Ferramentas para Programação Paralela em Ambientes de Memória Distribuída

Simone de L. Martins   Celso C. Ribeiro   Noemi de la R. Rodriguez

e-mail: {simone,celso,noemi}@inf.puc-rio.br

Pontifícia Universidade Católica do Rio de Janeiro
Department of Computer Science
Rua Marquês de São Vicente 225
Rio de Janeiro 22453-970
Brazil

PUCRioInf-MCC01/96

Janeiro 1996
Revisão: Junho 1996

Abstract:

This article presents a survey of software tools for parallel computing on a distributed memory environment. Some concepts of parallel programming are introduced. Some particular software tools for distributed memory environments (such as PVM, p4, Express, and Linda) are described in more details and compared in terms of performance, support, and ease of use.

Keywords: parallel programming, distributed memory, software tools

Resumo:

Neste trabalho é apresentado um levantamento sobre ferramentas de *software* para computação paralela em ambientes de memória distribuída. Alguns conceitos de programação paralela são introduzidos. Algumas ferramentas específicas para ambientes de memória distribuída (tais como PVM, p4, Express e Linda) são descritas em maiores detalhes e comparadas em termos de facilidade de uso, suporte e desempenho.

Palavras-chaves: programação paralela, memória distribuída, ferramentas de *software*

1 Introdução

A demanda por sistemas de computação mais rápidos e eficientes, tanto na área de computação científica como na área comercial, vem crescendo bastante no decorrer dos últimos anos. Os limites físicos existentes para o aumento da velocidade de um único processador e o alto custo associado ao desenvolvimento destes processadores mais velozes levaram ao surgimento de novas arquiteturas para sistemas de computação de alto desempenho. Estas novas arquiteturas têm como objetivo integrar dezenas ou até mesmo milhares de processadores rápidos e de custo moderado, de modo a compor uma máquina de alto desempenho. Estas máquinas podem ser classificadas basicamente em três tipos [32]:

- Multiprocessadores vetoriais: possuem um pequeno número de processadores, onde cada um é um processador vetorial de alta performance.
- Sistemas MPP (*Massively Parallel Processors*): possuem de centenas a milhares de processadores interconectados e constituem a linha principal de computadores paralelos. Podem ter memória distribuída ou compartilhada.
- Estações de trabalho conectadas por redes: máquinas interligadas por redes de média e alta velocidade, que podem trabalhar como uma única máquina virtual de alto desempenho.

Podemos ainda classificar as máquinas na última destas categorias em dois tipos:

- Sistemas projetados como máquinas paralelas, tipicamente formados por um conjunto de máquinas com processadores semelhantes e mesmo sistema operacional. Nestes sistemas, a conexão entre as máquinas é, em geral, baseada em um *switch* ou em uma rede de alta velocidade.
- Sistemas projetados inicialmente para operarem como uma rede local de estações de trabalho autônomas, tipicamente formados por máquinas com características de hardware bastante diferentes e sistemas operacionais distintos. Nestes sistemas, a conexão entre as máquinas é, em geral, baseada em uma rede de média velocidade.

A importância desta distinção é que o segundo tipo de plataforma tem um custo muito inferior ao das demais máquinas paralelas. Além disto, ela já se encontra disponível em um número muito grande de instituições, principalmente sob a forma de redes locais de estações de trabalho executando diferentes variantes do sistema operacional Unix.

A disponibilidade de arquiteturas de baixo custo é fundamental para a viabilidade do paralelismo como solução realista para aumento do desempenho em aplicações. No entanto, esta viabilidade depende ainda de vários outros fatores. Os algoritmos utilizados devem ser especialmente projetados para execução em paralelo, uma vez que nem sempre a melhor solução paralela para um problema é simplesmente uma adaptação da melhor solução sequencial. Esse assunto é tema de vários trabalhos recentes, como [16, 23]. Além disto, o programador deve dispor de ferramentas de programação que o auxiliem na implementação destes algoritmos.

O conjunto de ferramentas de programação disponível influencia diretamente o desempenho obtido nestes novos ambientes. Normalmente, estas ferramentas são comparadas e analisadas somente segundo o tempo de execução (ou eficiência paralela) de aplicações desenvolvidas com elas. No entanto, pode ser interessante levar-se outras métricas em consideração, como levantado em [16]. A facilidade de construção de uma aplicação e o tempo que um programador gasta nesta atividade podem em muitos casos ser fundamentais. O desenvolvimento de aplicações paralelas introduz diversos novos problemas de programação, em especial relacionados à comunicação e sincronização entre os vários componentes de um programa. O nível de suporte que uma ferramenta dá ao tratamento destes problemas tem uma influência direta sobre a qualidade das soluções implementadas. Algumas soluções podem nem mesmo ser viáveis se a ferramenta não oferece suporte para o modelo de programação apropriado [17].

Nesse trabalho, apresenta-se um levantamento de alguns sistemas de *software* para programação paralela em ambientes de memória distribuída, ou seja, sistemas de *hardware* formados por estações de trabalho conectados por redes. Essas ferramentas são analisadas sob o ponto de vista de disponibilidade, facilidade de instalação e utilização, e modelos de programação oferecidos. Apresenta-se também uma comparação entre alguns destes sistemas, onde se discute, além dos pontos acima, características de desempenho.

2 Considerações sobre programação paralela

Apresentam-se nesta seção algumas considerações sobre programação paralela, de modo a definir alguns conceitos que serão mencionados no decorrer do trabalho. Os termos mais utilizados nesta área são introduzidos e tanto os modelos de acesso à memória existentes em ambientes distribuídos, como os modelos de programação mais utilizados em processamento paralelo são apresentados.

2.1 Terminologia

Alguns termos utilizados em processamento paralelo são descritos a seguir [15, 16] :

- Tarefa ou processo: uma unidade lógicamente distinta de um programa.
- Execução seqüencial: execução de um programa em um único processador, onde uma instrução é processada de cada vez.
- Paralelização de código: transformação de um programa seqüencial em paralelo, através da identificação de tarefas independentes para o programa, de modo a executá-lo de modo paralelo. Geralmente, requer mudanças no código e no algoritmo utilizado no programa seqüencial.
- Aceleração de um programa: razão entre os tempos de execução das versões seqüencial e paralela de um mesmo programa ou algoritmo. Para um determinado algoritmo, considera-se a implementação seqüencial executada em menor tempo como referencial para comparação com a implementação paralela [23]:

$$\text{aceleração} = \frac{\text{tempo de execução do programa seqüencial}}{\text{tempo de execução da versão paralela}}$$

- Eficiência de um programa: razão entre a aceleração de um programa e o número de processadores utilizados para a execução do programa.

$$\text{eficiência} = \frac{\text{aceleração}}{\text{número de processadores}}$$

  Este parâmetro é uma medida da fração do tempo que o processador está sendo utilizado para executar o algoritmo. Idealmente, se p processadores utilizassem 100 % do seu tempo somente para operações de computação do algoritmo, a eficiência seria igual a 1. Como o programa paralelo requer comunicação entre os processadores, parte do tempo dos processadores é gasto nesta atividade, levando a valores de eficiência inferiores a 1.

- Sincronização: coordenação temporal entre tarefas, usada para a troca de informações entre elas. É um fator de decréscimo na aceleração atingida por um programa paralelo, devido ao fato de alguns processadores poderem ficar inativos, esperando pelo término de outras tarefas em outros processadores. A comunicação entre processadores é dita síncrona quando exige esta coordenação entre tarefas.

- Granularidade: volume de processamento realizado por cada tarefa, em relação ao volume de comunicação existente entre as tarefas. Um programa muito granular possui tarefas que realizam poucas instruções e necessitam se comunicar muito, enquanto um programa pouco granular possui tarefas que executam muitas instruções e se comunicam pouco. Caso a comunicação entre os processadores seja síncrona, a granularidade afeta o tempo de execução de um programa. Isto porque programas com maior granularidade necessitam de sincronização mais freqüente, introduzindo períodos de inatividade dos processadores e aumentando o tempo total de execução.

- Escalabilidade: diz-se que um sistema definido por um algoritmo paralelo e uma máquina paralela é escalável se a aceleração atingida cresce proporcionalmente ao número de processadores do ambiente de *hardware*. Os fatores que contribuem para a escalabilidade de um sistema são o *hardware* do sistema, o algoritmo utilizado e o código implementado.

- Balanceamento de carga: distribuição das tarefas entre os processadores, de modo a garantir a execução mais eficiente possível do programa paralelo. Este balanceamento pode ser feito de maneira estática, antes do início da execução, ou de maneira dinâmica, em tempo de execução.

- SPMD (*Single Program Multiple Data*): modelo de programação bastante utilizado, no qual todos os processadores executam o mesmo programa sobre diferentes conjuntos de dados.

- Determinismo: um algoritmo ou programa é determinístico se a sua execução com uma determinada entrada sempre resulta na mesma saída. Diz-se que um programa é não-determinístico caso exista a possibilidade de se obter saídas diferentes para várias execuções do programa, utilizando-se sempre a mesma entrada.

2.2 Modelos de organização da memória

Existem dois tipos básicos de organização da memória [15, 23]: memória compartilhada e memória distribuída.

Em máquinas que utilizam memória compartilhada, todos os processadores podem acessar quaisquer endereços de memória, conforme mostrado na Figura 1. A sincronização entre tarefas

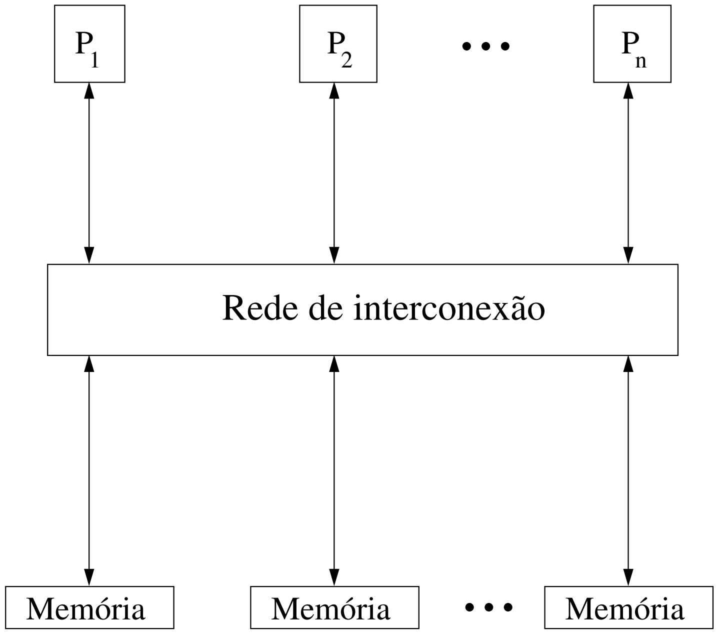

Figura 1: Arquitetura de memória compartilhada

é realizada através do controle das operações de leitura e escrita executadas pelas tarefas na memória compartilhada. A vantagem deste tipo de acesso é que o compartilhamento de dados pode ser executado de maneira rápida. A desvantagem está no nível de escalabilidade do sistema, que é limitado pelo número de caminhos existentes entre os processadores e a memória. Uma maneira de se eliminar esta desvantagem consiste em prover cada processador com uma memória local. Esta memória armazena o programa a ser executado e estruturas de dados que não são compartilhadas entre os processadores. As estruturas globais ficam armazenadas na memória compartilhada, conforme ilustrado na Figura 2. Uma extensão deste tipo de arquitetura consiste na eliminação total de compartilhamento físico da memória, conforme mostrado na Figura 3. Em ambos os casos, as referências feitas por um processador à memória de outro processador são mapeadas pelo *hardware* de interconexão da memória com os processadores. Deve-se lembrar que os tempos de acesso às memórias locais são tipicamente bem menores do que os tempos de acesso à memória remota. Um fator importante para a correta utilização deste modelo é o controle sobre o acesso à memória compartilhada pelas tarefas. Por exemplo, uma tarefa não pode alterar um dado enquanto outra tarefa está lendo este mesmo dado.

