Caso não haja necessidade de filtro a este nível, a este parâmetro deve

ser passado o valor `MPI_ANY_TAG`. Este filtro pode ser utilizado, por exemplo, para atribuir prioridades de tratamento: o programador pode definir que o `tag` 0 indica uma mensagem a ser tratada com alta prioridade, `tag` 1 mensagem com média prioridade e `tag` 2 mensagens com baixa prioridade. Um nó, ao tratar mensagens recebidas, filtra em primeiro lugar todas que possuem `tag` 0, em seguida as que possuem `tag` e relega a mais baixa prioridade de recepção as mensagens cujo `tag` seja 2. Observe que a semântica dada ao `tag` é exclusiva da aplicação.
- `grupo` informa o grupo a que pertencem os nós `origem` e `destino`, tipicamente `MPI_COMM_WORLD` para todos os nós da máquina virtual.
- `status` permite que o receptor tenha acesso a uma série de informações a respeito da mensagem recebida (por exemplo, tamanho em bytes).

Tabela 1: Principais de dados MPI predefinidos.

Tipo de dado MPI	Correspondente em C
`MPI_CHAR`	`char`
`MPI_INT`	`int`
`MPI_LONG`	`long`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_PACKED`	tipo a ser informado

O par de primitivas `MPI_Send` e `MPI_Recv` permite realizar comunicações síncronas entre processos. Isto quer dizer que um processo ao invocar uma primitiva `MPI_Send` fica bloqueado até que a comunicação seja concluída. De forma semelhante, ao invocar `MPI_Recv`, o processo fica bloqueado até que a mensagem desejada esteja presente na fila de mensagens.
Caso o sincronismo a este nível (bloqueamento do processo) não seja desejado, é possível optar por primitivas de comunicação assíncronas (não bloquantes), tipo `MPI_Isend` e `MPI_Irecv`, cujos parâmetros são os mesmos das sua homólogas síncronas. No caso de uma operação de envio assíncrono, a mensagem é postada na rede e o processo é desbloqueado, podendo continuar suas operações. Na recepção assíncrona, caso a mensagem já tenha sido recebida, ela é lida para o *buffer* de recepção e o processo pode continuar suas operações; caso a mensagem ainda não tenha sido recebida, o processo é também liberado para continuar sua execução, devendo em um momento posterior tentar uma nova leitura.

Comunicação de grupo

Outra possibilidade de comunicação em MPI é a comunicação de grupo, onde um processo pode enviar, ou receber, mensagens a, ou de, todos outros processos através

de mecanismos de *broadcast* e redução. Estas formas de comunicação coletiva são realizadas através das primitivas `MPI_Bcast` e `MPI_Reduce`. Observe-se que estas rotinas não aceitam tags para filtrar mensagens e que a comunicação envolve necessariamente todo o grupo de processos envolvidos (`MPI_Comm == MPI_COMM_WORLD`). No entanto, um destes processos é assumido como raiz da comunicação; quando da ocorrência de um *broadcast* o processo raiz é resisponsável pelo envio da mensagem, e no caso de uma redução, pelo recebimento e tratamento das mensagens.

Abaixo é apresentada a sintaxe para o *broadcast* em MPI. Existe apenas uma primitiva para realização do *broadcast* e esta deve ser realizada por todos os processos participantes, tanto pelo processo raiz, que envia a mensagem, como pelos processos que a receberão.

```
int MPI_Bcast( void *buff, int cont, MPI_Datatype tipo,
               int raiz, MPI_Comm grupo );
```

Ao utilizar `MPI_Bcast`, a semântica associada ao parâmetro `buff` é obtida pelo valor fornecido ao parâmetro `raiz`. Caso o valor informado para `raiz` corresponda a própria posição do processo no `grupo` (seu *rank*), `buff` contém os dados a serem enviados: este processo é considerado a raiz da comunicação. Nos demais processos, `buff` corresponde a área de dados onde deve ser armazenado os dados recebidos.

O trecho de código abaixo apresenta o esquema de um emprego prático de `MPI_Bcast`: o cálculo da integral de uma função entre os pontos x_i e x_f. Neste exemplo, o nó 0 é o raiz da comunicação e a máquina virtual deve contar com, ao menos 2 nós; todos os nós, exceto o nó 0, são considerados servidores de cálculo. A função do nó 0 é, primeiro, obter o intervalo (x_i, x_f) (do teclado, de um arquivo, etc.) e, em seguida, enviar este intervalo aos nós servidores. Cada nó servidor recebe em `xif` o intervalo total onde a integral deve ser calculada e responsabiliza-se por realizar o cálculo de um subintervalo correspondente à $\frac{1}{n-1}$ do intervalo total ($n-1$ == número de servidores); o resultado parcial de cada servidor é armazenado na sua própria cópia da variável `servarea`.

```
void func() {
  float xif[2];              //xi em xif[0] e xf em xif[1]
  float area = 0, serarea; //integral total e calculada no servidor
  int myrank;
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  if( myrank == 0 ) { // no nó raiz
    // obtêm xi e xf e armazena em xif
  }
  MPI_Bcast( xif, 2, MPI_FLOAT, 0, MPI_COMM_WORLD );
  if( myrank != 0 ) { //em um nó servidor
    // calcula a área em um subintervalo de (xif[1],xif[2])
    // e a armazena na variável servarea
  }
  ...
}
```

A redução é um mecanismo inverso ao *broadcast*, em que vários nós enviam mensagens (uma mensagem por nó) a um nó raiz. A sintaxe de `MPI_Reduce` é a seguinte:

```
int MPI_Reduce( void *operando, void* resultado, int cont, MPI_Datatype tipo,
                MPI_Op operador, int raiz, MPI_Comm grupo );
```

A quantidade de parâmetros necessários a uma operação de redução é maior que em um *broadcast* pois é necessário especificar a *ação de redução* a ser tomada quando os dados forem recebidos pelo nó raiz. Esta operação é especificada em `operador` (a lista de operações predefinidas em MPI se encontra na tabela 2, outras podem ser definidas pelo programador), que, no nó raiz é realizada sobre o valor recebido em `operando` e em `resultado`, armazenando o resultado em `resultado`. Em um outro nó que não seja o raiz, esta operação não é realizada, sendo o dado armazenado em `operando` transmitido na mensagem.

Tabela 2: Operações de redução predefinidas em MPI.

