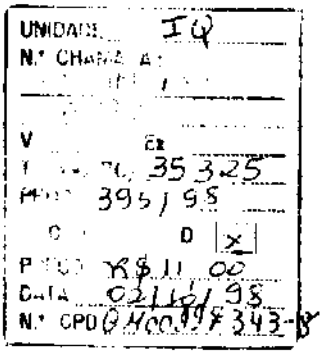
Técnicas de Visualização Científica e Computação Distribuída em Química
Uma Contribuição à Química Computacional

Pedro Antonio Muniz Vazquez
Orientador: Prof. Dr. Yoshiyuki Hase
Instituto de Química
Universidade Estadual de Campinas

Abril de 1998


UNIDADE IQ
V Ex
35325
395/95
D X
R$ 11,00
02/10/95

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO INSTITUTO DE QUÍMICA UNICAMP

M925t

Muniz Vazquez, Pedro Antonio

Técnicas de visualização cientifica e computação distribuida em química : uma contribuição à química computacional / Pedro Antonio Muniz Vazquez -- Campinas, [SP : s.n.], 1998.

Orientador: Yoshiyuki Hase.

Tese (doutorado) – Universidade Estadual de Campinas. Instituto de Química.

1. * Paralelismo. 2. Computação gráfica. 3. Unix. I. Hase, Yoshiyuki. II. Universidade Estadual de Campinas. Instiituto de Química. III. Títulom

Resumo

A aplicação de computação em rede em química foi estudada visando a visualização científica e a computação distribuida. Uma biblioteca de visualização científica capaz de produzir imagens tridimensionais de alta qualidade foi desenvolvida utilizando padrões abertos disponíveis no sistema operacional Unix. Esta biblioteca possui capacidades para realizar remoção de superfícies e linhas escondidas, cálculos de iluminação utilizando diversos modelos e um buffer para realização de efeitos de transparência. A partir desta biblioteca um pequeno conjunto de funções e objetos de interesse químico foram desenvolvidos para aplicação em visualização molecular. A biblioteca é capaz de explorar um ambiente de rede utilizando o protocolo X para display remoto sobre o protocolo TCP/IP.

Vários métodos de comunicação interprocessos foram investigados, técnicas baseadas em sockets BSD e memória compartilhada foram utilizadas para reescrever programas visando a execução distribuida. O desempenho relativo de mecanismos tradicionais de I/O e daqueles baseados em redes foram medidos e comparados assim como as técnicas baseadas em sockets e em memória compartilhada. Foram empregados padrões e bibliotecas publicamente disponíveis para o desenvolvimento das aplicações de realização das medidas.

Abstract

The application of network computing in chemistry was investigated aiming scientifc visualization and distributed computing. For scientific visualization a graphics library capable of high quality 3D rendering was developed using open standards available under the Unix operating system. This library features a depth buffer for hidden line and surface removal, light calculation for several ilumination models and an alpha buffer for transparency effects. On top of this library a small set of chemically interesting objects and functions were writen and used to develop application programs for molecular visualization. The library is able to run on networked environments using the X Window protocol for remote display using the TCP/IP protocol as the transport.

Several methods of interprocess comunication were investigated, BSD sockets and shared memory techniques were used to rewrite sequential programs for distributed execution. The relative performance of standard I/O mechanisms and networked methods were measured and compared in the same manner as the socket and shared memory techiques. Available publiclibraries and standards were used to develop applications and the measurements.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Introdução

Algumas áreas da química tem sido grandemente impulsionadas nas últimas décadas pelos avanços tecnológicos alcançados pela indústria eletrônica. Pode-se destacar as áreas de intrumentação e química computacional como aquelas que maiores benefícios tem recebido da eletrônica digital. Com os processadores dobrando de velocidade e as memórias de capacidade a cada nova geração novas fronteiras nos cálculos de propriedades moleculares foram estabelecidas e novos desafios de pesquisa encontrados. Observa-se, no entanto, que embora a tecnologia computacional tenha experimentado um grande avanço, as técnicas da química computacional, em sua grande maioria, ainda são as mesmas da década de 70, não explorando de forma eficiente os recursos atualmente disponíveis e limitando, desta forma, a dimensão dos estudos. A motivação e o objetivo deste trabalho é investigar técnicas de programação em química computacional que permitam expandir estes limites. Uma rápida análise da evolução da computação permite situar melhor o atual estágio tecnológico.

A história moderna da computação pode ser dividida em três eras [1]

- 1970 Computação centralizada (um computador com muitos usuários)
- 1980 Computação pessoal (um computador para cada usuário)
- 1990 Computação distribuída (muitos computadores compartilhados entre muitos usuários)

Até o início da década de 80 os computadores eram equipamentos de grande porte, caros e complexos e por isso localizavam-se em centros de processamento de dados (CPD's) que possuiam toda a infraestrutura e a equipe

técnica necessária à sua operação. Os usuários acessavam estas máquinas através terminais remotos de onde era feita toda a preparação dos dados de entrada, comandada a execução de programas e a análise dos resultados.

1.2 A Revolução dos Microcomputadores

A partir de 1977, aproximadamente, a tecnologia dos microprocessadores de 8 bits mostrou-se madura o suficiente para que os primeiros modelos de microcomputadores pessoais chegassem ao mercado dando início ao que na década de 80 seria conhecido como computação pessoal, microcomputação ou *Desktop Computing*. Embora alguns equipamentos deste período tenham alcançado um grande sucesso e colaborado de forma significativa para consolidar este modelo computacional (em particular o Apple II e o Sinclair ZX81), pode-se afirmar que foi somente a partir do lançamento do **PC** (*Personal Computer*) em 1981 pela IBM que o modelo de computação central da década anterior começou a ceder espaço à computação pessoal. Ao tornar público o projeto relativamente conservador de um microcomputador de 16 bits, com uma capacidade de memória até 10 vezes maior que os seus predecessores de 8 bits, capaz de executar tarefas de médio porte antes reservadas a mini-computadores e dotado de capacidade de expansão pela simples adição de cartões, a IBM estava estabelecendo um dos padrões industriais mais bem sucedidos da história da computação. Uma evidência disto é a capacidade demonstrada por este projeto de poder acompanhar os sucessivos avanços de hardware ocorridos em duas décadas ao passo que arquiteturas mais sofisticadas como a arquitetura VAX da Digital Equipment Corporation simplesmente sucumbiram ao aparecimento das arquiteturas RISC.