Já nas máquinas que utilizam memória distribuída, a memória é distribuída fisicamente entre os processadores e cada memória local só pode ser acessada pelo processador ao qual ela está associada. A interacão entre as tarefas é realizada através da troca de mensagens entre elas pela rede de comunicação, conforme ilustrado na Figura 4. O envio e o recebimento de mensagens entre os processadores pode ser feito de forma bloqueante ou não-bloqueante. No envio bloqueante, a execução da tarefa que gerou o envio de uma mensagem é interrompida até que a mensagem seja recebida pelo receptor, enquanto que, na forma não-bloqueante, ela inicia o processo de envio e continua sua execução, sem aguardar a confirmação de que a mensagem foi recebida pelo

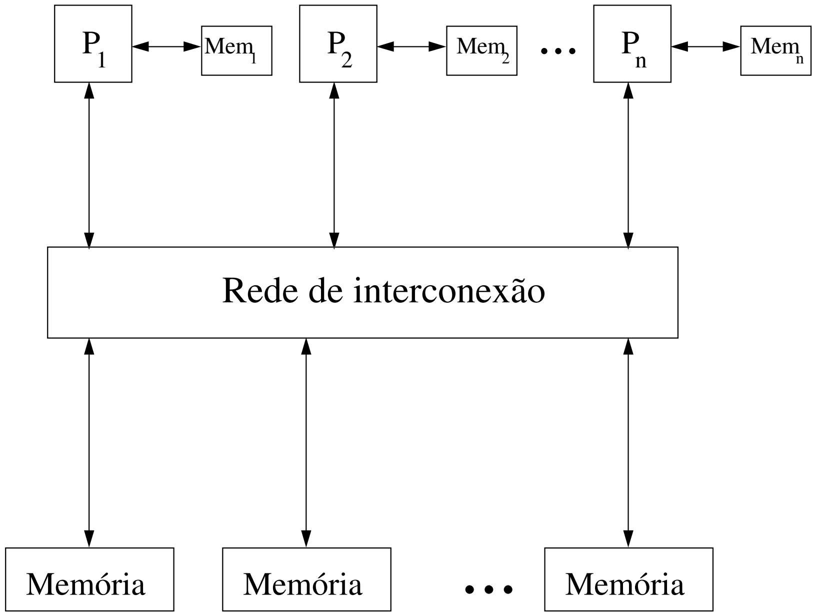

Figura 2: Arquitetura de memória compartilhada com memória local nos processadores

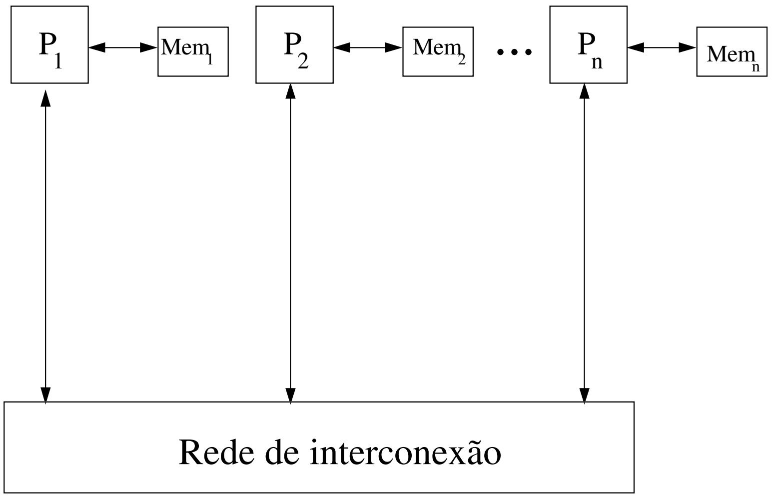

Figura 3: Arquitetura de memória compartilhada somente com memória local nos processadores

receptor. No recebimento bloqueante, a tarefa é interrompida enquanto espera a chegada de uma determinada mensagem. Já na forma não-bloqueante, ela verifica se a mensagem desejada está disponível e, caso não esteja, ela continua o processamento. As vantagens da utilização de memória distribuída são a escalabilidade do sistema, a possibilidade de maior confiança no sistema através da duplicação de dados entre os processadores e, conseqüentemente, tolerância a falhas e a facilidade de incorporação de processadores especializados. As desvantagens são a dificuldade de implementação de alguns algoritmos, tais como a decomposição dos dados de um *array*, e a reestruturação de implementações já existentes, sejam elas para aplicações seqüenciais ou então aquelas que utilizem memória compartilhada.

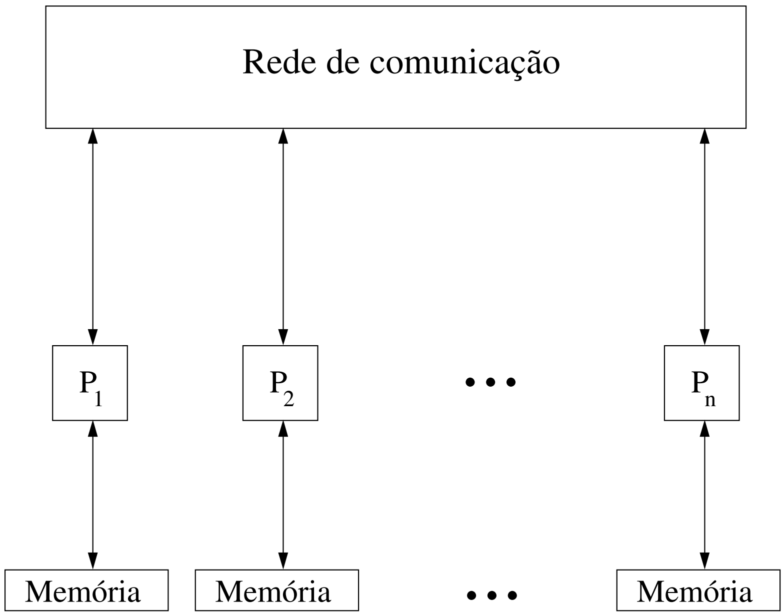

Figura 4: Arquitetura de memória distribuída

Deve-se notar a semelhança entre a arquitetura de memória compartilhada ilustrada na Figura 3 e a de memória distribuída mostrada na Figura 4. A maior diferença entre elas é que, na primeira, a rede de interconexão permite que os processadores executem operações de escrita e leitura em memórias pertencentes a outros processadores. Na segunda, o acesso à memória de um outro processador tem que ser feito de maneira explícita através da troca de mensagens entre os dois processadores.

É relativamente fácil fazer a emulação de uma arquitetura de memória distribuída em uma máquina com memória fisicamente compartilhada. Para isso, pode-se dividir o espaço da memória compartilhada em n espaços de endereçamento distintos, onde n é o número de processadores, e associar cada partição a um processador. O envio de uma mensagem de um processador p_i para um processador p_j é executado por uma operação de escrita por p_i na partição de memória associada a p_j. A tarefa de emular uma máquina de arquitetura de memória compartilhada em uma máquina de memória distribuída apresenta um custo mais elevado, porque cada operação de leitura e escrita na memória compartilhada implica em envio e recebimento de mensagens pelos processadores.

2.3 Modos de programação

Como já comentado anteriormente, a programação paralela é uma área relativamente recente, onde padrões ainda não foram definidos. De uma forma bastante geral, os modos de programação paralela atualmente utilizados podem ser classificados em três modelos [33]:

- Paralelismo de dados: os dados são distribuídos entre os processadores, que executam a mesma seqüência de instruções sobre eles. Um fator importante para a implementação eficiente deste tipo de programação é o gerenciamento da localização dos dados. Para que um programa deste tipo execute eficientemente em máquinas com memória distribuída, os dados devem estar alocados na memória de processadores próximos daqueles que vão necessitar destes dados. Este modelo também é conhecido como decomposição de domínio.
- Paralelismo de tarefas: o programa é particionado em tarefas cooperativas. Estas tarefas podem executar procedimentos bastante diferentes entre si, cujo processamento pode ser feito de maneira assíncrona. Para maior eficiência da implementação dos programas, deve-se levar em conta o gerenciamento da localização dos dados e o nível de granularidade de cada tarefa. Este modelo também é conhecido como decomposição funcional.
- Paralelismo de objetos: neste caso, o paralelismo pode ser realizado de diferentes formas, como por exemplo, (1) uma função membro de um objeto pode chamar uma função pública de outro objeto remoto ou não, e (2) um tipo abstrato de dados, como um *array*, pode ser implementado de uma maneira distribuída. São necessários os mesmos cuidados relacionados à eficiência discutidos no item anterior com relação à localidade dos dados e granularidade das tarefas.

Dependendo da aplicação, um modelo poderá ser mais adequado do que outro. Em muitas aplicações, uma combinação dos paradigmas acima pode ser útil para um melhor desempenho do programa.

3 Ferramentas para ambientes de memória distribuída

Como no caso de programação seqüencial, existem várias linguagens e ferramentas para a implementação de programas paralelos, sendo que cada uma delas apresenta uma maior adequação para uma determinada classe de problemas. A escolha da ferramenta a ser utilizada para o desenvolvimento de um programa paralelo depende da natureza do problema a ser resolvido. Algumas ferramentas são mais apropriadas para algoritmos numéricos baseados em decomposição regular do domínio, outras para aplicações que envolvam a criação dinâmica de tarefas e estruturas de dados irregulares e heterogêneas, ou ainda algoritmos que possuam uma estrutura SPMD bastante regular [16].

Nesta seção são apresentadas algumas ferramentas de *software* para ambientes de computação com memória distribuída, sendo que algumas delas também podem trabalhar com memória compartilhada. Elas estão divididas em duas classes: **linguagens paralelas** e **bibliotecas para troca de mensagens**. A primeira classe abrange as linguagens que possuem instruções especiais para a criação de aplicações paralelas pelos usuários. A segunda classe contém as ferramentas que oferecem funções de uma biblioteca de *software* para o usuário fazer a comunicação entre as tarefas de seu programa paralelo.