Operaçãp	`resultado` recebe
`MPI_MAX`	o maior valor entre `operando` e `resultado`
`MPI_MIN`	o menor valor entre `operando` e `resultado`
`MPI_SUM`	a soma de `operando` e `resultado`
`MPI_PROD`	o produto de `operando` e `resultado`
`MPI_LAND`	o E lógico entre `operando` e `resultado`
`MPI_BAND`	o E byte a byte entre `operando` e `resultado`
`MPI_LOR`	o OU lógico entre `operando` e `resultado`
`MPI_BOR`	o OU byte a byte entre `operando` e `resultado`
`MPI_LXOR`	o XOR lógico entre `operando` e `resultado`
`MPI_BXOR`	o XOR byte a byte entre `operando` e `resultado`
`MPI_MAXLOC`	o maior e a localização do maior
`MPI_MINLOC`	o menor e a localização do menor

Utilizando `MPI_Reduce`, podemos concluir o exemplo do cálculo de integral de uma função, inserido no final da função apresentada acima, o seguinte trecho de código:

```
void func() {
  ...
  MPI_Reduce( &servarea, &area, 1, MPI_FLOAT, MPI_SUM,
              0, MPI_COMM_WORLD );
  if( myrank == 0 ) { // no nó raiz
    // a variável area contêm a integral calculada
  }
}
```

3.4 Instalação e uso de LAM

A biblioteca LAM não faz parte das distribuições Linux convencionais, devendo ser instalada pelo usuário. O processo de instalação de LAM inicia com sua configuração

através do programa `configure`. Durante a configuração, diversos parâmetros podem ser selecionados, entre eles o diretório onde a biblioteca deve ser instalada. No exemplo abaixo esta sendo configurada uma instalação para LAM versão 6.4; foi solicitado que a biblioteca, após gerada, seja instalada no diretório `/usr/local/lam-6.4`. Observe que o proprietário do diretório solicitado é o usuário `root`, um usuário comum deve criar um subdiretório em sua própria árvore de diretórios. Note que LAM pode ser utilizada tanto por programas escritos em C/C++ como em Fortran, no caso foi solicitado a não geração da versão LAM para Fortran com o parâmetro `--without-fc`.

```
$ cd diretório_fontes_LAM
$ ./configure --prefix=/usr/local/lam-6.4 --without-fc
```

Após a configuração realizada, resta a fazer `make` para a biblioteca ser gerada e instalada no diretório solicitado:

```
$ make
```

Antes de utilizar LAM, certifique-se que a variável de ambiente `LAMHOME` contenha o diretório informado para instalação da biblioteca e a variável `PATH` inclua o diretório onde encontram-se os utilitários oferecidos (`lamboot`, `wipe`, etc.). Por exemplo, coloque as seguintes linha no seu arquivo `.bash_profile`:

```
export LAMHOME=/usr/local/lam-6.4
export PATH=$PATH:$LAMHOME/bin
```

A partir deste momento os serviços oferecidos por LAM podem ser utilizados. O primeiro passo é compilar o programa fonte, sendo a maneira mais simples através do aplicativo `mpicc`, que possui as mesmas funcionalidades do compilador C disponível no ambiente[3]. A linha de comando abaixo requisita a compilação do programa `prog1.c`, gerando o executável `prog`.

```
$ mpicc prog1.c -o prog
```

O executável `prog` pode ser executado através de uma sessão normal de MPI, como a apresentada na seção 3.1.

3.5 Exemplo

Um exemplo de um programa escrito em C utilizando MPI LAM pode ser visto na listagem a seguir. O programa apresentado calcula o valor do Fibonacci de um número, segundo o esquema de execução apresentado na figura 5. Na figura, é apresentado o esquema de execução do cálculo do valor da série de Fibonacci para o número 10 sobre uma arquitetura virtual composta de quatro nós.

[3] `mpiCC` para utilizar o compilador C++.

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

int Principal( int n, int myself, int group ) {
  int gres, rres, // Resultado global, resultado parcial produzido por um nó trabalhador
      i;
  MPI_Status status;

  if( myself == (group-1) ) return CalculaLocal( n ); // Apenas um nó, execução seqüencial
  else { // Mais de um nó virtual,
    int aux = n - 1; // Envia parte do problema ao próximo nó e ...
    MPI_Send( &aux, 1, MPI_INT, myself+1, 0, MPI_COMM_WORLD );
    gres = CalculaLocal( n - 2 ); // ... calcula local parte do problema

    for( i = 1 ; i < group ; i++ ) { // Finaliza regrupando todas as soluções parciais
      MPI_Recv( &rres, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status );
      gres += rres;
    }
  }
  return gres;
}

void Trabalhador( int myself, int group ) {
  int n, aux, lres;
  MPI_Status status;

  MPI_Recv( &n, 1, MPI_INT, myself-1, MPI_ANY_TAG, MPI_COMM_WORLD, &status );

  if( myself == (group-1) )      // Último nó virtual realiza todo
    lres = CalculaLocal( n );    // o cálculo local
  else {                         // Caso não seja o último,
    aux = n - 1;                 // envia o maior componente para o próximo nó
    MPI_Send( &aux, 1, MPI_INT, myself+1, 0, MPI_COMM_WORLD );
    lres = CalculaLocal( n - 2 ); // e calcula local a menor componente
  }
  MPI_Send( &lres, 1, MPI_INT, 0, 0, MPI_COMM_WORLD ); // Envia ao nó central o resultado
}                                                      // calculado localmente

int CalculaLocal( int n ) { // Cálculo seqüencial executado por cada nó
  if( n <= 2 ) return 1;
  else return CalculaLocal(n-1) + CalculaLocal(n-2);
}

int main( int argc, char **argv ) {
  int myself, group, res; // Identificação do nó, do número de nós e resultado final
  MPI_Status status;

  MPI_Init( &argc, &argv );                // Inicialização do LAM
  MPI_Comm_rank( MPI_COMM_WORLD, &myself );
  MPI_Comm_size( MPI_COMM_WORLD, &group );

  if( myself == 0 ) { // Repartição das tarefas, segundo o nó que está sendo executado
    res = Principal( atoi(argv[1]), myself, group ); // O nó 0 tem acesso aos parâmetros
    printf("O Fibonacci de %d e' %d", atoi(argv[1]), res); // e imprime resultado final
  }
  else Trabalhador( myself, group ); // Todos outros nós calculam parte da solução