As transformações mais significativas da filosofia computacional associadas a essa década podem ser resumidas em três pontos [2]:

- A divisão de tarefas entre as máquinas centrais e os microcomputadores aliviando as primeiras dos trabalhos de natureza interativa e concentrando o seu trabalho em processamento de grandes volumes de dados;
- A sofisticação das interfaces máquina-usuário através do uso de recursos gráficos, exemplos típicos são os processadores de texto, as planilhas de cálculo e os sistemas de janelas;
- A especialização do usuário que não mais resume-se ao conhecimento de um pequeno conjunto de comandos, linguagem de programação ou aplicações. É exigido também o conhecimento de conceitos e procedimentos básicos de sistemas operacionais e sua administração, a configuração e instalação de produtos e periféricos (impressoras, compiladores)

e a realização de cópias de segurança (*backups*), tarefas estas que antes eram responsabilidade das equipes dos CPD's.

Embora estes equipamentos tenham trazido à mesa do pesquisador o ambiente computacional e as ferramentas necessárias ao uso e ao desenvolvimento de programas, as tarefas de maior porte, em termos de velocidade de processamento, espaço em disco, ou uso de memória continuaram a ser realizadas nos CPD's e, em alguns casos, nos centros de supercomputação, emergentes desde o final da década de 70 com o lançamento da série Cray de supercomputadores.

1.3 Estações de Trabalho: A Rede é o computador

Foi em torno destes centros de supercomputação que começou a nucleação tecnológica que viria definir o atual cenário da computação científica de alto desempenho. Sendo equipamentos de alto custo, os supercomputadores não podem ter seu desempenho degradado pelas etapas de preparação e análise de dados [3]. Da mesma forma, embora possível, não é computacionalmente eficiente dotar estas máquinas dos recursos gráficos avançados necessários para a geração e visualização dos resultados obtidos durante ou após tarefas de longa duração. A solução adotada foi o desenvolvimento de máquinas microprocessadas, de alto desempenho em relação aos microcomputadores e dotadas dos recursos necessários para a estas tarefas e conectadas em rede ao computador central através de um meio físico de alta velocidade. A estes equipamentos deu-se o nome de estações de trabalho.

A conexão de um grande número de estações de trabalho utilizando o protocolo Internet de comunicação (IP) e executando o sistema operacional Unix, mesmo na ausência de um supercomputador, acabou por mostrar-se muito mais eficiente computacionalmente que o modelo anterior (máquina central + microcomputadores) e originou a atual era da *Computação em Rede* ou (*Network Computing*).

O sucesso deste novo modelo computacional resultou, como no caso dos microcomputadores IBM-PC, da utilização de padrões bem estabelecidos e conhecidos: o sistema operacional Unix, versão Berkeley, o protocolo Internet e o sistema de janelas X do MIT (*MIT X Window System*) [1].

Embora existam diversos sistemas operacionais capazes de realizar tão bem ou melhor as mesmas tarefas que o Unix em uma rede de estações de trabalho estes sistemas caracterizam-se por terem sido desenvolvidos e otimizados para uma arquitetura específica de hardware, muitas vezes em lingua-

gem de baixo nível, o que torna difícil, senão impossível, a sua transferência para outras arquiteturas de hardware. O Unix por sua vez é extremamente modularizado e escrito em linguagem C tendo todas as dependências de hardware localizadas em um único programa, o *kernel*. Uma vez feita a adaptação deste para uma nova arquitetura de hardware basta recompilar o restante do sistema operacional e os programas e aplicações dos usuários.

As mesmas considerações em termos de adaptabilidade podem ser feitas com relação ao protocolo TCP/IP e ao X. Esta combinação de sistema operacional, protocolo de comunicações e sistema de janelas, com interfaces de programação bem definidas e públicas originou a expressão *Sistemas Abertos* (*Open Systems*) em oposição aos produtos de funcionalidade semelhante oferecidos por alguns fabricantes (DEC, IBM, Microsoft) mas de especificações nem sempre de conhecimento público. Em resumo: adotando estes três padrões tanto o fabricante de estações de trabalho quanto o consumidor final tem certeza de que poderão conectar-se aos equipamentos já existentes em uma rede. A isto denomina-se *interoperabilidade*.

Uma vez definidas as características técnicas básicas de uma rede Unix cabe, neste ponto, caracterizar em que consiste a computação em rede e como ela pode ampliar a capacidade computacional de um conjunto de computadores [4].

A computação em rede busca, através do uso de técnicas apropriadas, reunir de forma aditiva o poder computacional de um grupo de computadores para a realização de uma tarefa procurando explorar em cada um dos componentes o recurso computacional mais apropriado. Desta forma pode-se imaginar uma rede com n computadores como sendo uma máquina virtual dotada de n processadores independentes cada qual executando de forma concorrente uma parte do programa. Exemplificando, imaginemos um ambiente de cálculos onde um microcomputador é utilizado para distribuir uma tarefa de cálculo entre um grupo de computadores com processamento de ponto flutuante extremamente rápido. Estes computadores, por sua vez, utilizam um servidor de discos e arquivos de grande capacidade para armazenar seus resultados. Estes resultados podem ser analisados pelo microcomputador que iniciou a tarefa ou por uma estação de trabalho com recursos gráficos sofisticados.

Desta forma o emprego adequado de métodos de computação distribuída na construção de um programa permite reduzir o tempo de execução de uma tarefa bem como também expandir a dimensão dos problemas. Adicionalmente a capacidade de agrupar dinamicamente sistemas com diferentes capacidades permite a criação de computadores virtuais especializados e otimizados para uma finalidade específica.