3.1 Linguagens Paralelas

A maior parte destas linguagens possui extensões feitas a linguagens seqüenciais já existentes, de modo a tornar transparentes ao usuário as primitivas de comunicação utilizadas no sistema de computação, numa tentativa de facilitar a implementação de programas paralelos. Estas linguagens possuem um compilador que transforma o programa do usuário e envia o resultado para um compilador da linguagem seqüencial.

Devido à grande diversidade destas linguagens [5, 35, 37], optou-se pela descrição de apenas algumas delas. Esta escolha foi feita visando mostrar a variedade de linguagens seqüenciais em que podem ser baseadas e o tipo de modelo de programação escolhido (paralelismo de dados ou paralelismo de tarefas, por exemplo), incluindo-se entre as linguagens consideradas algumas das mais importantes na área.

3.1.1 HPF

A linguagem **HPF** (*High Performance Fortran*) [16, 21] inclui algumas novas diretivas e instruções na linguagem Fortran 90, de modo a facilitar a geração de um código paralelo mais eficiente pelo compilador.

HPF é baseada em paralelismo de dados, ou seja, a concorrência na programação é obtida através da aplicação de uma mesma operação sobre subconjuntos diferentes de um determinado conjunto de dados. Em HPF, este conjunto de dados consiste de *arrays*, que são estruturas regulares. Em outras linguagens, tais como pC++ [22], este paralelismo pode ser estendido para estruturas irregulares, como árvores e conjuntos. O paralelismo de dados é obtido através de operações em *arrays*, que devem ser traduzidas automaticamente pelo compilador. Em um ambiente de memória distribuída, por exemplo, o compilador traduziria o programa em HPF em um programa SPMD, no qual cada processador executa o mesmo código em um subconjunto diferente dos dados. Neste caso, um fator essencial para a eficiência dos programas é a localidade dos dados. Como a otimização do mapeamento dos dados nos processadores pode ser difícil de ser executada automaticamente, a HPF oferece diretivas para o usuário fazer explicitamente esta distribuição.

Algumas ferramentas para o HPF são o FORGE 90, que é um paralelizador automático de programas seqüenciais Fortran para HPF [3] e o Adaptor [32], que traduz programas HPF para um código que contém troca de mensagens explicitamente. Uma implementação do HPF é comercializada pela *Applied Parallel Research* [3].

3.1.2 Fortran-M

A linguagem **Fortran-M** [16] é formada por um pequeno conjunto de extensões feitas sobre Fortran 77, que visam dar suporte a uma abordagem modular para a programação paralela, possibilitar a criação de programas paralelos determinísticos, permitir a especificação de processos dinâmicos e estruturas de comunicação entre processos, prover a integração entre paralelismo de dados e de tarefas, e possibilitar otimizações (executadas pelo compilador) em relação à comunicação e ao processamento.

Fortran-M oferece facilidades para atingir estes objetivos, tais como extensões para criação de tarefas e canais, para envio de mensagens nos canais, para mapeamento de tarefas e para armazenamento de dados em processadores da rede. Para preservar o determinismo da execução do programa paralelo, as instruções básicas não permitem que múltiplos processos recebam ou enviem dados em um mesmo canal. Caso seja necessária a comunicação de muitos processos para um, pode-se utilizar uma instrução específica que permite este tipo de comunicação, e, neste caso, os programas paralelos não têm a execução determinística garantida. O paradigma básico desta linguagem é o paralelismo de tarefas e seu objetivo é integrar-se com a linguagem HPF de modo a combinar os paradigmas de paralelismo de dados e de tarefas em um único sistema.

Esta linguagem é de domínio público e pode ser obtida através de ftp anônimo no diretório `pub/fortran-m` no endereço `ftp.mcs.anl.gov`.

3.1.3 Network Linda

A linguagem **Network Linda** [11, 12] (ou, simplesmente, **Linda**) oferece uma coleção de primitivas para criação e comunicação de processos, utilizando o conceito de memória associativa compartilhada. Esta memória recebe o nome de *tuplespace* (TS) e consiste de uma coleção de registros de dados chamados tuplas. As primitivas oferecidas por Linda permitem que os processos depositem tuplas no TS e que retirem ou leiam do TS tuplas que correspondam a um padrão especificado. Deste modo, se obtém um acesso associativo à memória, no qual a informação é obtida pelo conteúdo e não pelo endereço da mesma.

Embora o TS seja logicamente compartilhado pelos processos, ele pode ser implementado em ambientes de arquitetura de memória distribuída, sendo as diferentes partes distribuídas entre as memórias locais de cada processador. Deste modo, pode-se utilizar o TS para o armazenamento de estruturas de dados distribuídas e as tarefas sendo executadas por cada processador podem operar concorrentemente sobre diferentes elementos de dados.

Uma implementação comercial de Linda é fornecida pela *Scientific Computing Associates* [31]. Nesta implementação são fornecidas primitivas para as linguagens C e Fortran.

3.1.4 CC++

A linguagem **Compositional C++** (CC++) [10, 16] fornece algumas extensões para a linguagem seqüencial C++, de modo a fornecer um sistema que seja adequado para uma ampla variedade de modos de programação paralela. Pode ser utilizada em ambientes com memória compartilhada ou distribuída, possibilita o desenvolvimento de programas com diferentes granularidades, permite o uso de diferentes paradigmas de programação (tais como paralelismo de dados ou de tarefas, ou ainda ambos conjuntamente) e oferece mecanismos de sincronização razoavelmente poderosos.

Para atingir os objetivos acima expostos, a linguagem C++ foi estendida com oito novas construções. Estas novas construções fornecem mecanismos para o usuário definir as tarefas que devem ser executadas em paralelo, sincronizá-las, controlar o acesso a dados compartilhados, definir a distribuição destas tarefas nos processadores que não possuem memória compartilhada e definir o modo de transferência de dados entre estes processadores.

Esta linguagem é de domínio público e pode ser obtida através de ftp anônimo para o diretório `CC++/docs/tutorial` no endereço `cs.caltech.edu`.

3.1.5 Regis

O sistema **Regis** [14] não é propriamente uma linguagem, como as apresentadas anteriormente. Ele é um ambiente para a construção de programas paralelos, que se baseia em paralelismo de tarefas. O modelo de programação consiste de componentes que realizam processamento, interconectados por objetos de comunicação. A instanciação destes componentes e a ligação entre eles é realizada através da linguagem de configuração Darwin.

A linguagem utilizada para a programação dos componentes e dos objetos de comunicação é C++, sendo oferecido um conjunto de classes C++ de modo a facilitar a construção destes elementos.

Este sistema é de domínio público e pode ser obtido através de ftp anônimo no endereço `gummo.doc.ic.ac.uk`.

3.1.6 SR

A linguagem **SR** [28] foi desenvolvida de modo a oferecer suporte para ambientes de programação diferentes. Ela é diferente das outras linguagens aqui descritas, pois não foi desenvolvida visando apenas oferecer extensões a linguagens seqüenciais. Foi desenvolvida para ser uma linguagem paralela com recursos para desenvolvimento de programas em máquinas multiprocessadas com memória compartilhada. Oferece operações para o compartilhamento de variáveis e também recursos para a implementação de programas em ambientes com memória distribuída, assim como operações de comunicação e facilidades para que os processos sejam executados em máquinas diferentes.

Sua implementação consiste de um compilador que traduz SR para C, um ligador que serve como frontal para o ligador `ld` da linguagem C e algumas bibliotecas. Uma análise comparativa entre SR e uma plataforma de programação funcional, com comentários sobre desempenho e facilidades de programação, pode ser encontrada em [17].

Esta linguagem é de domínio público e pode ser otida através de ftp anônimo para o diretório `pub/sr` no endereço `cs.arizona.edu`.

3.2 Bibliotecas para troca de mensagens

Uma maneira simples de se adicionar novas funcionalidades a uma linguagem de programação é através da criação de bibliotecas de *software*. O usuário executa chamadas às novas rotinas da biblioteca para processar as novas funções, sem que a linguagem original tenha que ser modificada. Devido à facilidade de implementação acima descrita, foram desenvolvidas várias ferramentas baseadas em bibliotecas de troca de mensagens para possibilitar o desenvolvimento de programas paralelos. Existe um número bastante grande de bibliotecas com esta finalidade [35, 37]. Algumas delas são descritas nesta seção, de modo a mostrar diferentes abordagens utilizadas para o seu desenvolvimento.

3.2.1 PVM

O sistema **PVM** (*Parallel Virtual Machine*) foi desenvolvido visando o desenvolvimento de programas paralelos e distribuídos a serem executados em um conjunto heterogêneo de máquinas interligadas [18].

O processamento paralelo é tratado pelo PVM como uma coleção de processos que se comunicam através de mensagens. O núcleo do modelo por ele utilizado é uma máquina paralela virtual. Esta máquina virtual é composta por um conjunto de máquinas físicas interligadas, utilizada pelo usuário como uma máquina paralela fracamente acoplada onde os programas paralelos são executados. Para a programação são oferecidas funções de envio e recebimento de mensagens, criação dinâmica de tarefas, criação de grupos e sincronização entre tarefas, entre outras. Estas funções foram desenvolvidas para as linguagens C e Fortran.

Este sistema é de domínio público e pode ser obtido através do endereço `http://www.netlib.org/pvm3/index.html`.

3.2.2 p4

O sistema **p4** [8, 9] oferece uma biblioteca de rotinas que podem ser executadas em sistemas com diferentes modelos de acesso à memòria. Para ambientes de memória distribuída, existem as funções para envio e recebimento de mensagens, para comunicação global e para sincronização. Para ambientes de memória compartilhada, são oferecidas primitivas para o uso de monitores.

Este sistema pode ser utilizado em ambientes compostos de máquinas heterogêneas, tanto em termos da arquitetura da memória, como do fabricante. Uma característica forte do p4 é o gerenciamento estático de tarefas, ou seja, todos os processos envolvidos na programação têm que ser criados quando o programa é iniciado, não sendo possível a criação de novos processos no decorrer da execução do programa.

Este sistema é de domínio público e pode ser obtido através de ftp anônimo para o diretório `pub/p4` no endereço `info.mcs.anl.gov`.