  MPI_Finalize();
  return 0;
}
```

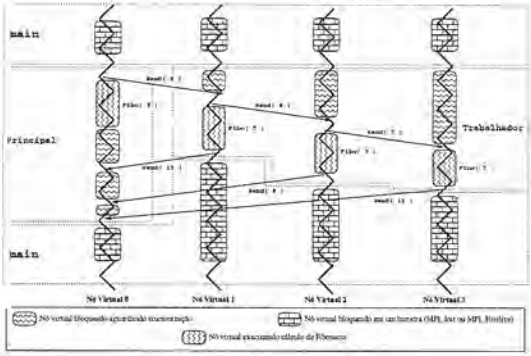


Figura 5. Esquema da execução do cálculo de Fibonacci de 10 em uma arquitetura composta por quatro nós.

4 Multiprogramação Leve

A ferramenta típica de exploração da concorrência intra-nó faz uso da multiprogramação leve (*multithreading*), ou seja, permite a criação de vários fluxos de execução no interior de um processo. Cada um destes fluxos de execução é chamados de processo leve ou *threads*. O termo "leve" faz referência ao fato de que os recursos de processamento alocados a um processo são compartilhados por todas suas *threads* ativas [23], não sendo necessário que cada *thread* possua sua própria descrição de recursos, a manipulação de *threads* é menos onerosa (mais leve) ao sistema operacional.

Dentre os recursos compartilhados pelas *threads*, a memória alocada ao processo desempenha um papel especial: é através desta memória que as *threads* compartilham dados e se comunicam.

Cabe salientar que uma *thread* não é necessariamente uma tarefa, devendo ser vista como um suporte à execução do conjunto de instruções pertencentes a uma tarefa. Esta distinção permite diferenciar a concorrência existente entre as atividades de uma aplicação da concorrência real que pode ser obtida em uma determinada arquitetura [4]: assim o programador restringe-se a definir as tarefas de sua aplicação, sendo estas associadas a *threads* no momento da execução do programa. Um mecanismo de fila pode ser empregado para armazenar tarefas prontas (cf. 1.1), no aguardo de serem disparadas sobre uma *thread* [19, 9]. *Thread* passa ser sinônimo de tarefa em certos programas uma *thread* é criada para suportar a execução de uma única tarefa, sendo destruída logo que esta tarefa for concluida.

Na próxima seção são apresentados os modelos básicos que definem o comportamento de execução de *threads* e na seqüência são apresentadas algumas funionalidades do padrão POSIX para *threads*.

4.1 Modelos de *threads*

Muitos sistemas operacionais, como Solaris, AIX e Linux, disponibilizam bibliotecas oferecendo recursos para a manipulação de *threads*. Em outros ambientes, como Minix e Windows 95/98, *threads* são disponíveis através de bibliotecas não integradas ao sistema operacional. O fato de serem ou não disponibilizadas diretamente pelo sistema operacional influi diretamente no modelo de *thread* oferecido. Os três modelos básicos, que podem ser identificados pelo mecanismo de escalonamento de *threads* ao processador, são [7]: 1:1 (*one-to-one*), N:1 (*many-to-one*) e M:N (*many-to-many*).

O modelo 1:1

Este modelo provê o que se convencionou chamar *threads* sistema (ou *kernel*). Estas *threads* são suportadas diretamente pelo sistema operacional, tendo os mesmos direitos a dos processos no escalonamento do processador. 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 que *threads* são sistema 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 na execução das atividades da aplicação. Pelo mesmo princípio, no momento em que uma *thread* sistema bloqueia suas atividades para executar uma operação de E/S, as demais continuam suas respectivas execuções sem nenhum prejuízo.

Modelo N:1

O modelo *many-to-one* é normalmente oferecido por bibliotecas no intuito de prover o recurso de *threads* quando estas não são oferecidas pelo sistema operacional. Neste caso as *threads* são denominadas *threads* usuário, executando ao mesmo nível da aplicação. Isto significa que as *threads* são escalonadas no interior do processo, quando este obtiver acesso ao processador. O fato de *threads* usuário serem escalonadas no interior de um processo implica que arquiteturas multiprocessadoras não sejam exploradas por completo e que todo o conjunto de *threads* usuário de um processo seja bloqueado quando uma destas *thread* iniciar uma operação de E/S.

Em contrapartida, a manipulação de *threads* usuário é ainda menos onerosa que a manipulação de *threads* sistema, o que possibilita o programador utilizar um número superior de *threads* em sua aplicação.

Modelo M:N
Finalmente o modelo *many-to-many* permite que as características de ambos modelos anteriores sejam mescladas. Neste modelo, no interior de cada processo podem existir 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 de *thread* usuário, em geral M é muito superior a N, reflete no desempenho de execução e o uso de *thread* sistema aporta as vantagens de um mesmo programa dispor de várias unidades de escalonamento. Outra vantagem é que o programador não precisa restringir o grau de concorrência de seu programa em função dos recursos de hardware disponíveis, bastando achar a relação entre *threads* sistema e *threads* usuário que oferece um bom compromisso de desempenho.
Solaris [17] oferece este modelo de *threads*, utilizando *light weight processes*, as LWPs.

4.2 Pthreads básico

A biblioteca Pthreads é uma biblioteca de *threads* que segue o padrão POSIX. Esta biblioteca é normalmente distribuída junto ao sistema operacional Linux, podendo ser utilizada em programas escritos em C/C++. Para poder utilizar a Pthread em seu programa, o programador deve fazer uso de dois arquivos oferecidos pela biblioteca: o arquivo de *header* e a biblioteca de funções. Os passos necessários para obter um executável são descritos abaixo[4].
Include A primeira preocupação que o programador deve ter é de incluir o arquivo header da biblioteca Pthreads em todo módulo de um programa que utilize alguma de suas primitivas:

```
#include <pthread.h> // arquivo header da biblioteca Pthread
```

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 `modulo1.o`, que corresponde ao código objeto do módulo `modulo1.c`. Neste exemplo, foi fornecido o parâmetro de compilação `-I` especificando um outro diretório além do default para buscar os *headers*; neste caso específico o uso deste parâmetro é desnecessário, pois o diretório informado é o próprio diretório default.

```
$ gcc -I/usr/include -c modulo1.c
```

[4] Os exemplos apresentados neste texto empregam a biblioteca POSIX de *threads* em Linux, a **Pthread**, e foram escritos na linguagem C, utilizando o compilador GNU 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; outro diretório pode ser informado através do parâmetro -L. A inclusão da biblioteca pthread deve ser explicitada pelo parâmetro -l de linkedição. Abaixo um exemplo:

```
$ gcc -L/usr/lib modulo1.o modulo2.o princ.o -lpthread -o princ
```

4.3 Criação e destruição de *threads*

Dentro de um programa C, o conjunto de instruções a ser executado por uma *thread* é definido por uma função. Esta função, construída pelo programador, deve receber como parâmetro um `void *` e igualmente retornar `void *`. O cabeçalho desta função é o seguinte:

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

A criação da *thread* se dá através da invocação da primitiva `pthread_create` dentro de um bloco qualquer de comandos. Esta primitiva interagem com a biblioteca de manipulação de *threads*, permitindo a criação e a manipulação de um novo fluxo de execução. A primitiva `pthread_create` possui o seguinte cabeçalho:

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

Cada invocação a `pthread_create` cria uma nova *thread* responsável pela execução da função `funcao`. Nesta criação, a biblioteca de *thread* pode ser instruída para manipular a *thread* com alguns atributos especiais, atributos estes fornecidos especificados pelo programador em `atrib` (se `atrib` é `NULL`, utiliza atributos default, o tipo de dado `pthread_attr_t` é detalhado a seguir). Eventuais parâmetros podem enviados para a nova *thread* através do ponteiro void `args`. O primeiro parâmetro requisitado, `thid`, recebe na execução da primitiva `pthread_create` um identificador da nova *thread* criada; este identificador é representado por um valor numérico, o que permite identificar as *threads* individualmente. O retorno da primitiva permite verificar se a operação ocorreu normalmente ou sem foi verificado algum erro, se o retorno for 0 (zero), a nova *thread* foi criada corretamente.
É comum encontrar casos onde lógica do algoritmo concorrente implementado necessita a identificação da *thread*. Neste caso, a primitiva `pthread_self` permite que uma *thread* tenha acesso ao seu identificador; abaixo seu protótipo.

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

Uma pthread criada tem uma tempo de vida igual ao tempo necessário à execução da função, ou seja, no momento em que a função termina, a *thread* é destruída. Caso uma *thread* necessite sincronizar com o término de uma outra, por exemplo para obter os dados de saída desta, a primitiva `pthread_join` pode ser empregada:

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

O uso do `pthread_join` permite que uma *thread* reste Bloqueada aguardando o término da computação da *thread* identificada em `thid`. 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.
Um exemplo da utilização deste primeiro conjunto de primitivas é apresentado a seguir.

```
1. #include <stdio.h>   //E/S em C
2. #include <stdlib.h>  //exit
3. #include <pthread.h> //biblioteca de threads
4.
5. void * OiMundo( void * str ) {
6.   printf( (char *) str );
7.   printf( "Eu sou a thread %d!!!\n", (int) pthread_self() );
8. }
9.
10. void main() {
11.   pthread_t thid;
12.   char *str = "Oi Mundo !!!!";
13.
14.   if( pthread_create( &thid, NULL, OiMundo, NULL ) != 0 ) {
15.     printf( "Ocorreu um erro!!!\n" );
16.     exit(0);
17.   }
18.   printf( "Foi criada a thread %d.\n", (int) thid );
19.   pthread_join( thid, NULL );
20.   printf( "A thread %d ja terminou.\n", (int) thid );
21. }
```

Após uma execução deste programa, as mensagens poderiam ser impressas na tela nas seguintes ordens:

Foi criada a thread 1.	Oi Mundo!!!	Eu sou a thread 1!!!
Oi Mundo!!!	Foi criada a thread 1.	Oi Mundo!!!
Eu sou a thread 1!!!	Eu sou a thread 1!!!	Foi criada a thread 1.
A thread 1 ja terminou.	A thread 1 ja terminou.	A thread 1 ja terminou.

Entre a criação da *thread* realizada na linha 14 e a operação de join realizada na linha 19, duas *threads* estão sendo executadas potencialmente ao mesmo tempo. Isto explica porque a mensagem "`Foi criada a thread 1.`" não tenha uma posição específica em relação as mensagens impressas pela função `OiMundo`. A mensagem "`A thread 1 ja terminou.`", por sua vez, será impressa sempre após as duas mensagens escritas por `OiMundo`. A sincronização oferecida pela primitiva `pthread_join`

garante que OiMundo está concluída, todos seus resultados estão portanto produzidos.
É possível pensar que as mensagens produzidas por duas *threads* possam aparecer intercaladas na tela, resultando em linhas na tela tais como "`Oi Foi criada Mundo!!!`". Isto não ocorre na maioria dos ambientes, pois as bibliotecas de entrada e saída de C normalmente incorporam elementos que permitem o tratamento de múltiplas chamadas simultâneas, evitando sobreposições de saídas. Diz-se que bibliotecas que tratam multiplas chamadas simultâneas são *thread-safe*.

Atributos de criação de *threads*

Dentre os parâmetro necessários a função `pthread_create` para a criação de uma *thread*, um informa os atributos que devem ser considerados para sua execução e manipulação. Estes atributos são informados através de um tipo de dado fornecido pela biblioteca Pthread: `pthread_attr_t`, que nada mais é do que uma estrutura onde cada campo corresponde a um atributo. Esta estrutura é manipulada por um conjunto de primitivas, entre elas as citadas abaixo:

```
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 detachstate);
```

Um dos atributos da *thread* permite permite informar se a ela poderá ou não ser sincronizada com uma outra através de uma operação de *join*. Caso o *join* deva ser realizado, o atributo a ser utilizado é `PTHREAD_CREATE_JOINABLE`, caso contrário, `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. 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 realizar o escalonamento da *thread*. São três as possibilidades `SCHED_OTHER` para utilizar a política default implementada na bilbioteca, `SCHED_RR`, que implementa uma política do tipo *Round-Robin* e `SCHED_FIFO`, que implementa uma política do tipo *first-in first-out*. As políticas `SCHED_RR` e `SCHED_FIFO` estão disponíveis apenas quando o processo em execução tiver privilégios de super-usuários. A segunda função permite informar se a política de escalonamento deve seguir aquela adotada pela *thread* que executou a operação de criação ou utilizar os parâmetros informados explicitamente

por `pthread_attr_setschedpolicy`. O default é que sejam utilizados os parâmetros que foram informados explicitamente, `PTHREAD_EXPLICIT_SCHED`, a herança pode ser selecionada através de `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*. 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.

4.4 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. O problema é garantir o correto acesso às informações pelas *threads*, a solução emprega algum mecanismo de sincronização: exclusão mutua ou coordenação (em [19] competição e cooperação, respectivamente). A sincronização no acesso à memória é necessária para limitar o indeterminismo na execução de programas concorrentes. 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[5].

Mutex

Suponha duas *threads* executando ao mesmo tempo e compartilhando uma variável inteira, como no exemplo abaixo:

`int x; // x é uma variável global`	
// Thread A	// Thread B
a = x; // lê dado compartilhada	b = x; // lê dado compartilhada
a = a + 1; // realiza uma operação	b = b - 1; // realiza uma operação
x = a; // altera dado compartilhado	x = b; // altera dado compartilhado

Este exemplo mostra claramente uma situação onde duas sessões críticas trabalham sobre uma mesma área de dados, 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.

[5] Outra forma de sincronismo foi vista anteriormente, com a primitiva `pthread_join`.

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

O que levaria `x` conter 312, um valor inconsistente.
Neste caso, é obrigatório garantir a exclusividade no acesso ao dado por uma sessão crítica, o que pode ser garantido com a utilização de um **mutex**.
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 é bastante simples, e 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 da seguinte 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, e é feita através de uma invocação à:

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

Inicializa o mutex `m` com os atributos `atrib`. O atributo pode ser utilizado para definir o valor inicial para mutex (aberto ou fechado). A opção default (aberto) pode ser selecionada passando `NULL` para `atrib`.
A manipulação permite executar exatamente as operações de "pegar" e "soltar" o mutex, *lock* e *unlock*, respectivamente:

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

Com o uso do mutex, o exemplo apresentado no início desta sessão será reapresentado, corrigido.