Por outro lado, por incorporarem todos os avanços das eras que a antece-

deram, as redes de computadores não excluem os modelos convencionais de programação sequencial permitindo o aproveitamento integral dos programas existentes.

1.4 Objetivos deste Trabalho

Considerando o baixo custo e a disponibilidade cada vez maior das redes de computadores nas universidades e centros de pesquisa é urgente e necessária a exploração de técnicas de programação disponíveis neste ambiente.

Pretendemos, portanto, neste trabalho:

- Localizar as necessidades da química computacional em termos de computação distribuída e visualização científica identificando métodos e ferramentas de programação apropriados;
- Os principais obstáculos encontrados pelos químicos computacionais para a utilização dos recursos gráficos de visualização tridimensional disponíveis em Unix são a falta de familiaridade com a linguagem C, a relativa complexidade do sistema X de janelas que é baseado em eventos e a estreita relação entre o X e o sistema operacional que exige um conhecimento mais profundo deste. A finalidade do trabalho aqui realizado em computação gráfica é construir uma biblioteca com os componentes básicos de interesse químico baseada nos recursos e ambientes gráficos do X e do Unix de forma a isola-los do pesquisador;
- Desenvolver e escrever programas de visualização molecular utilizando esta biblioteca;
- Aplicar as técnicas de computação distribuída em rede a programas e métodos numéricos de interesse químico.

1.5 Bibliografia

1. *Salus, P. H.*, **A Quarter century of Unix**, Addison-Wesley, EUA, 1994.

2. *Stoll, C.*, **Silicon Snake Oil**, Doubleday, EUA, 1994.

3. *Dowd, K.*, **High Performance Computing**, O'Reilly & Associates, Sebastopol, EUA, 1993.

4. *Pountain, D., Bryan, J.*, **All Systems Go**, Byte, 112, **17**, 8, 1992.

Capítulo 2

Recursos Computacionais Utilizados

2.1 Equipamentos

Este trabalho foi desenvolvido utilizando uma série de computadores e estações de trabalho rodando diferentes versões do Unix. Os resultados apresentados foram obtidos nas seguintes estações de trabalho e microcomputadores:

- Sun Microsystems SparcServer modelo 330, 24Mbytes RAM, 1.9Gbytes disco, SunOS 4.1.1 e NetBSD1.2 [1];
- Sun Microsystems SparcStation 1+, 16Mbytes RAM, 200Mbytes disco, SunOS 4.1.1 e NetBSD1.2;
- IBM PS/2 modelo 95, Pentium 60MHz, 32Mbytes RAM, 2Gbytes disco, Solaris 2.1 e 2.4;
- IBM PC/AT clone, 386 DX 40 MHz, 8 Mbytes RAM, 400 Mbytes disco, FreeBSD [2] 1.0 e 2.0;
- IBM PC/AT clone, 486 DX (33 MHz e 66 MHz), 16 Mbytes RAM, 320 Mbytes disco, FreeBSD 2.0;
- IBM PC/AT clone, Pentium (100MHz, 133MHz e 166MHz), 16, 32 e 64 Mbytes RAM, 800 a 1.8Gbytes de disco, FreeBSD2.0;

2.2 Compiladores e Bibliotecas

Visando a interoperabilidade entre todas as plataformas de hardware e sistemas operacionais evitou-se a utilização de ferramentas de programação, bibliotecas e recursos de software que não fossem comuns a todos os sistemas e equipamentos empregados. Esta opção resultou na escolha de compiladores e bibliotecas para os quais dispõe-se dos fontes para compilação em cada sistema operacional escolhido.

Para o desenvolvimento dos programas os seguintes compiladores foram utilizados:

- Compilador GNU C, GCC, versões 2.2 a 2.7, [3];
- Conversor Fortran77 para C f2c, [4];
- Compilador GNU Fortran77 versão 0.5, [5];
- Interpretador Perl versão 5.0, [6];
- Biblioteca PVM versões 2 e 3, [7]
- Biblioteca TCGMSG [8];
- Biblioteca VOGL versão 3 [9];
- Biblioteca Mesa versões 1 e 2 [10];
- X Window System, versão 11, revisões 5 e 6 [11];
- Sun Xview Toolkit versão 3.2 [12];
- GLUT (OpenGL Utility Toolkit) versão 3.0 [13]

2.3 Ferramentas de Depuração e Análise de Desempenho

As ferramentas de depuração e aquelas utilizadas para medidas de desempenho foram:

- GNU Debugger - gdb versão 4.0 [14];
- BSD profiler - gprof - 4.4BSD [1,2];
- bonnie - versão 1.0, benchmark de sistema de discos [15];

- STREAM - benchmark de sistemas de memória [16];
- FLOPS - benchmark de desempenho numérico [17];
- bing - benchmark de desempenho de rede (ICMP) [18];
- tcpblast - versão 1.0, benchmark de desempenho de rede (TCP) [19];
- netperf - versão 2.1.1, benchmark de desempenho de rede (TCP e UDP) [20];

2.4 Ferramentas de Visualização e Processamento de Imagens

- XV versão 3.10 [21];
- XPaint versão 2.4.8 [22];
- ghostscript versão 4.0 [23];

2.5 Referências

1. **NetBSD:** The NetBSD Project, ©1993 1997, disponível em *ftp://ftp.netbsd.org/pub/NetBSD*

2. **FreeBSD:** The FreeBSD Project, ©1993 1997, disponível em *ftp://ftp.freebsd.org/pub/FreeBSD*

3. **gcc:** Richard Stallman *et. all*, ©1989 1997 Free Software Foundation, disponível em *ftp://prep.ai.mit.edu/pub/gnu*

4. **f2c:** S. L. Feldman *et. all*, AT&T Bell Labs, ©1990 1995, disponível em *http://www.netlib.org/f2c/*

5. **g77:** Craig Burley, ©1991 1997 Free Software Foundation, disponível em *ftp://prep.ai.mit.edu/pub/gnu*