3.2.3 Express

O sistema **Express** [29] oferece dois modelos de programação: o *Cubix* e o *host-node*. No primeiro, a aplicação consiste de um único programa a ser executado em um ambiente de estações interligadas por rede. Existem processos do sistema para tratar pedidos de entrada/saída, de chamadas ao sistema e de utilização de processamento gráfico feitos por este programa. No segundo, o programa é dividido em duas partes, uma que vai ser executada por um processador frontal (*host*) e outra pelos nós (*nodes*). O processador frontal é responsável pelo gerenciamento do processamento do programa, fazendo a inicialização dos programas nos nós, recolhendo os dados gerados pelos nós e fazendo medições de tempo, entre outras tarefas. Os nós executam o processamento propriamente dito. Para este modelo, são oferecidas funções para envio e recebimento de mensagens e mecanismos de comunicação global. Para ambos modelos, existem funções para facilitar a manipulação de estruturas de dados, tais como estruturas do tipo grade. Algumas ferramentas para depuração de programas e paralelização automática de código estão disponíveis.

Este sistema é comercial e informações podem ser obtidas através do endereço `http://www.express.com`.

3.2.4 ISIS

O sistema **ISIS** [6, 7] foi desenvolvido de forma a fornecer ferramentas para garantir a construção de programas distribuídos tolerantes a falhas. Os processos que compõem a computação distribuída são organizados em grupos. Para um processo participar da computação, ele tem que se associar a um grupo já existente ou criar um grupo novo. Para cada grupo são fornecidas funções para sincronização entre processos, comunicação global para o grupo, transferência dos dados dos processos do grupo para um novo processo (no momento em que ele se incorpora ao grupo), reconfiguração dinâmica do sistema (baseado nos dados replicados no grupo) e monitoração do estado dos processos e da localidade dos dados.

Este sistema possui uma versão comercial e outra de domínio público. Maiores informações podem ser obtidas no endereço `http://www.cs.cornell.edu/Info/Projects/ISIS/ISIS.html`.

3.2.5 MPI

MPI [16, 26, 36] não é propriamente um sistema implementado, mas sim uma proposta de padronização para a interface de troca de mensagens para máquinas paralelas com memória distribuída. A idéia de se padronizar esta interface é a de possibilitar a portabilidade dos programas entre as diferentes máquinas e facilitar o seu uso.

O padrão MPI não contém nenhuma definição de suporte para tolerância a falhas e assume que o ambiente em que será executado é confiável. Ele define uma interface de troca de mensagens e não pretende definir um ambiente completo para programação paralela. Por esta razão, não são tratados tópicos como entrada/saída paralelas, estruturação do programa paralelo e depuração. Também não é definido nenhum suporte explícito para *multithreading*, apesar de um dos objetivos do projeto do padrão ser o de garantir que ele possa ser implementado de maneira eficiente em ambientes que possibilitem *multithreading*.

O núcleo do MPI é formado por rotinas para comunicação ponto a ponto entre pares de processos. Estas rotinas podem ser chamadas de forma bloqueante ou não bloqueante. Existem três modos de comunicação: *ready*, no qual a mensagem só pode ser enviada se uma operação de recepção correspondente já tiver sido iniciada; *standard*, no qual a mensagem pode ser enviada independentemente de haver uma operação de recepção para ela; e *synchronous*, que funciona de maneira semelhante ao modo *standard*, com a diferença de que a operação de envio não será considerada finalizada até que a operação de recepção correspondente seja iniciada no processo de destino.

Outras facilidades definidas são as rotinas para grupos de processos, que visam coordenar a comunicação entre os processos pertencentes a um determinado grupo. Estas rotinas permitem a transmissão coletiva dos dados e oferecem as seguintes funcionalidades (ver Figura 5):

- Envio de um mesmo dado por um determinado participante do grupo para todos os demais (*broadcast*): na figura 5.a, o processo p_0 envia o bloco de dados 0 para os processos $p_j, j = 0, 1, 2, 3$.
- Envio de dados diferentes por um determinado participante do grupo para todos os demais (*scatter*): na figura 5.b, o processo p_0 envia o bloco de dados j para os processos $p_j, j = 0, 1, 2, 3$.
- Envio de dados de todos os participantes do grupo para um determinado participante (*gather*): na figura 5.b, cada processo $p_i, i = 0, 1, 2, 3$, envia o bloco de dados 0 para o processo p_0.
- Envio de dados de cada participante para todos os demais (*allgather*): na figura 5.c, cada processo $p_i, i = 0, 1, 2, 3$, envia o bloco de dados 0 para os processos $p_j, j = 0, 1, 2, 3$.
- Envio de dados diferentes de cada participante para todos os demais (*alltoall*): Na figura 5.d, cada processo $p_i, i = 0, 1, 2, 3$, envia o bloco de dados j para o processo $p_j, j = 0, 1, 2, 3$.

Também são definidas operações para a computação de certas funções sobre dados pertencentes aos participantes de um determinado grupo. Estas operações são denominadas redução e varredura. Na operação de redução, uma função pré-definida ou definida pelo usuário é aplicada sobre os dados localizados nos diversos processadores que estão executando os processos pertencentes a um determinado grupo. O resultado desta função pode ser enviado a todos participantes do grupo ou a somente um deles. Um exemplo desta operação seria determinar o valor máximo de um conjunto de dados, onde cada elemento deste conjunto está associado a um processo pertencente a um determinado grupo. A operação de varredura executa o mesmo procedimento que a redução, mas sobre subconjuntos dos participantes do grupo.

Uma implementação deste padrão pode ser obtida através de ftp anônimo para o diretório `pub/mpi` no endereço `info.mcs.anl.gov`.

4 Análise comparativa

Como apresentado na seção anterior, existe um grande número de ferramentas para a implementação de programas paralelos em ambientes distribuídos. A escolha da ferramenta a ser utilizada é uma decisão que deve levar em conta vários fatores, tais como os ambientes de *hardware* em que ela pode ser utilizada, as facilidades que ela oferece para apoiar o desenvolvimento de programas, a adequação do paradigma por ela utilizado ao algoritmo a ser implementado e o desempenho do sistema.

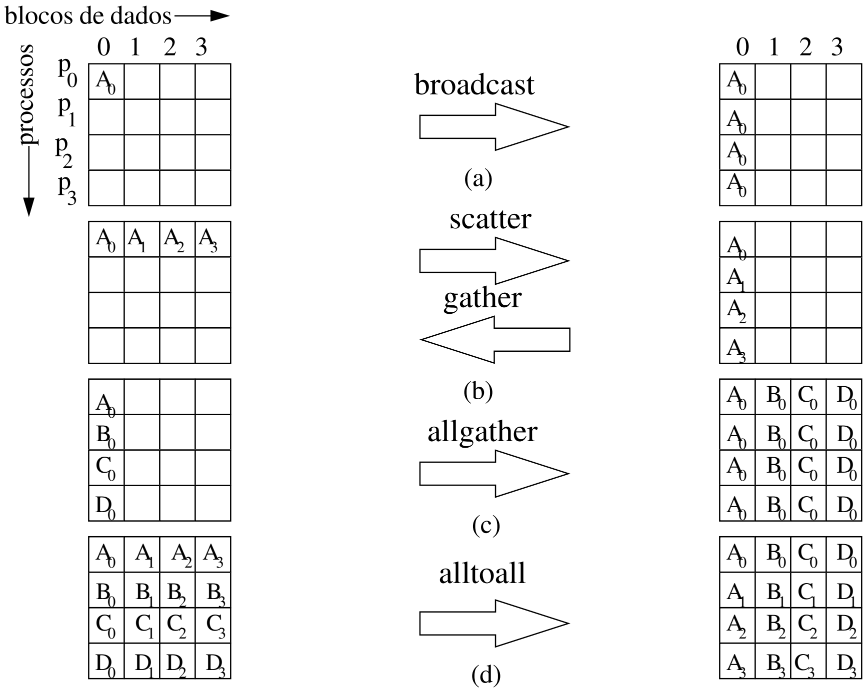

Figura 5: Rotinas do MPI para comunicação global [24]

Nesta seção, apresenta-se um estudo comparativo de algumas destas ferramentas. A maior parte dos estudos encontrados na literatura visava fazer uma comparação de desempenho das ferramentas em termos do tempo de execução de diferentes programas. Devido a esta fato, a maior parte dos resultados encontrados provém de ferramentas baseadas em bibliotecas de troca de mensagens, que em geral apresentam um melhor desempenho do que as linguagens paralelas.

Os dados mais significativos encontrados, em termos da quantidade de informações e da diversidade de testes realizados, foram obtidos para **Express**, **p4**, **PVM** e **Linda**. Deve-se observar que as três primeiras ferramentas utilizam bibliotecas de troca de mensagens, enquanto que a última foi desenvolvida como uma linguagem paralela, mas de uma maneira bem simples, de modo a atingir um desempenho competitivo com os sistemas de bibliotecas. Apresenta-se a seguir um estudo comparativo destas quatro ferramentas, em termos de programação, instalação e suporte, facilidade de depuração e desempenho.

4.1 Programação

Cada uma das ferramentas exibe certas particularidades no modo de utilização. Será apresentada uma breve descrição do uso de cada uma delas, conjuntamente com um exemplo de um programa simples baseado num modelo mestre-escravo e escrito utilizando-se cada uma das ferramentas. Este programa consiste de um processo mestre que inicia N processos escravos, os quais enviam a sua identificação para o processo mestre e encerram seu processamento local. O processo mestre recebe as identificações dos escravos e finaliza o processamento.

4.1.1 Express 3.2

Para se iniciar o sistema Express [29], é necessário executar o programa `exinit`, que carrega o *kernel* ou *daemon* e começa sua operação. Alguns parâmetros de inicialização podem ser definidos, tais como o número de *buffers* a serem alocados por cada nó do sistema e o tamanho de cada *buffer*. Estes parâmetros são definidos no arquivo `express.cst`.

Existem dois modelos de programação disponíveis: *host-node* e *Cubix*. No modelo *host-node* deve existir um programa para o computador frontal (*host*) e outro para os demais nós (*nodes*), os quais se comunicam através de troca de mensagens. Existem funções para difusão múltipla (*broadcast*) de mensagens e comandos globais. No modelo *Cubix* deve existir um único programa, que será executado pelos processadores através do comando `cubix`.

O sistema Express é uma biblioteca de rotinas, que pode ser implementada para qualquer linguagem de alto nível que disponha de um compilador. Atualmente, estão disponíveis bibliotecas para C e Fortran. A implementação do programa descrito no início da seção 4.1 pode ser vista nas Figuras 6 e 7.

4.1.2 p4-1.4

O gerenciamento de processos é feito de forma estática em p4 [8]. Para iniciar o sistema, o usuário necessita criar um arquivo chamado `procgroup`, que determina as máquinas que serão utilizadas na computação, quantos processos devem ser executados em cada uma delas, o arquivo executável que será utilizado por cada processo e qual a identificação de *login* a ser utilizada em cada máquina.

Várias funções são oferecidas ao usuário para o controle de envio e recebimento de mensagens. O usuário pode escolher entre envios bloqueantes ou não-bloqueantes, utilizar a conversão dos dados XDR (para possibilitar a troca de dados entre máquinas que utilizam diferentes formatos de dados) e escolher se o gerenciamento dos *buffers* utilizados para o armazenamento de mensagens será feito pelo p4 ou pela aplicação.