```
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	// Thread B
pthread_mutex_lock(&m);	pthread_mutex_lock(&m);
a = x; // lê dado compartilhada	b = x; // lê dado compartilhada
a = a + 1; // realiza uma operação	b = b - 1; // realiza uma operação
x = a; // altera dado compartilhado	x = b; // altera dado compartilhado
pthread_mutex_unlock(&m);	pthread_mutex_unlock(&m);

Observe no exemplo que o mutex `m` é uma variável como outra qualquer. Seu uso depende da lógica adotada no algoritmo implementado. Neste exemplo, o algoritmo associou o mutex m à variável `x`. Cada sessão crítica que acessara variável `x` deve explicitar as operações *lock* e *unlock*. Caso existissem outras variáveis compartilhadas, outros mutex poderiam ser criados e manipulados dentro do programa.

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 uma destas condições não estiver satisfeita, 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.

As variáveis de condição, associadas a um mutex, permitem sincronizar duas (ou mais) *threads* em uma alteração de memória. Um uso típico é na sincronização de *threads* em um algoritmo do tipo produtor/consumidor.

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 );
```

Inicializa a variável de condição `c` com os atributos `atrib`. O atributo pode ser utilizado para definir o valor inicial para a variável de condição (satisfeita ou não). A opção default (não satisfeita) pode ser selecionada informando o valor `NULL` para `atrib`.

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* permite que uma *thread* seja bloqueada na espera de uma sinalização. Observe que sempre que uma *thread* realizar uma operação *wait* ela será bloqueada na condição `c`. As duas outras primitivas permitem a sinalização de uma

condição, sendo que a primitiva *signal* libera a execução de apenas uma das *threads* que esteja bloqueada na condição, enquanto que a primitiva *broadcast* libera a execução de todas as *threads* bloqueadas.

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 automáticamente (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 para sincronizar a produção e o consumo de itens em um *buffer*. No algoritmo são utilizadas primitivas da biblioteca Pthread.

```
// Área de memória compartilhada entre o Produtor e o Consumidor
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 entre produtor e 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 Consumidor() {
 Item it;
 for( ; ; ) {
  pthreads_mutex_lock( &mb );
  while( nb_item <= 0 )
   pthread_cond_wait( &mb, &c );
  it = LeBuffer();
  nb_itens--;
  pthreads_mutex_unlock( &mb );
 }
}
```

É importante observar que uma sinalização em uma variável de condição não é memorizada. Somente as *threads* em estado de *wait* recebem o sinal.

Semáforo

Um outro mecanismo empregado para sincronizar a cooperação entre *threads* são os **semáforos**, os quais possibilitam o controle do avanço de um grupo de *threads*.

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 alemão *passeren* e *vrijgeven*, passar e liberar, respectivamente). O valor associado ao contador indica quantas *threads* tem permissão de avançar e executar instruções sobre os dados 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 do contador seja maior que 0, a *thread* tem sua passagem liberada; caso contrário – o valor do contador igual a 0 –, o contador não é decrementado e a *thread* não obtém liberação de passagem, restando bloqueada. A liberação da passagem é realizada através da operação V (executada por uma outra *thread*); esta operação incrementa o contador do semáforo. Caso hajam *threads* bloqueadas no momento de um V, uma das *threads* terá sua passagem liberada, voltando a ser 0 o valor do contador no mesmo instante.

Embora presente em diversas bibliotecas de *threads*, a Pthread para Linux não oferece este recurso. No entanto, semáforos podem ser facilmente implementados utilizando mutex e um contador global. Os algoritmos para as operações P e V são apresentados abaixo.

```
typedef struct {
  int             cont;   // o contador do semáforo
  pthread_mutex_t pv;     // controle de acesso às operações P e V
  pthread_cond_t  cond;   // controle de alterações no contador
} semaforo_t;
```

```
int SemaforoInit( semaforo_t * s, int inicial ) {
 int ret;
 if( inicial < 0 ) return inicial;     // valor inicial não valido
 ret = pthread_mutex_init( &(s->pv ), NULL );
 ret |= pthread_cond_init( &(s->cond), NULL );
 if( inicial == NULL ) s->cont = 0;    // valor default
 else s->cont = inicial;               // valor informado
 return ret;                           // indica inicialização
}                                      // correta (0) ou não
```

```
void P_Semaforo( semaforo_t * s ) {
 pthread_mutex_lock( s->pv );     // acesso exclusivo sobre cont
 while( s->cont == 0 )            // não libera caso cont == 0
   pthread_cond_wait( s->cond ); // aguarda uma operação V
 cont--;                          // agora cont > 0, decrementa e segue
 pthread_mutex_unlock( s->pv ); // a execução - apenas uma thread
}                                 // sai do laço while
```

```
void V_Semaforo( semaforo_t * s ) {
 pthread_mutex_lock( s->pv );      // acesso exclusivo sobre cont
 cont++;                           // libera a passagem de uma thread
 pthread_cond_signal( s->cond ); // sinaliza a liberação de passagem
 pthread_mutex_unlock( s->pv );
}
```

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*. Tal mecanismo é bastante útil em certos casos, onde o estado de um sinal necessita ser memorizado para um tratamento futuro. Em [20] encontramos a implementação de um algoritmo produtor/consumidor utilizando semáforos. Observa-se no entanto que o uso de um semáforo para tal algoritmo implica na utilização de dois mutex, o que pode prejudicar o desempenho de execução.

4.5 Exemplo

Nesta seção é apresentado um exemplo de utilização da biblioteca Pthread. Trata-se da implementação de um algoritmo que permite a ordenação de um vetor de tamanho t contendo elementos cujos valores estejam distribuídos de forma aleatória dentro de um intervalo entre 0 e um número máximo conhecido. O procedimento básico deste algoritmo é compor n "baldes" (*buckets*), e "preencher" cada um destes baldes com os elementos pertencentes a um sub-intervalo dos valores possíveis para os elementos do vetor. A figura 6 apresenta o esquema da execução.

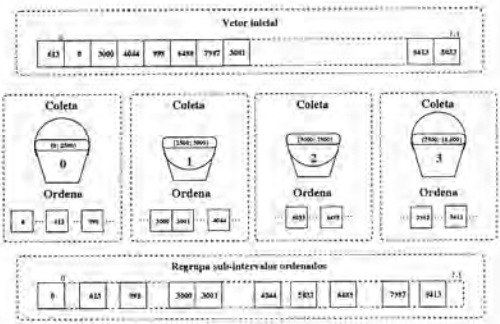


Figura 6: Esquema da execução paralela de ordenação de um vetor utilizando *buckets*.

No programa, listado na seqüência, foram omitidas as funções `greqthan` e `initVect`, devido a limitações de espaço. As *threads* criadas executam as operações de coleta dos elementos pertencentem a cada sub-intervalo em um sub-vetor, ou seja, preenchem cada um dos baldes, para em seguida ordena-lo.

A concatenação dos sub-vetores produzidos por cada *thread* permite obter um vetor resultado contendo os elementos do vetor inicial devidamente ordenados.