6. *Wall, L., Christiansen, T, Shwartz, R.*, **Programming Perl**, O'Reilly & Associates, Inc. Usenix, Sebastopol, EUA, 2ed, 1996. disponível em: *http://www.perl.com/src/latest.tar.gz*

7. **PVM:** Al Geist *et. all*, ©1991, 1997, University of Tennessee e Oak Ridge National Laboratory, disponível em *ftp://netlib2.cs.utk.edu/pvm3*

8. **TCGMSG:** Robert J. Harrison, *rj_harrison@pnl.gov*, Battelle Pacific Northwest Laboratory,disponível em *ftp://ftp.tcg.anl.gov/pub/tcgmsg/*

9. **VOGL:** Department Of Engineering Computer Resources, Faculty Of Engineering, University of Melbourne, ©1991, 1997, disponível em *ftp://gondwana.ecr.mu.OZ.AU*

10. **Mesa:** Brian Paul, *brianp@ssec.wisc.edu* , ©1996, 1997, disponível em *ftp://iris.ssec.wisc.edu/pub/Mesa*

11. **X11R6:** X Consortium e XOpen, ©1984, 1997, disponível em *ftp://ftp.xfree86.org*

12. **XView:** Sun Microsystems Inc., ©1988, 1997, disponível em *ftp://ftp.freebsd.org/pub/FreeBSD/pub/distfiles*

13. **glut:** Mark J. Kilgard , *mjk@sgi.sgi.com*, ©1994, 1995, 1996, diponível em *http://reality.sgi.com/employees/mjk_asd/glut3/glut3.html*

14. **gdb:** Richard Stallman *et. all*, ©1989 1997 Free Software Foundation, disponível em *ftp://prep.ai.mit.edu/pub/gnu*

15. **bonnie:** Tim Bray, ©1990 disponível em *ftp://ftp.sunet.se/pub/benchmark/Bonnie/*

16. **STREAM:** John D. McCalpin *mccalpin@cs.virginia.edu* *http://www.cs.virginia.edu/stream/*

17. **flops:** Al Aburto, *aburto@nosc.mil*, ©1992 disponível em *ftp://ftp.nosc.mil/pub/aburto/*

18. **bing:** Pierre Beyssac, *pb@fasterix.freenix.fr*, 1995, disponível em *ftp://ftp.ibp.fr/pub/networking/*

19. **tcpblast:** Daniel Karrenberg, *dfk@nic.eu.net*, disponível em *ftp://ftp.freebsd.org/pub/FreeBSD/ports/benchmarks/tcpblast/*

20. **netperf:** Information Networks Division, Hewlett-Packard Company, ©1993,1996 disponível em *ftp://ftp.cup.hp.com/dist/networking/benchmarks/netperf/*

21. **XV:** John Bradley, *xv@devo.dccs.upenn.edu* , ©1989, 1994, disponível em *ftp.cis.upenn.edu*

22. **Xpaint:** Torsten Martinsen , *torsten@danbbs.dk* , e David Koblas, *koblas@netcom.com* , 1994,1995,1996 dispon'ível em *http://www.danbbs.dk/ torsten/xpaint/*

23. **ghostscript** Aladdin Enterprises, ©1996, disponível em *ftp://ftp.cs.wisc.edu/ghost/aladdin/*

Capítulo 3

Visualização Molecular

3.1 Introdução

A computação gráfica molecular situa-se entre as áreas da computação gráfica com maior demanda computacional devido a natureza tridimensional dos objetos de interesse da química computacional e a necessidade de manipulação interativa em tempo real destes.

A criação de objetos geométricos, visualização de estruturas moleculares, superfícies de energia potencial, isosuperfícies com ou sem codificação de cor em função de uma dada propriedade são as funções mais comuns e requisitadas como auxílio à análise de dados de entrada ou resultados numéricos de cálculos.

Devido às limitações de hardware existentes até a década de 80 a computação gráfica molecular inicialmente baseou-se em programas que utilizavam bibliotecas bidimensionais (ex. Calcomp [2]) para a produção de imagens de objetos sólidos. Desta maneira ficava a cargo de cada programa todo o processamento geométrico necessário para a realização das operações básicas de escala, rotação, perspectiva e projeção da imagem [1]. Estas operações eram traduzidas, então, em uma série de chamadas à uma biblioteca vetorizada bidimensional para o desenho de segmentos de reta e o resultado enviado a um dispositivo de saída, normalmente um *plotter* ou terminal vetorial. O nível de interação era mínimo e para produzir uma nova imagem era necessário editar um arquivo de dados modificando os parâmetros necessários e refazer todo o processo. Um exemplo típico é o programa ORTEP-II [4]. Os programas desenvolvidos nessa época eram em sua maioria escritos em Fortran e por utilizarem uma biblioteca bidimensional externa ao programa caracterizaram-se por serem facilmente tranferíveis entre diversos sistemas operacionais.

A partir da década de 80 a popularização de microcomputadores equipados com cartões gráficos baseados em mapas de bits (*bit mapped display*) [5], bem como a incorporação de algoritmos sofisticados em coprocessadores gráficos de estações de trabalho, e o desenvolvimento de novos algoritmos [7,8,6,9] voltados para esta tecnologia conduziram à criação das primeiras bibliotecas tridimensionais que incorporavam em si toda as operações de processamento geométrico citadas anteriormente e adicionavam novos recursos poderosos de auxílio à visualização de objetos tridimensionais complexos como iluminação, interpolação de cor, sombreamento de profundidade e remoção de linhas e superfícies escondidas.

Neste trabalho foi desenvolvida uma biblioteca gráfica 3D de alta qualidade para computadores dotados de cartões gráficos `bit mapped`. A seguir são expostos de forma abreviada os conceitos básicos associados a estes dispositivos e como os mesmos foram empregados para atingir os objetivos propostos.

3.2 Dispositivos Gráficos de Saída de Vídeo

3.2.1 Introdução

A natureza interativa da computação gráfica molecular exige a produção de imagens em tempo real e, consequentemente, a utilização de dispositivos com capacidade de atualização dinâmica das imagens geradas. O caráter tridimensional dos objetos manipulados torna obrigatório o emprego de técnicas de realce de percepção visual como o uso de cores e iluminação. Os dispositivos de vídeo baseados na tecnologia de varredura fixa (*raster display*) são os mais adequados para estas tarefas pois atendem estes requisitos e estão disponíveis em todos os computadores modernos dotados de saída de vídeo.