Também são oferecidas operações para que informações sejam passadas para todos os processos globalmente de uma única vez, sem que seja necessário fazer a comunicação par a par. Existe uma rotina de difusão múltipla (*broadcast*), que permite enviar de forma eficiente uma mensagem para todos os processos, utilizando o algoritmo *broadcast tree*. Uma rotina do tipo barreira faz com que os processos que a chamarem fiquem parados, até que todos os processos também chamem a mesma rotina. Ela é utilizada como mecanismo de sincronização entre os processos. Além destas rotinas, existe uma outra para permitir a execução de algumas operações sobre *arrays*, cujos elementos estejam distribuídos entre os processos. Estas operações podem ser de multiplicação, divisão, obtenção do valor máximo e obtenção do valor mínimo.

Para sistemas com memória compartilhada, são oferecidos alguns monitores específicos, que foram implementados e estão localizados na biblioteca do p4, além de funções para que o usuário construa o seu próprio monitor. A biblioteca p4 pode ser utilizada com as linguagens C ou Fortran. O programa exemplo escrito com a ferramenta p4 utilizando-se a linguagem C é mostrado na Figura 8.

```
/* Este e' o programa executado pelo no' mestre, iniciado da seguinte maneira:
 *              nome_do_programa numero_de_processos
 * Ele inicia o programa escravo no numero de nos fornecido como parametro de
 * entrada, ou assume como default o valor 4. Depois, espera que todos os nos
 * mandem a sua identificacao e finaliza o programa.
*/

#include <stdio.h>
#include "express.h"

main (int argc, char *argv[])
{
        long procid;  /* numero do processo recebido */
        int nodes;  /* numero de nos que participam do programa */
        int src=DONTCARE; /* indica que pode receber de qualquer processo */
        int type=123;  /* define o tipo de mensagem a ser recebida */
        int k, pgind;

/* Le o numero de nos da linha de comando, ou assume o valor default */
        nodes = (argc > 1) ? atoi (argv[1]) :  4;
/* Inicia o programa NODE nas estacoes */
        if (pgind=exrun(nodes, "node") < 0) {
                fprintf (stderr,  "Houve problemas na alocacao dos nos \n ");
                exit(1);
        }
/* Espera receber a identificacao de todos os nos */
         for (k=0; k<nodes; k++) {
                exread (procid, 4, &src, &type);
                fprintf (stdout,"\n Recebeu identificacao do processo %d \n",procid);
         }
         fprintf(stdout, "\n Acabou \n ");
}
```

Figura 6: Programa do computador frontal (*host*) utilizando Express

```
/* Este e' o programa que e' executado  pelos nos escravos. Como ele e' chamado
 * pelo nome "node" na chamada da rotina "exrun" no programa "host", o executavel
 * gerado por este programa tem que ter o nome "node".
*/

#include "express.h"

struct nodenv env;  /* contem informacoes sobre os  processadores
                       participantes da execucao */
main ()
{
        long procid;  /* numero do processo a ser enviado*/
        int dest;  /* indica o destino da mensagem */
        int type=123;  /* define o tipo de mensagem a ser enviada */

/* Le os parametros do sistema: numero do no em que o programa esta' sendo executado,
 * numero de nos participando da computacao e o numero do no' que e' o host
*/
        exparam(&env);
/* Coloca o identificador do no' no buffer de envio */
        procid=env.procnum;
/* Envia a identificacao do no' para o no' host */
        dest=env.host;
        exwrite (procid, 4, &dest, &type);
}
```

Figura 7: Programa para cada nó (*node*) utilizando Express

```
/* Este programa e' executado em todos os processadores, em modo mestre/escravo.
 * O mestre e' o processador no qual este programa foi chamado na linha de comando.
 * Ele cria instancias deste programa nos processadores escravos e espera receber a
 * identificacao de todos eles para finalizar o programa.
*/
#include <stdio.h>
#include "p4.h"
main (int argc, char *argv[])
{
        int procid;  /* numero do processo recebido */
        int nodes;  /* numero de nos que participam do programa */
        int src;  /* identifica de quem a mensagem deve ser recebida */
        int dest;  /* identifica para quem a mensagem deve ser enviada */
        int type;  /* define o tipo de mensagem a ser recebida e/ou
                      enviada */
        int ID,k;
/* Inicializa o p4 e cria os processos escravos.
 * A definicao das maquinas nas quais os processos escravos devem ser executados
 * e o nome do executavel, que deve ser processado por elas, sao parametros
 * definidos no arquivo "procgroup" associado a este programa e criado pelo usuario.
 * Estas instrucoes sao executadas apenas pelo programa mestre
*/
         p4_initenv(&argc, argv);
         p4_create_procgroup();
/*  Obtem numero do processo */
        ID = p4_get_my_id();
/* Se e' o processo mestre,  espera receber identificacao dos escravos */
        if (ID ==0) {
/* Obtem o numero de processos */
                nodes=p4_num_total_ids();
/* Espera receber identificacao dos processos escravos */
                for (k=0; k<nodes-2; k++) {
                        type=123;
/* Recebe de qualquer no */
                        src=-1;
                        p4_recv (&type, &src, &procid, &size);
                        fprintf (stdout,  "\n Recebeu identificacao do
                                 processo %d \n", procid); }
                fprintf(stdout, "\n Acabou \n "); }
        else {
/* Os processos escravos mandam a identificacao para o processo mestre */
                dest=0;
                p4_send(type, dest,ID,sizeof(int)); }
/* Espera que todos os processos acabem */
        p4_wait_for_end();
}
```

Figura 8: Programa utilizando p4

4.1.3 PVM 3.3.9

Os programas que utilizam o PVM são processados em uma máquina paralela virtual. Esta máquina virtual é composta por computadores interligados gerenciados por uma coleção de processos *daemons* chamados `pvmd` disparados por um mesmo usuário. Para que um computador possa fazer parte desta máquina virtual, é necessário que o processo *daemon* `pvmd` esteja sendo executado por ele. O *daemon* `pvmd` pode ser iniciado na console da máquina pelo usuário ou por um programa que já esteja sendo executado na máquina paralela virtual. Ou seja, computadores podem ser adicionados dinâmicamente à máquina virtual no decorrer da computação.

Para se iniciar uma máquina virtual utiliza-se o arquivo de configuração `hostfile`. Este arquivo contém a lista dos computadores que devem fazer parte da máquina virtual e parâmetros opcionais, tais como o diretório onde os arquivos executáveis podem ser encontrados ou o *login* que deve ser utilizado em uma determinada máquina. Informações detalhadas sobre a inicialização e utilização do PVM podem ser obtidas em [18]. Para que um programa possa ser executado nesta máquina virtual, é necessário que ele se registre com o *daemon* `pvmd` da estação correspondente, através da chamada de uma função da biblioteca PVM. Após ser registrado, ele pode utilizar as várias funções que o PVM oferece, tais como iniciar processos em outros computadores, verificar o estado dos processos que estão sendo executados na máquina virtual, enviar e receber mensagens de modo bloqueante ou não, enviar mensagens de difusão múltipla (*broadcast*), executar rotinas do tipo barreira e rotinas globais de comunicação de dados.

A biblioteca de funções PVM pode ser utilizada com as linguagens C e Fortran. Na Figura 9 é mostrado um exemplo de um programa PVM, utilizando-se a linguagem C.

4.1.4 Network Linda 3.1

O sistema Network Linda [31] utiliza um paradigma diferente, em relação às ferramentas descritas anteriormente. Neste sistema é utilizado o conceito de memória associativa compartilhada denominado de *tuplespace* (TS).

A comunicação neste sistema é feita através da operações para manipular registros de dados denominados tuplas neste espaço TS. Estas operações são extensões das linguagens C e Fortran. Um compilador desenvolvido para Linda é utilizado para a geração dos programas executáveis. São oferecidas instruções para criar novos processos e para colocar ou retirar as tuplas do TS com um padrão definido pela chamada da instrução.

O Linda é para ser utilizado em um ambiente de estações de trabalho interligadas por uma rede. O espaço de dados compartilhado é distribuído pela memória de cada estação. Para se processar um programa em Network Linda é necessário fazer sua chamada juntamente com o gerenciador de rede `ntsnet`. Este programa é responsável por examinar as opções fornecidas pela linha de comando e pelo arquivo de configuração (`.tsnet.config`), de modo a criar o TS distribuído e definir onde os processos irão ser executados, através da obtenção de informações sobre a carga de cada estação. Ele inicia os processos nas estações e destrói o TS distribuído quando todos os processos finalizam a sua execução.

Na Figura 10 se encontra um exemplo de programa em Network Linda, executando as mesmas funções que os anteriores e utilizando a linguagem C.

```
/* Este programa e' executado em todos processadores, iniciado da seguinte maneira:
 *                nome_do_programa numero_de_processos
 * O programa iniciado pela chamada na linha de comando cria novas instancias e espera
 * receber os identificadores dos processos para finalizar sua execucao. A quantidade
 * de novas instancias e' definida pelo parametro de entrada ou assume como default o
 * valor 4.
*/

#include <stdio.h>
#include "pvm3.h"

main (int argc, char *argv[])
{
        int procid;  /* identificacao do processo */
        int parentid;  /* identificacao do processo pai */
        int procs;  /* numero de processos que participam do programa */
        int info;  /* valor de retorno das chamadas de rotinas PVM */
        int filhos[MAXFILHOS];  /* contem os identificadores dos filhos */
        int src;  /* identifica de quem a mensagem deve ser recebida */
        int dest;  /* identifica para quem a mensagem deve ser enviada */
        int type;  /* define o tipo de mensagem a ser recebida e/ou enviada */
        int k;
/* Faz o registro na maquina virtual e obtem o identificador de processo */
        procid=pvm_mytid();
/* Obtem o identificador do pai do processo */
        parentid=pvm_parent();
/* Se o processo nao tem pai, entao ele e' o primeiro a ter sido executado
 * e deve iniciar as outras instancias do programa */
        if (parentid ==PvmNoParent) {
/* Le o numero de processos da linha de comando, ou assume o valor default */
                procs = (argc > 1) ? atoi (argv[1]) :  4;
/* Verifica se numero de processos e' legal */
                if ((procs ,  1) || (procs > MAXFILHOS)) {
                        pvm_exit();
                        exit();
        }
/* Inicia as outras instancias do  programa, deixando que o gerenciador
 * PVM escolha as maquinas em que irao ser criados os novos processos */
                info=pvm_spawn(argv[0],(char**)0,PvmTaskDefault,
                               (char*)0,procs,filhos);
/* Se nao conseguir iniciar nenhum processo, sai do programa */
                if (info == 0) {
                        fprintf (stderr,"Houve problemas na alocacao
                                 dos processos \n");
                        pvm_exit();
                        exit();
                }
```

Figura 9: Programa utilizando PVM (início)

```
/* Atualiza o numero de processos com o numero de processos que foram iniciados
 * corretamente */
                procs=info;
/* Espera informacao dos processos */
                for (k=0; k<procs; k++) {
                        type=123;
/* Recebe mensagem de qualquer processo */
                        pvm_recv(-1, type);
                 pvm_upkint(&procid,1,1);
                        fprintf (stdout,"\n Recebeu a identificacao do
                                 processo %d \n", procid);
                }
                pvm_exit();
                exit();
        }

/* No caso de ser um filho, envia a identificacao para o pai */
        else {
/* Indica que vai usar conversao XDR para transferencia dos dados */
                pvm_initsend(PvmDataDefault);
/* Coloca o identificador de processo no buffer de envio */
                pvm_pkint(&procid,1,1);
/* Envia a mensagem com o identificador do processo para o processo pai */
                type=123;
                pvm_send(parentid, type);
                pvm_exit();
        }
```

Figura 9: Programa utilizando PVM (continuação e fim)

4.1.5 Considerações sobre facilidade de programação

Linda é considerada por vários estudos comparativos [25, 27, 30, 34] como a ferramenta de programação mais fácil de ser aprendida e utilizada. Express foi considerada em [34] como a mais difícil, enquanto que PVM3.x e p4 foram consideradas satisfatórias. O PVM foi considerado em [25] como o mais difícil de ser utilizado, em comparação com p4 e Linda.