```
#include <pthread.h>
#include <string.h> // Utilização de memcpy
#define NUM_MAX 10000

struct dtain_t  int *vect, size, nb_buckets, my_bucket; ; // Entrada da thread
struct dtaout_t  int *bucket, size; ; // Saída da thread

void * Bucket( void * dtain ) {   // Função a ser executada pela thread
  int *bucket = NULL, b_size = 0, // Área de memória para o bucket
      buc_inf, buc_sup,           // Limite superior e inferior dos valores no bucket
      i, b;
  struct dtain_t  *dta = (struct dtain_t *) dtain;
  struct dtaout_t *dtaout;

  buc_inf = dta->my_bucket * NUM_MAX/dta->nb_buckets;
  buc_sup = buc_inf + NUM_MAX/dta->nb_buckets;

  for( i = 0, b = -1 ; i < dta->size ; i++ ) // Para todos os elementos do vetor
    if( (dta->vect[i] >= buc_inf) && (dta->vect[i] < buc_sup) ) { // O elemento pertence
      b++;                                                       // ao intervalo ?
      if( b == b_size ) {  // Aloca área de memória para o bucket conforme a necessidade
        b_size += dta->size/dta->nb_buckets;
        bucket = realloc( bucket, b_size * sizeof(int) );
      }
      bucket[b] = dta->vect[i]; // O valor pertence ao intervalo do bucket
    }
  free( (struct dtain_t *) dtain );
  if( b > 0 ) qsort( bucket, b+1, sizeof(int), greqthan ); // Ordena o bucket
  dtaout = (struct dtaout_t *) malloc( sizeof(struct dtaout_t) );
  dtaout->bucket = bucket; dtaout->size = b; // Dados de saída: o bucket e o seu tamanho
  return (void *) dtaout;
}
int main( int argc, char **argv ) {
  struct dtain_t  *in;    // Parâmetros de entrada da função a ser executada pela Thread
  struct dtaout_t *out;   // e os dados retornados por esta função
  int *vect, *resr, size, // O vetor de entrada, o vetor de saída e o número de elemantos
      nb_buckets, desl, i, j; // Número de threads a serem criadas e variáveis auxiliares
  pthread_t *thid;        // Vetor de descritores de threads

  size = atoi(argv[1]); nb_buckets = atoi(argv[2]); // Tamanho do vetor e número de buckets
  vect = initVect( size ); // Inicializa um vetor com valores aleatórios entre 0 e NUM_MAX
  thid = (pthread_t *) malloc( nb_buckets * sizeof(pthread_t) );
  for( i = 0 ; i < nb_buckets ; i++ ) { // Criação das nb_buckets threads
    in = (struct dtain_t *) malloc( sizeof(struct dtain_t) );
    in->size = size; in->vect = vect; in->nb_buckets = nb_buckets;
    in->my_bucket  = i; // A identificação da thread para cálculo dos limites do buckets
    pthread_create( (thid+i), NULL, Bucket, (void *)in );
  }
  res = (int *) malloc( size * sizeof(int) );
  for( i = 0 , desl = 0 ; i < nb_buckets ; i++ ) {
    pthread_join( *(thid+i), (void *)&out );
    if( out->size > -1 ) {
      memcpy( (void *)(res+desl), (void *) (out->bucket), (out->size + 1) * sizeof(int) );
      desl += out->size+1;
      free( (int *)(out->bucket) );
    }
    free( out );
  }
  return 0;
}
```

5 Processos Leves e Comunicações

Em um agregado, encontramos os dois níveis de concorrência aqui descritos. Nada mais natural que unir tanto os recursos de comunicação de processos quanto os de multiprogramação por processos leves. De fato, segundo o padrão MPI, nada impede que *threads* sejam utilizadas na programação de um nó. No entanto, são poucas as bibliotecas MPI que permitem associar *threads* aos processos usuário, por exemplo, a oferecida por Helwett-Packard para o sistema HP-UX 11.0 e a oferecida pela IBM para o sistema AIX 4.2.
Uma discussão sobre os problemas ligados ao uso conjunto de *threads* e comunicações pode ser encontrado em [7].
Mantendo a idéia de apresentar ambientes padronizados para exploração da concorrência, optou-se por discutir brevemente o uso de *threads* e de comunicações (RMI) em Java [10], mesmo considerando que até o momento atual, esta linguagem não pode ser considerada propriamente apta para o processamento de alto desempenho[6].

5.1 Sobreposição cálculo/comunicação

Como visto anteriormente, o compartilhamento de dados em um ambiente com memória distribuída faz, necessáriamente, uso de um mecanismo que permita a troca de mensagens entre tarefas executando em nós distintos. Uma vez que estas mensagens são transmitidas via um meio físico que é a rede de interconexão, um custo extra, refletido em tempo de processamento disperdiçado enquanto uma tarefa aguarda seus dados de entrada, pode ser adicionado a cada etapa envolvendo uma comunicação/sincronização. Tal situação pode ser vista na figura 5, onde os processos permanecem bloqueado enquanto aguardam que os dados atendidos pelas operações de comunicação sejam recebidos.
Um mecanismo que pode ser adotado para reduzir esta perda, e portanto melhorar o desempenho global de execução é utilizar multiplas *threads*, *multithreading*, em um processo: no momento em que uma destas é bloqueado em razão da necessidade de uma sincronização, uma outra *thread* pode ser ativada. A técnica básica adotada [24] consiste a simular q processadores virtuais executando sobre p processadores de uma arquitetura real ($q = p \log p$). Sobre esta técnica, encontramos diversas ferramentas, tais como Athapascan-0 [7], Nexus [12] e PM^2 [11], além de diversos trabalhos teóricos como em [14].

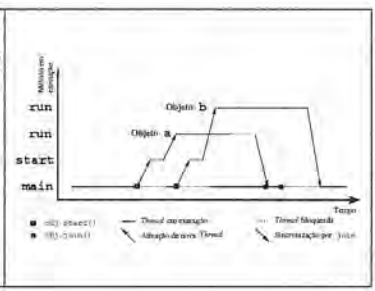
5.2 *Threads* em Java

Ao contrário das *threads* POSIX apresentadas anteriormente, o conceito de *thread* em Java está embutido na linguagem, não sendo necessário "incluir" uma biblioteca. No

[6] Esforços neste sentido tem sido realizado, maiores informações podem ser encontradas em www.jhpc.org e www.javagrande.org (novembro/2000).

caso, sendo Java uma linguagem orientada a objetos, para poder contar com objetos que possuam *threads* independentes, o programador deve definir classes que herdem da classe `Thread`. Desta classe são herdados três métodos cujo uso é associado à manipulação de *threads*: `run`, `start` e `join`.
O método `run` é um método virtual[7] na classe `Thread`, devendo ser implementado pela classe do usuário com o serviço que deve ser executado pela *thread* a ser criada. Os métodos `start` e `join` são implementados pela propria classe `Thread`. Eles permitem, respectivamente, iniciar a execução de uma nova *thread* e sincronizar com seu término.
É importante observar que a função a ser executada pela *thread* é definido por um método de uma classe. Assim sendo, para executar uma *thread*, é necessário a criação de um objeto da classe que especializa a classe `Thread`. Os métodos `start` e `join` são invocados neste objeto, e o método `run`, invocado implicitamente por `start`, é executado no escopo deste objeto.
Abaixo um exemplo simplificado[8] de manipulação de *threads* em Java. Também é apresentado um diagrama mostrando como poderiam estar distribuídos no tempo a execução concorrente das *threads*.