Os dispositivos de varredura fixa operam da seguinte forma [8,9,10]. A imagem é gerada pelo processador e armazenada em um bloco de memória. Este bloco de memória (*frame buffer*) é acessado periodicamente pelo dispositivo e o seu conteúdo convertido em sinais elétricos analógicos (conversão DA) apropriados para a formação dos quadros da imagem. Estes quadros são então mostrados em um tubo de raios catódicos (TRC) ou tela de cristal líquido, (figura 3.1). Qualquer alteração realizada pelo processador no conteúdo do *frame buffer* reflete-se automaticamente imagem gerada no monitor. A qualidade da imagem gerada está diretamente relacionada às dimensões do *frame buffer* e às características do conversor DA. A tecnologia atual utiliza *frame buffers* com 8, 16 ou 24 bits por elemento de tela (pixel) e conversores DA de 6, 8 ou 24 bits. O desempenho destes dispositivos, está relacionado à velocidade com a qual o processador é capaz de gerar a imagem a ser arma-

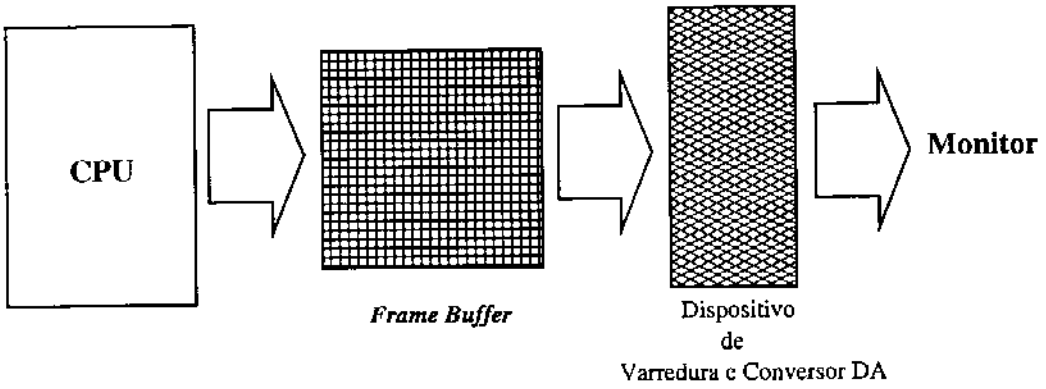


Figura 3.1: Dispositivo genérico de varredura fixa (*raster display*)

zenada no *frame buffer* e à velocidade que este pode ser varrido e convertido em sinais analógicos pelo dispositivo. Em virtude da sua importância para o presente trabalho os fatores acima são detalhados a seguir.

3.2.2 Dispositivos Gráficos de 24 bits

Os dispositivos gráficos com *frame buffers* de 24 bits, também denominados de dispositivos com 24 planos de côr ou dispositivos de côr verdadeira, armazenam a cor de cada pixel apresentado na tela em 3 bytes do *frame buffer*. O valor armazenado em cada byte representa a intensidade das cores elementares vermelha, verde e azul. Esta representação, também denominada RGB (do inglês, Red Green Blue) permite a subdivisão de cada uma das componentes em 256 níveis (0 a 255) de intensidade resultando num total de 256^3 ou 16777216 de cores. Durante uma varredura do *frame buffer* pelo sistema de conversão DA cada posição de memória é acessada e o conteúdo utilizado para geração do sinal a ser enviado ao monitor. A figura 3.2 ilustra esquematicamente estes dispositivos. A construção de um dispositivo de 24 bits de côr capaz de produzir uma imagem com resolução espacial de 1024 linhas por 768 colunas exige que este possua um *frame buffer* com 3(1024*768), ou 2359296 bytes de tamanho. Para aplicações de alta qualidade é necessária uma resolução espacial de 1280 linhas por 1024 colunas ou 3932160 bytes de memória. Ao mesmo tempo, considerando-se uma taxa de varredura de 70 Hz, ou 70 quadros por segundo, o sistema de conversão deve dispender no máximo 18 nanosegundos para efetuar a leitura e conversão de cada pixel, para uma resolução espacial de 1024 por 768, e de 11 nanosegundos para uma resolução de 1280 por 1024. Em termos de banda de memória necessária, a 70Hz, isso significa aproximadamente 270MBytes/s de banda Este fator,

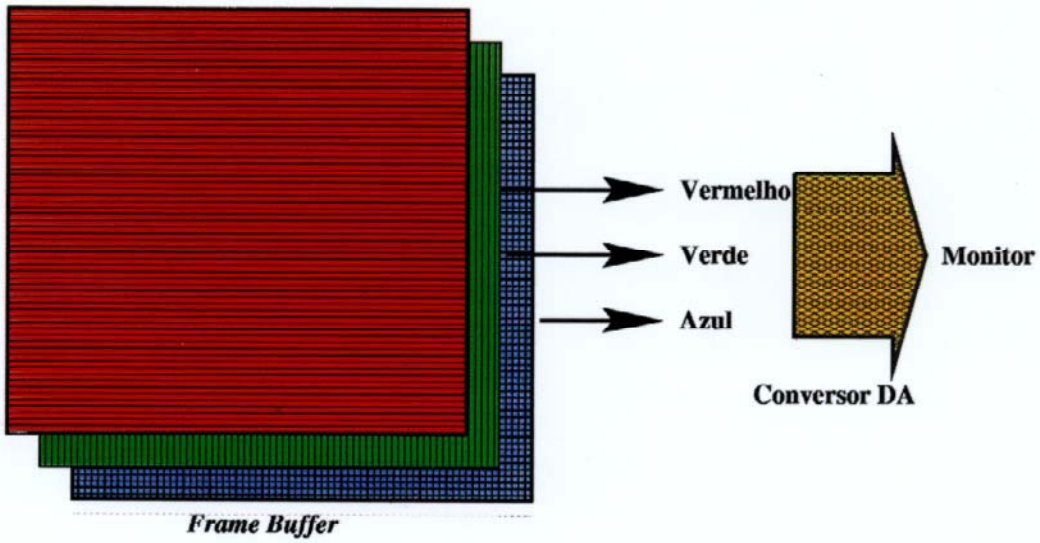


Figura 3.2: Dispositivo Gráfico de 24 bits de côr.