A razão pela qual normalmente Linda é considerada como a mais fácil de ser aprendida, é que ela possui apenas 8 novas instruções incorporadas às linguagens seqüenciais C e Fortran. Contrariamente, as demais ferramentas utilizam-se de um grande número de novas funções: da ordem de 50 para o PVM, da ordem de 60 para o p4 e, finalmente, da ordem de 130 para o Express.

Em todos estes estudos, é ressaltado o nível de adequação da ferramenta ao algoritmo a ser implementado. Por exemplo, Linda é mais fácil de se utilizar para a implementação de algoritmos baseados em estruturas de dados distribuídas. Por outro lado, programas que requerem operações globais, e que precisam ser executados em um ambiente constituído por estações com arquiteturas

```
/* A parte referente ao real_main e' executada pelo programa que e' chamado na
 * linha de comando. Para se iniciar o programa, ele deve ser executado da
 * seguinte maneira:
 *                       ntsnet nome_do_programa
 * O programa cria varios processos que executam a funcao ``hello'' e espera
 * receber os identificadores dos processos para finalizar. A quantidade de novas
 * instancias e' definida pelo parametro de entrada ou assume como default o valor 4
*/

real_main (int argc, char *argv[])
{
        int procs;  /* numero de processos a serem inicalizados */
        int num_procs;  /* numero de processos que podem ser inicializados */
        int procid;  /* identificador do processo */
        int hello();  /* funcao que sera' executada pelos processos
                           para enviar a identificacao da funcao */
        int j;
/* Le o numero de processos da linha de comando, ou assume o valor default */
        procs = (argc > 1) ? atoi (argv[1]) :  4;
/* Obtem numero de processos possiveis de serem inicializados */
        num_procs=lprocs();
/* Verifica se numero de processos e' legal
        if (procs > num_procs) {
                exit();
        }
/* Inicializa os processos que vao enviar o identificador de processo */
        for (j=0;  j<procs;  j++)
                eval ("worker", hello(j));
/* Recebe o identificador de cada processo */
        for (j=0;  j< procs;  j++) {
/* Procura uma tupla com o campo "done" no TS e coloca seu segundo campo
 * na variavel procid */
                in ("done",  ?procid);
                fprintf (stdout,"\n Recebeu a identificacao do
                         processo %d\n",procid);
        }
}
hello(i)
        int i;
{
/* Coloca uma tupla em TS com o campo identificador "done" e o identificador da
 * tarefa no segundo campo */
        out("done", i);
        exit ();
}
```

Figura 10: Programa utilizando Network Linda muito heterogêneas, podem ser implementados mais facilmente utilizando-se o PVM.

4.2 Instalação e Suporte

Mattson [25] considerou simples a instalação das ferramentas PVM, p4 e Linda. A parte mais difícil foi a elaboração dos arquivos de configuração da rede: o arquivo `procgroup` para o p4, o arquivo `hostfile` para o PVM e o arquivo `.tsnet.config` para Linda. A configuração deste último para o sistema Linda foi considerada como a tarefa mais difícil, devido ao fato deste sistema fazer um balanceamento dinâmico da carga dos processos, com base nos parâmetros do arquivo de configuração, que podem então afetar bastante o desempenho do programa. Deve-se observar que, dos quatro sistemas analisados, Linda é o único que oferece um mecanismo de balanceamento de carga que pode ser gerenciado pelo usuário. Sukup [34] considerou a configuração dos sistemas PVM 2.x, PVM 3.x e Linda bastante adequada. A configuração do Express foi considerada apenas satisfatória e a do p4 foi colocada entre a do Express e aquelas do PVM e do Linda.

Uma dificuldade na instalação do Linda apontada em [25, 30] é o gerenciador de licenças utilizado pela *Scientific Computing Associates* (SCA). As licenças são geradas utilizando-se o identificador de cada estação utilizada para a execução de uma determinada aplicação. Quando há uma mudança na rede, como por exemplo, a troca de uma máquina por outra, é necessário gerar uma nova licença que tem que ser fornecida pelo sistema de suporte da SCA. Na nova versão 3.1 este esquema foi simplificado, com a criação de um único processo servidor de licenças. Neste caso, só é necessária a criação de uma licença para a estação servidora de licenças e não mais para cada estação da rede.

Como Linda e Express são produtos comerciais, é natural que seu suporte seja bastante efetivo, conforme observado em [25, 27]. Segundo Mattson [25], apesar de ser um *software* de domínio público, o PVM possui um excelente suporte através de correio eletrônico. Outro fator importante para o suporte do PVM é a existência de um *newsgroup* bastante ativo, denominado `comp.parallel.pvm`. As outras ferramentas não possuem *newsgroups* ativos.

A documentação do PVM provê uma grande quantidade de informações para o usuário, tanto a nível de utilização do sistema, quanto a nível da forma como o PVM foi implementado. Um manual bastante atualizado para a versão atual (3.3.9) corrige deficiências apontadas em [25, 30]. Além do manual, um livro [18] oferece maiores informações sobre o sistema, inclusive sobre sua implementação. O manual do Linda fornece poucas informações em termos de instalação, mas os problemas encontrados são prontamente atendidos pela equipe de suporte da *Scientific Computing Associates*. O p4 possui um manual razoavelmente bom, com alguns exemplos interessantes. Quanto ao Express, um ponto interessante a ressaltar é o fato de toda sua documentação poder ser acessada livremente através do endereço `http://www.parasoft.com`.

4.3 Facilidades de depuração

Um ponto a ser levado em conta na depuração e nos testes de um programa é o tempo que o sistema precisa para se configurar antes de começar a executá-lo. Segundo [34], o PVM 3.x é a ferramenta que atingiu o menor tempo de inicialização, enquanto que Express e Linda apresentaram tempos intermediários e p4 os piores resultados. Outro ponto interessante, apontado no trabalho citado acima, é a facilidade de se depurar um programa sem a utilização de um depurador, ou seja, somente através das mensagens de erro fornecidas em tempo de execução. O p4 foi considerado como aquele que ofereceu maior facilidade, seguido pelo PVM 3.x e logo depois pelo Express e pelo Linda.

Linda possui a ferramenta Tuplescope para depuração de programas, que foi considerada bastante útil em [25, 27, 30]. Ela é processada em uma única estação de trabalho, utilizando o esquema multi-tarefa para simular o paralelismo do programa. A interface com o usuário é totalmente gráfica, sendo bastante útil para detecção de erros no programa. Para erros que necessitem que a execução do programa seja realmente realizada no ambiente de rede, Linda possibilita que os processos sejam executados utilizando-se depuradores, tais como o dbx.

O PVM oferece a ferramenta denonimada xpvm, que permite que um usuário monitore a execução de um programa paralelo em uma estação, através de uma interface gráfica. Estações podem ser incorporadas ou retiradas da máquina virtual e programas podem ser iniciados utilizando-se diretamente a interface gráfica. É fornecido um diagrama de tempo, útil para se fazer o acompanhamento do programa que está sendo executado na máquina virtual. Neste diagrama são mostrados o envio e o recebimento de mensagens entre as estações, assim como o que está sendo executado em cada estação em determinado instante. Além desta ferramenta, o PVM também possibilita a execução dos processos utilizando-se depuradores, como em Linda.

O p4 contém mecanismos de depuração de programas que foram considerados bastante úteis [34]. Ele oferece funções para acompanhar a execução do programa, através da impressão de informações sobre eventos que ocorrem nos processos. O usuário pode escolher diferentes níveis de acompanhamento. Quanto maior o nível, maior a quantidade de informações geradas. Por exemplo, se o nível escolhido for 10 (*default*), cada vez que um processo for criado ou uma mensagem for enviada ou recebida, uma mensagem indicando o número do processo seguido do evento que lá ocorreu (criação, envio ou recebimento) é enviada para o dispositivo padrão de saída. O pacote ALOG é distribuído junto com o p4 e fornece um conjunto de rotinas para instrumentar os programas de modo a registrar eventos definidos pelo usuário. Estes registros são colocados em arquivos e podem ser visualizados graficamente utilizando-se ferramenta Upshot[24].

O sistema Express apresenta a ferramenta NDB, que parece ser bastante similar ao depurador dbx [34]. Um problema encontrado foi o fato do depurador não apresentar automaticamente informações sobre o identificador do processo que está executando a instrução mostrada.

4.4 Análise de desempenho

Mattson [25] apresenta duas comparações entre os sistemas p4-1.3, PVM 3.2.6 e Linda 2.5.2, executadas em um ambiente formado por estações IBM RS/6000 modelo 560, interconectadas por uma rede Ethernet. Durante os experimentos, os arquivos eram compartilhados na rede através do NFS, mas a rede só era utilizada pelos programas de teste. A primeira comparação foi feita utilizando-se um programa chamado *ping-pong*, desenvolvido para medir o tempo de comunicação entre dois nós da rede. Este programa mede o tempo que transcorre para que uma mensagem seja enviada para um nó de destino, recebida, re-enviada pelo nó destino para o nó origem e, finalmente, recebida de volta pelo nó origem. Esta medida foi repetida 100 vezes para uma mensagem com um determinado tamanho. Na Figura 11 podem ser observadas as maiores taxas de transferência obtidas para quatro diferentes tamanhos de mensagem. Deve-se observar que, como a rede não é isolada pois há compartilhamento de arquivos, as medidas se referem à maior taxa de transferência obtida e não à taxa média nas 100 iterações.