```
class ThExemplo extends Thread {
 public ThExemplo() { ... }
 public void run() { Implementação do serviço }
}
...
public void static main( String args[] ) {
 ThExemplo a = new ThExemplo(),
           b = new ThExemplo();
 a.start();
 b.start();
 ...
 a.join();
 b.join();
}
```



O controle da sincronização

O controle de execução de sessões críticas em Java é garantido por um mecanismo de monitores. Este mecanismo garante que determinada sessão de código será executada por apenas uma *thread* em um determinado instante de tempo. Caso uma *thread* deseje executar algum serviço do monitor, o primeiro passo é tentar obter a permissão de execução: caso obtenha a permissão, é garantido que o monitor não esta sendo utilizado por nenhuma outra *thread*. Caso a permissão seja negada, é sinal que uma outra *thread* esta sendo servida pelo monitor; neste caso a *thread* requisitante permanece bloqueada aguardando que o monitor seja liberado. Caso duas ou mais

[7] Muitos dos termos utilizados nesta seção referem-se a terminologia adotada em orientação a objetos. Maiores detalhes em [19] ou mesmo em [10].
[8] Todo tratamento de exceções foram suprimidos nos exemplos de programas em Java apresentados.

threads estejam bloqueadas aguardando o acesso ao monitor, no momento em que o monitor for liberado, uma das *threads* será escalonada a executar, enquanto a outra continuará aguardando. A regra mais comum é de permitir acesso ao monitor à *thread* que está aguardando a mais tempo, contudo esta regra não deve nunca ser considerada para provar a correção de um programa. A sincronização pode ser aplicada a diferentes níveis de granularidade: objeto, classe e bloco.
É de se notar que um monitor tem uma estrutura bastante próxima a um objeto: uma entidade que encapsula dados e fornece métodos para acessar estes dados. O mecanismo de monitores em Java permite garantir que trechos de código internos aos objetos sejam executados por apenas uma *thread*. Os mecanismos de sincronização entre *threads* são discutidos a seguir.

Sincronização a nível de objeto

A sincronização a nível de objeto garante que apenas uma *thread* possa executar métodos dentro do escopo de um objeto.
Esta forma de sincronização faz uso (implícito) do ponteiro `this`, permitindo que o objeto sirva uma única *thread* por vez. Seu uso é bastante intuitivo, como mostra o exemplo abaixo:

```
class Buffer {
 private Pilha p;
 public Buffer() { p = new Pilha(); }
 syncronized public void Insere( Object _o ) {
  p.Push( _o );
 }
 syncronized public Object Retira() {
  return p.Pop();
 }
}
```

A classe `Buffer` representa uma área de memória compartilhada entre duas *threads*, como no caso de um *buffer* para um algoritmo do tipo produtor/consumidor. `Insere` e `Retira` consistem em métodos que atuam sobre uma memória compartilhada, portanto sessões críticas. A garantia de que somente uma *thread* vai executar o código correspondente a um destes métodos de um determinado objeto `Buffer` que vir a ser criado é dada pelo modificador `syncronized`. Método que possuam este modificador garantem a interface de acesso aos serviços do objeto, executados no escopo do objeto – em outras palavras, se existirem dois objetos `Buffer` criados em um programa, ambos poderão executar ao mesmo tempo seus respectivos métodos `Insere` e `Retira`, mas um mesmo objeto que esteja sendo utilizado ao mesmo tempo por duas *threads* não poderá ter os métodos `Insere` e `Retira` executando mesmo tempo.

Sincronização a nível de classe

A sincronização a nível de classe garante que apenas uma *thread* possa executar métodos dentro do escopo de uma classe.

Uma classe pode possuir atributos de classe, ou seja, membros cujo escopo seja a classe, necessitando portanto de um controle de sincronização a nível de classe. Para garantir a exclusividade de execução de *threads* no escopo de uma classe, faz-se uso de métodos static. Seu uso remete ao exemplo anterior, como mostra o exemplo abaixo:

```
class Buffer {
 private static Pilha p;
 initialize { p = new Pilha(); }
 public Buffer() {}
 syncronized static public void Insere( Object _o ) {
  p.Push( _o );
 }
 syncronized static public Object Retira() {
  return p.Pop();
 }
}
```

Neste exemplo, a classe `Buffer` define que a área de dados a ser compartilhada entre os produtores e os consumidores é um atributo de classe, o que significa que todos produtores e consumidores compartilharão a mesma área de dados. Desta forma o escopo deste atributo é a classe, e, portanto, não pode ter seu acesso protegido no escopo de um objeto, mas deve ser protegido considerando a execução de todos os produtores e consumidores que venham a ser construídos.

Sincronização a nível de bloco

A sincronização a nível de bloco garante que apenas uma *thread* possa executar o conjunto de instruções definidas dentro do escopo de um bloco. Para controlar a execução de uma sessão crítica definida por um bloco, é utilizado um mecanismo de sincronização apoiado em um objeto auxiliar. Este objeto auxiliar é necessário pois, ao contrário dos mecanismos descritos anteriormente, não existe um ponto de apoio à sincronização. Este objeto auxiliar pode ser um objeto qualquer, mas a garantia de correção semântica é de responsabilidade do usuário que deve utilizar sempre o mesmo objeto para blocos que compartilhem um mesmo dado. O único cuidado é que este objeto não seja `null` e deve ser evitado seu uso dentro da sessão crítica. Seu pode ser exemplificado da seguinte forma:

```
class Buffer {
 private Pilha p;
 public Buffer() { p = new Pilha(); }
 public void Insere( Object _o ) {
  syncronized( this ) { p.Push( _o ); }
 }
 syncronized public Object Retira() {
  Object aux;
  syncronized( this ) { aux = p.Pop(); }
  return aux;
 }
}
```

Neste exemplo, retornamos à primeira implementação dada à classe `Buffer`; observe que a presente implementação oferece um resultado semântico idêntico ao proposto anteriormente. O objeto que esta sendo utilizado para garantir o acesso à sessão crítica é a própria referência ao objeto: a referência `this`. Observe que caso duas *threads* tentem executar os serviços Insere ou Retira, será permitido: o monitor é aplicado somente no momento em que iniciar a execução da sessão crítica propriamente dita: o acesso ao *buffer*. Este mecanismo é especialmente interessante no caso em que os trechos de sessões críticas são pequenos, não justificando a sincronização efetuada no método.

5.3 Invocação remota de métodos

O objetivo do RMI – *Remote Method Invocation* – em Java é de permitir a execução de métodos em objetos remotos. Algo semelhante ao tradicional mecanismo de RPC – *Remote Procedure Call* –. Utilizando RMI, o mecanismo de invocação a um método em um objeto remoto é realizado de forma síncrona: o fluxo de execução onde foi realizada a chamada ao serviço remoto permanece bloqueado enquanto não for recebido o retorno dos resultados.
A programação utilizando RMI necessita que sejam fornecidas três classes: uma classe **servidora**, uma classe **cliente** e uma classe de **interface** aos acessos remotos.

Classe de interface

Nesta classe encontram-se definidos os serviços a serem oferecidos por um objeto remoto. No exemplo a seguir é apresentada a interface para um objeto servidor de cálculo do fatorial de um número.

```
public interface FatorialRemoto extends java.rmi.Remote {
 public int fatorial( int n );
}
```

Nesta classe está especificado que `FatorialRemoto` define uma interface de serviços para um objeto remoto. O servidor oferece um método chamado `fatorial`, que recebe um valor inteiro como parâmetro, o número para o qual o fatorial deve ser

calculado, e retorna outro inteiro como resultado. Da classe `java.rmi.Remote` são herdadas as funionalidades que permitem a invocação remota de métodos.
Em relação a passagem de parâmetros (e retorno de resultados), deve-se observar que ela é realizada *por valor*. Ou seja, o método remoto trabalha com cópias locais dos parâmetros recebidos. Salienta-se que todo objeto enviado como parâmetro a um objeto remoto deve implementar a classe `java.io.Serializable`, que é o caso dos `int`.

A classe servidora

A classe servidora implementa a interface dos serviços especificados por uma interface. No exemplo abaixo a implementação de um servidor para o cálculo do fatorial.

```
class FatorialServidor extends UnicastRemoteObject
                                implements FatorialRemoto {
 public int fatorial( int n ) {
  implementação do serviço
 }
}
```

A classe `UnicastRemoteObject`, herdada pela classe servidora, implementa as funcionalidades básicas necessárias a objetos servidores de invocações remotas. Entre estas funcionalidades, está a exportação do objeto, permitindo a comunicação deste com um cliente (via um protocolo *unicast*).
A esta classe deve estar associado um código `main`, o qual é responsável por (*i*) criar um objeto servidor e (*ii*) associar um nome lógico a este objeto.

```
public void static main( String args [] ) {
 FatorialServidor fs = new FatorialServidor();
 Naming.rebind("fatorial", fs);
}
```

A associação do objeto servidor com o nome lógico é feito através da invocação ao método `Naming.rebind`. No caso, o nome lógico é `fatorial`.

A classe cliente

Para utilizar os serviços de um método remoto, o cliente deve criar um objeto da classe interface e associa-lo ao objeto remoto servidor. Isto é feito através do método `Naming.lookup`, que possibilita que um objeto servidor seja buscado em um nó remoto a partir de seu nome lógico.

```
public class ClienteFatorial {
 ...
 public void static main( String args [] ) {
  FatorialRemoto fat;
  int res;
  fat = (FatorialRemoto) Naming.lookup(" nome da máquina/fatorial");

  res = fat.fatorial(10);
 }
}
```

Observe que, após a associação do objeto interface com o objeto servidor, o acesso aos métodos do objeto servidor é realizado como se ele estivesse sendo acessado localmente.

5.4 Compilação e execução

Como é sabido, programas escritos em Java são executados sobre uma máquina virtual, a JVM – *Java Virtual Machine* –. Em geral, cada classe é escrita em um arquivo independente, este arquivo tendo como nome o próprio nome da classe e a extensão `.java`. A compilação se dá através de uma invocação ao compilador `javac`, por exemplo:

```
$ javac ClienteFatorial.java
```

produz como saída o arquivo `Cliente.class` que é o código objeto (para a JVM) gerado.
Este programa pode ser executado através da JVM da seguinte forma:

```
$ java ClienteFatorial
```

Caso trate-se de uma aplicação RMI, alguns passos devem ser adicionados. Após a compilação com `javac`, devem ser gerados dois outros arquivos a partir da classe servidora com um segundo compilador, o `rmic`. Estes arquivos tem como primeira parte do nome o nome da classe servidora e terminam por `_Stub` e `_Skel`. Portanto, para a classe `FatorialServidor`, a execução de

```
$ rmic FatorialServidor
```

ira gerar os arquivos `FatorialServidor_Stub` e `FatorialServidor_Skel`. Uma cópia do arquivo `FatorialServidor_Stub` deve se encontrar na máquina onde o cliente for executar.
O cliente, como foi visto acima, pode ser executado em qualquer máquina sem nenhuma manipulação especial. O único cuidado, é lança-lo somente após que o objeto servidor esteja em execução. Para lançar o servidor, é necessário primeiro inicializar o registro de RMI através do comando `rmiregistry` e em seguida executar através da JVM programa que implementa os serviços:

```
$ rmiregistry
$ java FatorialServidor
```