associado à resolução do conversor DA e a quantidade de memória de alta velocidade necessária, faz com que este tipo de dispositivo tenha um custo elevado. Tendo em vista esse aspecto e também a incapacidade do olho humano distinguir este número de cores foram desenvolvidos dispositivos de 16 ou 8 bits de côr, menos dispendiosos, para trabalhos que não exijam este padrão de qualidade.

3.2.3 Dispositivos Gráficos de 16 bits

Dispositivos de 16 bits são intermediários entre os dispositivos de 24 e os dispositivos de 8 bits. São capazes de representar 64k cores utilizando uma palavra de 16 bits para armazenar as três componentes de cor. Esta armazenagem pode ser programada da mesma forma que é descrita a seguir para os dispositivos de 8 bits. Estes dispositivos apresentam uma qualidade inferior aos de 24 bits mas imperceptível ao olho humano para a maioria das aplicações.

3.2.4 Dispositivos Gráficos de 8 bits

Os dispositivos de 8 bits utilizam o *frame buffer* de forma diferente dos dispositivos de 24 bits. No lugar de armazenar valores de côr cada byte do *frame buffer* contém um valor entre 0 e 255 que será utilizado como índice para a localização, em uma tabela de cores contendo 256 tripletes RGB, das intensidades de cada côr elementar do pixel. Dependendo do modo de operação

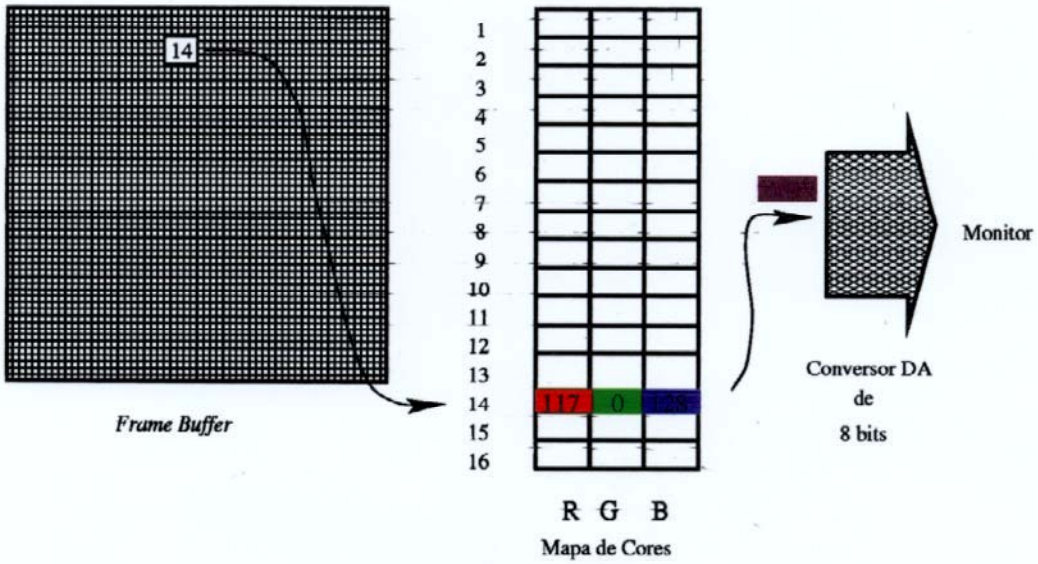


Figura 3.3: Dispositivo Gráfico de 8 bits de côr

do sistema de conversão DA a informação contida no *frame buffer* é interpretada de forma diferente de acordo com um mapa de cores armazenado no dispositivo. Estes modos de operação podem ser classificados em:

- Modo monocromático ou escala de cinza:
- Indexação RGB;
- Decomposição de índice em RGB;

Nos três modos de operação cabe ao programa do usuário ou ao sistema operacional a tarefa de carregar no dispositivo gráfico uma tabela de 256 tripletes RGB com a codificação de como deve ser traduzida a informação que será armazenada no *frame buffer*. Esta tabela é conhecida por mapa de cores `color map`. Durante a geração da imagem o processador armazena no *frame buffer* os índices correspondentes a cada côr de acôrdo com a tabela de cores. O sistema de conversão DA durante a varredura do *frame buffer* lê, para cada pixel, o valor armazenado e consulta a tabela de cores para obter o valor a ser usado na conversão. A figura 3.3 ilustra de forma esquemática a forma de operação de um dispositivo de 8 bits de côr. A interpretação do valor armazenado na tabela de cores depende do modo de operação do sistema DA, estes modos são descritos a seguir [10].

3.2.5 Modo monocromático ou escala de cinza

Neste modo de operação cada elemento da tabela de cores representa uma intensidade, entre 0 e 255, de apenas uma das cores elementares, vermelho, vêrde, azul ou uma intensidade, também entre 0 e 255, de cinza.

3.2.6 Indexação RGB

Neste modo de operação o sistema de conversão DA utiliza o valor armazenado no *frame buffer* para localizar na tabela de cores o triplete RGB que deve ser convertido para produzir a côr do pixel.

3.2.7 Decomposição de índice em RGB

Neste modo de operação, também conhecido por modo de côr verdadeira ou direta, os 8 bits que compõe a entrada na tabela são divididos em 3 blocos que armazenam a intensidade de cada uma das cores. Considerando a baixa sensibilidade do olho humano à côr azul uma divisão comum é a denominada 3/3/2 onde 3 bits são alocados para representar a intensidade do vermelho, 3 bits para o verde e 2 para o azul. Com este esquema é possível representar 2^3 ou 8 intensidades de vermelho e verde e 2^2 intensidades de azul num total de 256 cores.