Tamanho das mensagens (Kbytes)	**p4** (Mbytes/s)	**PVM** (Mbytes/s)	**Linda** (Mbytes/s)
0.8	0.35	0.28	0.23
8.0	0.88	0.85	0.48
80.0	1.07	1.00	0.70
800.0	1.10	1.00	0.73

Figura 11: Maiores taxas de transferência obtidas para o teste *ping-pong*

A segunda comparação foi feita utilizando-se um programa para enviar um *array* em um anel composto de quatro nós. Cópias do *array* são inicialmente colocadas nos quatro nós, que enviam sua cópia do *array* para o vizinho sucessor e recebem a cópia do vizinho predecessor. O processo de envio e recebimento desta cópia foi executado 100 vezes por cada processador e ao final o tempo gasto por cada um deles neste processo foi avaliado. Selecionou-se o menor tempo entre eles para se calcular a maior taxa de transferência obtida. Os resultados para diferentes tamanhos de *array* podem ser observados na Figura 12.

Tamanho das mensagens (Kbytes)	**p4** (Mbytes/s)	**PVM** (Mbytes/s)	**Linda** (Mbytes/s)
0.8	0.88	0.50	0.68
8.0	1.02	1.10	0.84
80.0	1.00	1.00	1.02
800.0	1.00	1.10	1.01

Figura 12: Maiores taxas de transferência obtidas para o teste do anel

Segundo Mattson [25], o pior desempenho do Linda na primeira comparação é possivelmente devido a dois fatores: gerenciamento do espaço de tuplas e protocolo utilizado para comunicação. A taxa de transferência obtida engloba desde o início do envio da mensagem até o seu recebimento. Nos sistemas p4 e PVM o usuário é responsável pela alocação de espaço para a mensagem, assim como por seu envio e recebimento. Em Linda, o espaço de tuplas é distribuído em cada memória dos processadores e esta distribuição é transparente para o usuário, causando um *overhead* para o sistema. Na implementação em Linda do programa *ping-pong* utilizado para esta comparação, o envio de uma mensagem é feito através da colocação de uma tupla no espaço de tuplas e o seu recebimento através da retirada desta mesma tupla. No envio de uma mensagem, Linda tem toda a responsabilidade de alocar espaço para esta tupla e definir em qual processador este espaço será alocado. Já no recebimento de uma mensagem, Linda tem que verificar onde a tupla está localizada e enviá-la para o processador que a requer. Este *overhead* pode ser diminuído com otimizações no compilador Linda, mas sempre existirá a necessidade de um processamento em tempo de execução, que pode levar à degradação das taxas de transferência. Como as tuplas estão distribuídas entre as memórias dos processadores, elas são transportadas de um processador para outro. Para realizar este transporte é utilizado o protocolo UDP. No experimento realizado, PVM e p4 utilizam o protocolo TCP para envio e recebimento de mensagens. Esta também pode ser uma razão do pior desempenho do Linda, porque a utilização do protocolo TCP pode levar a maiores taxas de transferência na rede.

Na segunda comparação, os desempenhos dos sistemas convergiram para um mesmo valor quando as mensagens se tornaram maiores. Este fato era esperado e pode ser explicado pela saturação da rede ocorrida devido ao tamanho das mensagens, que fez com que a rede se tornasse o principal gargalo de comunicação. Os resultados obtidos para o PVM foram considerados menos precisos devido à instabilidade deste sistema no ambiente testado. Algumas vezes os programas fucionavam satisfatoriamente, outras não.

Estes testes representam os piores casos para as ferramentas, porque não há nenhum processamento sendo feito nos nós que permita diluir o custo de comunicação. Geralmente, o projeto de aplicações, que serão executadas em ambiente de rede, visa gastar menos tempo em comunicação e mais tempo em processamento executado por cada máquina. Por isso, é interessante comparar-se o tempo de execução obtido pelas ferramentas em aplicações específicas.

Em Carriero et al. [12] consta uma comparação entre as ferramentas Linda 2.5.0 e PVM 2.4.0. A aplicação implementada foi a resolução da equação de condução de calor em uma malha bi-dimensional. A avaliação dos tempos de computação em função do número de estações pode ser observada na Figura 13. Estes resultados mostram desempenhos bastante semelhantes para as duas ferramentas, em termos de tempo de execução.

Número de estações	**PVM** (segundos)	**Linda** (segundos)
1	1370.6	1370.6
2	648.0	662.2
4	328.0	342.6
8	168.4	175.8
16	90.0	92.8
32	54.0	54.0
38	54.0	51.3

Figura 13: Tempos de execução obtidos para resolução da equação de condução de calor

Sukup [34] implementou quatro programas para a solução de problemas que fazem parte do *benchmark* paralelo *Numerical Aerodynamic Simulation* (NAS) [4], utilizando as ferramentas Express, p4, PVM e Linda.

- **EP:** um programa trivialmente paralelo para a geração de um conjunto de pares de números estatisticamente independentes, cujas distribuições de probabilidade são gaussianas de média zero e variância um.

- **MG**: um programa simples para um *multigrid* 3D para a solução aproximada do problema discreto de Poisson $\nabla^2 u = v$ em uma malha $n \times n \times n$.

- **CG**: um programa para resolução de sistemas lineares não estruturados esparsos pelo método de gradiente conjugado para estimação do menor autovalor de uma matriz esparsa simétrica definida positiva, contendo um padrão aleatório de elementos não nulos.

- **IS**: um programa para definição da posição de uma chave em ordem crescente, dentro de um conjunto de chaves geradas por um algoritmo seqüencial.

Cada um destes programas foi executado várias vezes e as médias dos tempos de execução foram calculadas. Os tempos de processamento verificados na solução destes problemas pelas quatro ferramentas são apresentados na Figura 14.

Programa	**PVM 2.x** (segundos)	**PVM 3.x** (segundos)	**p4** (segundos)	**Express** (segundos)	**Linda** (segundos)
EP	584	586	589	586	586
MG	23	35	24	24	45
CG	442	685	623	277	322
IS	175	208	277	175	179

Figura 14: Tempos médios de execução em segundos

O programa EP gera inicialmente um conjunto de N números pseudo-aleatórios com distribuição uniforme no intervalo $(-1, 1)$. A partir deste conjunto são gerados pares de números estatisticamente independentes com distribuições gaussianas. Este problema é considerado trivialmente paralelo, porque é extremamente simples criar um algoritmo paralelo para ele. Basta gerar seqüencialmente os N números e distribuir N/p números pelos p processadores. Cada um deles criará um conjunto de pares de números com distribuição gaussiana, a partir dos N/p números iniciais. Como este problema não requer praticamente nenhuma comunicação entre os processos, os resultados não diferem muito entre as diferentes ferramentas.

O programa MG é o que requer maior comunicação entre os processadores. Além disso, ele é o único que requer comunicação ponto-a-ponto. Estes fatores podem explicar o pior desempenho do Linda, devido ao *overhead* de gerenciamento do espaço de tuplas explicado anteriormente.

O programa CG utiliza-se basicamente do envio de mensagens de um processo para todos os outros (*broadcast*). Neste caso, o PVM 3.x e o p4 não parecem ser eficientes, provavelmente devido ao modo de implementação utilizado para o *broadcast* de mensagens. Express e Linda apresentam os melhores desempenhos, que foram atingidos através do teste de várias combinações de parâmetros referentes a cada uma delas. Para o Express foram testados os valores de tamanho e número de *buffers* para recepção de mensagens, de modo a fornecer um melhor desempenho. Os parâmetros para execução do Linda, tais como a utilização de otimização de *broadcast* e redirecionamento das tuplas, foram ajustados para permitir um melhor desempenho do programa.

O programa IS foi implementado com um número menor de chaves do que o programa do *benchmark* NAS original, devido à quantidade reduzida de memória no ambiente de processamento. O algoritmo é definido de tal modo que isto não afeta a quantidade de comunicação necessária para a resolução do problema, mas a computação feita por cada nó sofre uma redução. Neste caso, o p4 apresentou o pior desempenho, enquanto que PVM 2.x, Express e Linda apresentaram desempenhos semelhantes e o do PVM 3.x ficou entre o do p4 e o dos demais. Para Express e Linda foram executados ajustes nos parâmetros, do mesmo modo que para o programa anterior.

As ferramentas PVM 2.x, PVM 3.x e Linda se mostraram bastante estáveis durante toda a duração dos testes dos quatro programas. Na execução dos programas utilizando-se as ferramentas p4 e Express ocorreram alguns problemas, que fizeram com que os programas tivessem a sua execução interrompida de maneira anormal. Os responsáveis pelo suporte destas ferramentas resolveram estes problemas de modo a possibilitar a execução completa dos programas.

Müller [27] desenvolveu um estudo comparativo envolvendo Express 3.2 e Linda 2.4.6. Foram implementadas três aplicações:

- Resolução de sistemas lineares de ordem n através da eliminação gaussiana
- Transformada discreta de Fourier e sua inversa
- Problema do caixeiro viajante

O ambiente de *hardware*, onde estas aplicações foram implementadas, era constituído de quatro estações IBM RISC System/6000 Modelo 320H interligadas por uma rede Ethernet e estava dedicado para processamento exclusivo das aplicacações de teste. Para o problema da eliminação gaussiana, foram feitos testes para $n = 100$ que se mostraram muito ineficientes, devido à grande quantidade de comunicação exigida. Na Figura 15 se encontram os resultados obtidos para a resolução de um sistema de equações de ordem $n = 1000$.

Número de estações	**Express 3.2.5** MFLOPS	**PVM 2.4.1** MFLOPS	**Linda 2.4.6** MFLOPS
1	10.6	10.6	10.7
2	13.9	16.3	17.4
3	16.6	19.3	21.7
4	19.6	19.7	24.1

Figura 15: Eliminação gaussiana paralela com $n = 1000$

Linda apresenta um desempenho superior que as outras duas ferramentas. Segundo Müller [27], isto pode ser explicado pela natureza do problema implementado. Esta aplicação apresenta um comportamento complexo em relação às necessidades de comunicação e às necessidades de processamento. Por exemplo, o tempo total de execução para a solução de um sistema de três equações utilizando-se Linda foi de 34 segundos, sendo que cada processo gastou 8 segundos na fase de comunicação. Este exemplo mostra que esta aplicação apresenta um nível relativamente alto de sobreposição de processamento nas estações e de comunicação entre elas. Além disso, a implementação utilizando Express demonstrou uma exigência alta em relação à quantidade de memória utilizada no *buffer* de recepção. Isto também pode ser explicado pelo fato dos níveis de comunicação e processamento serem semelhantes. Como a estação não pode ficar dedicada a tratar as mensagens recebidas, pois ela tem que fazer um certo processamento, as mensagens vão se acumulando nos *buffers*. Apesar de não haver muitas informações sobre a estratégia interna utilizada pelo Linda, presume-se que o melhor desempenho atingido se deva ao modelo cliente-servidor por ela utilizado. Este modelo permite que o receptor de informação influencie nos envios de mensagens efetuados por uma estação, ao contrário do modelo de troca de mensagens, onde a mensagem sempre é enviada ao estar pronta, independentemente do fato de seu receptor necessitá-la naquele instante ou não. Isto permite que o meio de comunicação seja melhor utilizado, porque a informação pode circular no meio somente quando um determindo receptor a necessite. Além disso, evita retransmissões através do meio devido à saturação do *buffer* do receptor, que pode ocorrer pelo fato de ele não estar pronto para receber uma mensagem.

No caso da transformada discreta de Fourier e de sua inversa, foi processada uma implementação paralela onde o número de termos sempre é uma potência de 2. Na Figura 16 são apresentados os resultados para as ferramentas Express e Linda para 2 e 4 estações. Esta aplicação requer muita comunicação entre os processos, o que justifica o melhor desempenho da ferramenta Express sobre Linda. Isto também explica o pior desempenho de ambas as ferramentas para valores pequenos de n, porque neste caso o *overhead* de comunicação sobre o processamento é maior. Estes resultados mostram que esta aplicação não pode ser paralelizada de maneira eficiente em um ambiente de estações conectadas por uma rede, devido às taxas elevadas de comunicação requeridas pela sua implementação.

Figura 16: Comparação entre Express e Linda para cálculo da transformada de Fourier e de sua inversa [25]

Na Figura 17 são mostrados os resultados obtidos para o problema do caixeiro viajante [13]. Pode-se observar um melhor desempenho da ferramenta Express em termos de processamento, embora o comportamento das duas ferramentas em relação ao número de nós do problema seja bastante semelhante. O melhor desempenho de Express pode ser creditado ao modelo de comunicação utilizado, que permite uma maior redução dos custos de comunicação do que aquele utilizado por Linda.

Finalmente, pode-se citar ainda o trabalho de Aiex, Martins, Ribeiro e Rodriguez [1], no qual é feita uma comparação detalhada entre o PVM 3.3.9 e o Network Linda 3.1. A aplicação empregada para a comparação envolve o uso de estratégias assíncronas da paralelização da metaheurística de busca tabu [19, 20] para a solução de um problema de particionamento de grafos no contexto de testes lógicos pseudo-exaustivos de circuitos VLSI [2].

Figura 17: Comparação entre Express e Linda para o problema do caixeiro viajante [25]

5 Conclusões

Neste trabalho foi apresentado um panorama das ferramentas atualmente disponíveis para programação paralela em ambientes de memória distribuída. O objetivo desta apresentação foi mostrar quais os tipos de ferramentas disponíveis e fazer uma comparação entre elas.

Foram mostradas duas abordagens: utilização de linguagens paralelas e utilização de bibliotecas para troca de mensagens. A primeira oferece uma interface de mais alto nível para o programador, mas apresenta maiores problemas em termos de desempenho e flexibilidade de programação. Já a segunda pode oferecer melhor desempenho e flexibilidade, mas oferece uma interface menos amigável para o programador, podendo tornar mais difícil o desenvolvimento dos programas.

Atualmente existem vários projetos [10, 16, 32] para o desenvolvimento de compiladores paralelos eficientes e de ferramentas para conversão automática de programas seqüenciais em paralelos. Com o advento do padrão MPI, é provável que todas as ferramentas baseadas em bibliotecas para troca de mensagens ofereçam as funcionalidades existentes no padrão.

É necessário ressaltar, mais uma vez, que a escolha da ferramenta a ser utilizada vai depender de vários fatores, tais como sua adequação ao algoritmo utilizado, o ambiente em que vai ser implementado o programa e a disponibilidade da ferramenta (algumas são de domínio público e outras são comerciais).

Para exemplificar, Linda é mais adequada para aplicações que apresentam um maior equilíbrio entre os níveis de processamento em cada estação e de comunicação entre as estações do que o PVM. Já o PVM e o Express são indicados para aplicações que requerem um maior nível de comunicação entre as estações. Linda e Express são produtos comerciais e portanto envolvem um custo para a sua aquisição, mas por outro lado apresentam uma alta qualidade de suporte técnico. O PVM, p4 e a implementação MPICH do MPI são distribuídos gratuitamente, mas o suporte e atendimento ao usuário parecem não ser tão eficientes quanto aqueles das ferramentas comerciais.

Referências

[1] R.M. Aiex, S.L. Martins, C.C Ribeiro e N. de La R. Rodriguez, "Evaluation of Distributed Programming Environments in the Framework of the Parallelization of Heuristic Search Procedures for Combinatorial Problems", em preparação, 1995.

[2] A.A. Andreatta e C.C. Ribeiro, "A Graph Partitioning Heuristic for the Parallel Pseudo-exhaustive Logical Test of VLSI Combinational Circuits", *Annals of Operations Research* 50 (1995), 1–36.

[3] Applied Parallel Research, *Home page da APR*, 1995. (disponível através do WWW em `http://www.infomall.org/apri`)

[4] D. Bailey, J. Barton, T. Lasinski e H. Simon, "The NAS Parallel Benchmarks", Technical Report RNR-91-002 Revision 2, NASA Ames Research Center, 1991.

[5] H.E. Bal, J.G. Steiner e A.S. Tanenbaum, "Programming Languages for Distributed Computing Systems", *ACM Computing Surveys* 21 (1989), 261–322.

[6] K.P. Birman, "The Process Group Approach to Reliable Distributed Computing", *Communications of the ACM* 36 (1993), 37–53.

[7] K.P. Birman e T.A. Joseph, "Exploiting Virtual Synchrony in Distributed Systems", *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 1987, 123–138.

[8] R.M. Butler e E.L. Lusk, "Monitors, Messages, and Clusters: The p4 Parallel Programming System", *Parallel Computing* 20 (1994), 547–564.

[9] R.M. Butler e E.L. Lusk, "User's Guide to the p4 Parallel Programming System", Technical Report ANL-92/17, Argonne National Laboratory, 1994. (disponível através de ftp anônimo no endereço `info.mcs.anl.gov:pub/p4`)

[10] California Institute of Technology, *CC++ Tutorial*, 1994. (disponível através de ftp anônimo no endereço `cs.caltech.edu no diretório CC++/docs/tutorial` e através do WWW em `http://www.compbio.caltech.edu/ccpp/`)

[11] N.J. Carriero, D. Gelernter e T.G. Mattson, "Linda in Context", *Communications of the ACM* 32 (1989), 444–458.

[12] N.J. Carriero, D. Gelernter e T.G Mattson, "The Linda Alternative to Message-Passing Systems", *Parallel Computing* 20 (1994), 633–655.

[13] T.H Cormen, C.E. Leiserson e R.L. Rivest, *Introduction to Algorithms*, MIT Press, 1990, Cambridge.

[14] S. Crane, J. Magee, N. Dulay e K. Twidle, *Regis Programmer's Manual*, 1994. (disponível através de ftp anônimo no endereço `gummo.doc.ic.ac.uk` no arquivo `pub/regis/regis-0.6.1.ps.gz`)

[15] D. Elias, "Introduction to Parallel Programming Concepts", *Workshop on Parallel Programming on the IBM SP*, Cornell Theory Center, 1995. (disponível através do WWW em `http://www.tc.cornell.edu/Edu/Workshop`)

[16] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995, Reading. (disponível através do WWW em `http://www.mcs.anl.gov/dbpp`)

[17] V. Freeh, "A Comparison of Implicit and Explicit Parallel Programming", *Journal of Parallel and Distributed Computing* 34 (1996), 50–65.

[18] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek e V. Sunderman, *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994, Cambridge. (disponível através do WWW em `http://www.netlib.org/pvm3/book/pvm-book.html`)

[19] F. Glover, "Tabu Search – Part I", *ORSA Journal on Computing* 1 (1989), 190–206.

[20] F. Glover, "Tabu Search – Part II", *ORSA Journal on Computing* 2 (1990), 4–32.

[21] High Performance Fortran Forum, *Home page do Forum para padronização do HPF*, 1995. (disponível através do WWW em `http://www.erc.msstate.edu/hpff/home.html`)

[22] Indiana University, *Home page do projeto pC++*, 1995. (disponível através do WWW em `http://www.extreme.indiana.edu/sage`)

[23] V. Kumar, A. Grama, A. Gupta e G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms*, Benjamin/Cummings, 1994, Redwood City.

[24] E. Lusk e V. Herrarte, "Studying parallel program behavior with upshot", Argonne National Laboratory, Technical Report ANL-91/15, 1991.

[25] T.G. Mattson, "Programming Environments for Parallel and Distributed Computing: A Comparison of p4, PVM, Linda and TCGMSG", *The International Journal of Supercomputing Applications and High Performance Computing* 9 (1995), 138-161.

[26] MPI Forum, "A Message-Passing Interface Standard", 1995. (disponível através do WWW em `http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html`)

[27] J. Müller, "Parallelverarbeitung auf Workstation-Clustern mittels Express und Network-Linda", Technical Report KFA-ZAM-JUL-2819, Forschnungszentrum Julich GmbH, 1993.

[28] R.A. Olsson, G.R. Andrews, M.H. Coffin e G.M. Townsend, "A Language for Parallel and Distributed Programming", Technical Report TR 92-09, University of Arizona, 1992.

[29] Parasoft Corporation, *Express-Building Parallel and Distributed Programs - Version 3.2*, 1992. (disponível através do WWW em `http://www.parasoft.com/express.html`)

[30] A. Petrie e R. Kerr, "A Qualitative Comparison of Network Linda and PVM", Technical Report Parallel Processing Memoranda series, University of Newcastle upon Tyne, 1994.

[31] Scientific Computing Associates Inc., *Network Linda User's Guide*, 1995.

[32] SCAI-GMD, "Página WWW com uma introdução às atividades do Institut für Algorithmen und Wissenschaftliches Rechnen (SCAI) do GMD - Forschungszentrum Informationstechnik GmbH", 1995. (disponível através do WWW em `http://www.gmd.de/SCAI/ParComp.html`)

[33] P.H. Smith, "System Software and Tools for High Performance Computing Environments", 1992. (disponível através do WWW em `http://www.ccsf.caltech.edu/PSTP.html`)

[34] F. Sukup, "Efficiency Evaluation of Some Parallelization Tools on a Workstation Cluster Using the NAS Parallel Benchmarks", Technical Report ACPC/TR 94-2, Vienna University of Technology, 1994.

[35] L.H. Turcotte, "A Survey of Software Environments for Exploiting Networked Computing Resources", Technical Report MSU-EIRS-ERC-93-2, Engineering Research Center for Computational Field Simulation, 1993.

[36] D.W. Walker, "The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers", *Parallel Computing* 20 (1994), 657–673.

[37] J. Wang, *Jonathan Wang's Bookshelf on Parallel Computing*, 1995. (disponível através do WWW em `http://www.cs.yale.edu/HTML/YALE/CS/HyPlans/jwang/parallel.html`, com várias referências sobre programação paralela e distribuída)