3.2.8 Outras Características de Dispositivos de Varredura Fixa

Além das características que definem as resoluções espacial e cromática dos dispositivos de varredura fixa discutidas acima a demanda por alto desempenho fez com que outras capacidades fossem incorporadas a estes dispositivos, nos modelos de mais alto custo [9], tornando-os uma espécie de co-processador gráfico capaz de realizar um certo número de tarefas normalmente realizadas pelo processador central, deixando este livre para tarefas mais complexas. Dentre estas, as seguintes são de extrema importância em computação gráfica molecular

3.2.9 Buffer de Profundidade

O buffer de profundidade, também denominado Z-buffer [6,9], consiste num bloco de memória de 16 ou 32 bits onde são armazenados todos os valores das coordenadas Z de todos os pixels presentes em uma tela. A sua finalidade é auxiliar na remoção de superfícies e linhas ocultas. Cada vez que

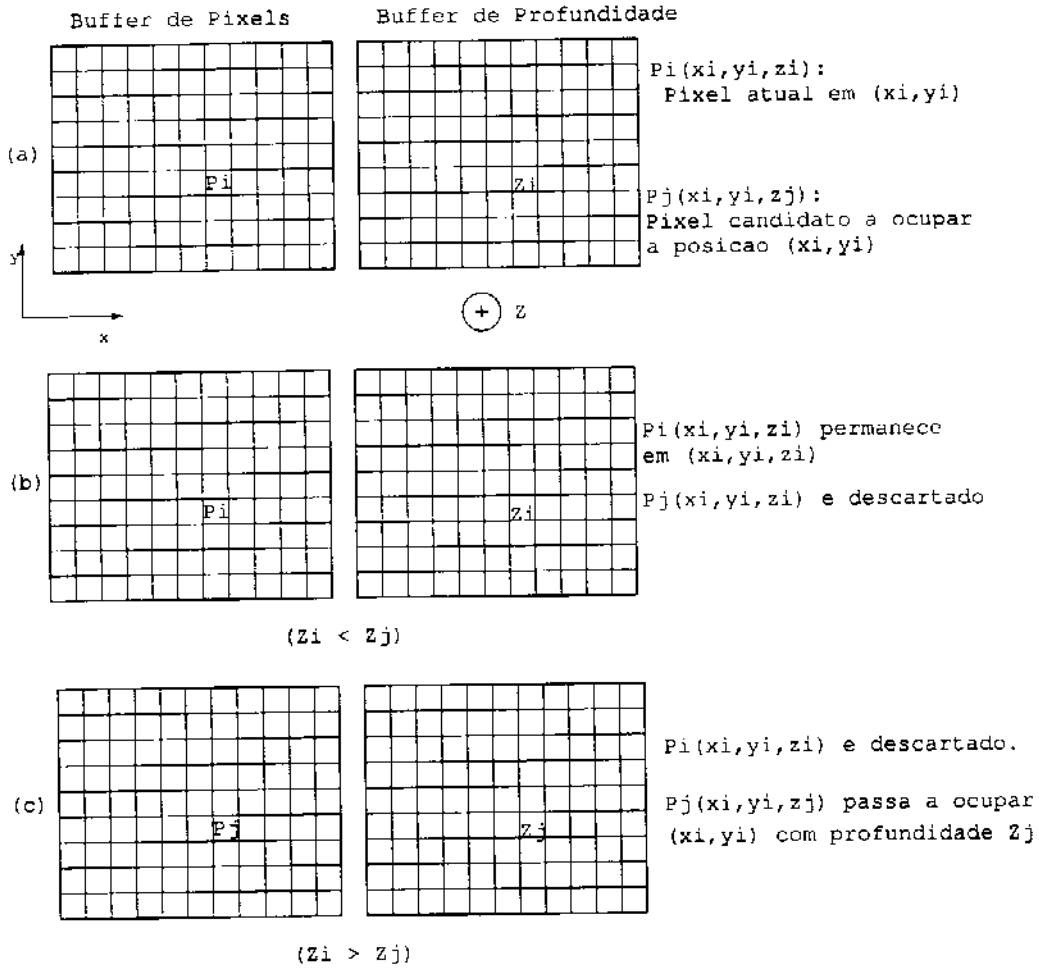


Figura 3.4: Funcionamento do buffer de profundidade

o processador envia um novo pixel para ser mostrado na tela o dispositivo gráfico compara o valor da coordenada Z deste contra o valor armazenado no Z-buffer, caso a comparação indique que o novo pixel encontra-se mais próximo do observador que aquele presente na tela então o novo pixel passará a ocupar aquela posição na tela e o seu valor da coordenada Z será armazenado no Z-buffer substituindo o valor antigo. Caso contrário ele será descartado por encontrar-se oculto (figura 3.4). A qualidade da remoção de linhas e superfícies ocultas dependerá da precisão numérica do Z-buffer (16, 24 ou 32 bits) e das dimensões da cena sendo processada no seu sistema de coordenadas original. Embora a construção de cenas associadas à computação gráfica molecular não seja particularmente suceptível a erros de precisão o mesmo não é válido para cenas envolvendo descrições topográficas

ou de engenharia de grande porte.

A disponibilidade de um Z-buffer em um dispositivo gráfico implica na implementação da lógica computacional necessária às comparações de profundidade, na restruturação da forma de processamento para a disponibilização da coordenada Z de cada pixel (na maior parte das vezes envolvendo interpolações numerícas de retas e planos) e na adição de um bloco de memória capaz de armazenar as coordenadas Z de todos os pixels existentes na tela. Este último requisito, para uma tela com dimensões de 1024 linhas por 768 colunas e um Z buffer de 32 bits, significa um bloco com dimensões de 3154944 bytes, que somados aos 2359296 bytes necessários para um dispositivo de 24 bits, implica em 5505024 bytes de memória ou cerca de 5.25 Mbytes. Esta quantidade atinge 8.75 Mbytes de memória para resoluções de 1280 linhas por 1024 colunas.

3.2.10 Buffer de Transparência

A capacidade de desenhar um objeto parcialmente transparente pode ser de extrema utilidade na composição de cenas complexas onde ocorra a superposição de diversas superfícies de interesse. Esta capacidade depende da existência de um buffer onde possam ser armazenados os coeficientes de transparência [9] de cada pixel existente na imagem. Este buffer de transparência, também conhecido como *Alpha-buffer*, atua de forma análoga ao Z-buffer sendo utilizado no instante em que o processador envia um novo pixel à tela.

O dispositivo gráfico, após realizar o teste de profundidade, realiza uma análise dos coeficientes de transparência (valores Alfa) para calcular a visibilidade do pixel e a côr resultante. Usualmente, devido à quantidade adicional de memória necessária este buffer consiste de valores de 8 bits de precisão e é armazenado juntamente com os 24 bits utilizados para a a definição da cor do pixel.

3.2.11 Processador de Vértices

Grande parte do esforço computacional envolvido na computação gráfica está associado aos cálculos trigonométricos de rotações, produtos matriciais e produtos escalares necessários para levar um objeto das coordenadas do mundo real para as coordenadas da tela de acordo com um conjunto de parâmetros que definem as condições de iluminação, perspectiva do observador e escalonamento. Se parte destas tarefas forem transferidas do processador central para o dispositivo gráfico de saída o desempenho computacional pode ser

incrementado liberando o processador central para a realização de outras tarefas. Com isto em mente foram desenvolvidos dispositivos gráficos de saída sofisticados capazes de receber do processador os parâmetros de uma cena, e a partir destes processa-la completamente utilizando as coordenadas dos vértices dos objetos. Ao processador cabe a tarefa de atualizar, conforme necessário, os parâmetros de definição da cena e transferir da memória principal para o dispositivo gráfico as coordenadas dos vértices, componentes de côr, vetores normais dos objetos e coeficiente de transparência.

A construção de dispositivos desta natureza varia desde processadores de triângulos com iluminação simples até sistemas de 4 ou mais processadores que operam em paralelo, com custo e desempenho proporcionais a sua complexidade e sofisticação [9].

3.2.12 Considerações Sobre Dispositivos Gráficos de Saída

Embora existam dispositivos gráficos com elevado grau de sofisticação e desempenho a grande maioria dos computadores ainda faz uso de dispositivos de 8 ou 16 bits de cor sem qualquer forma de coprocessamento auxiliar à CPU. Isto deve-se em parte ao custo associado aos sistemas mais sofisticados mas, principalmente, a ausência de padrões estabelecidos na indústria para a especificação e construção de tais dispositivos. Como consequência cada fabricante utiliza dispositivos de 8 bits de baixo custo e padrões bem estabelecidos ou busca a sua própria solução tecnológica. Neste segundo caso os custos associados ao desenvolvimento de software e hardware refletem-se no produto final. A utilização de tecnologia proprietária e de acesso restrito, por sua vez, impede a produção de software em larga escala que empregue o dispositivo.

Duas iniciativas foram tomadas para minimizar este problema, provendo bibliotecas e interfaces de programação independentes de hardware. Uma dessas iniciativas foi tomada pela Silicon Graphics Inc. através da especificação da biblioteca gráfica OpenGL [11], e a outra pelo X Consortium [12], formado por um conjunto de empresas e entidades envolvidas no desenvolvimento do ambiente gráfico X, na forma da especificação PEX [13] conforme será descrito a seguir.

Em vista do que foi exposto pode-se concluir que os dispositivos gráficos de saída de 8 e 16 bits de côr constituem, na atualidade, os elementos básicos de trabalho e desenvolvimento de aplicação da computação gráfica molecular. Qualquer trabalho realizado nesta área deve levar em conta as limitações de qualidade e desempenho associadas a estes dispositivos de forma que uma

aplicação possa ser executada utilizando os mesmos de forma adequada e aceitável. Nos casos em que dispositivos de 24 bits estiverem disponíveis (com ou sem aceleração adicional) a aplicação deve ser capaz de fazer uso destas capacidades sem necessidade de alteração do programa.

3.3 Bibliotecas e Padrões Gráficos

O desenvolvimento de programas gráficos é extremamente dependente da escolha de uma biblioteca e de um padrão para elaboração do projeto. Nos últimos anos várias propostas de bibliotecas e padrões foram feitas. Apenas algumas consolidaram-se na comunidade científica, seja por sua funcionalidade, ou seja, pelo seu desempenho quando associado ao computador e dispositivo gráfico apropriado. A seguir são descritas as principais bibliotecas gráficas para visualização científica.

3.3.1 Iris GL

Entre as diversas bibliotecas desenvolvidas a GL (*Graphics Library*) da Silicon Graphics foi a que obteve maior sucesso por associar um conjunto poderoso de primitivas gráficas a um hardware extremamente otimizado. A GL possui todos as funções e requisitos necessários para a computação gráfica de alto desempenho mas estes recursos de programação só estão disponíveis nos equipamentos dotados do hardware fabricado pela Silicon Graphics, nominalmente a própria Silicon e alguns equipamentos IBM.

3.3.2 PHIGS, PEX

Devido a grande variedade de hardware gráfico e de bibliotecas de programação gráficas utilizados pela indústria ao longo da década de 70, cada uma dependente do hardware do fabricante, a partir de 1984 foi iniciada a definição de um padrão de biblioteca gráfica tridimensional pelo ISO que foi denominado de PHIGS (Programmer's Hierarchical Interactive Graphics System). Este padrão foi estabelecido em 1988 [14] e logo acrescido de uma série de extensões que definiram o PHIGS-PLUS [15].

Paralelamente desde 1984 vem sendo desenvolvido no Massachussets Institute of Technology, e posteriormente pela Open Software Foundation, em consórcio com os maiores fabricantes de computadores (*MIT X-Consortium* [12]) um padrão de comunicação gráfica entre computadores conectados em rede denominado *X Window System*. Este padrão permite que computadores

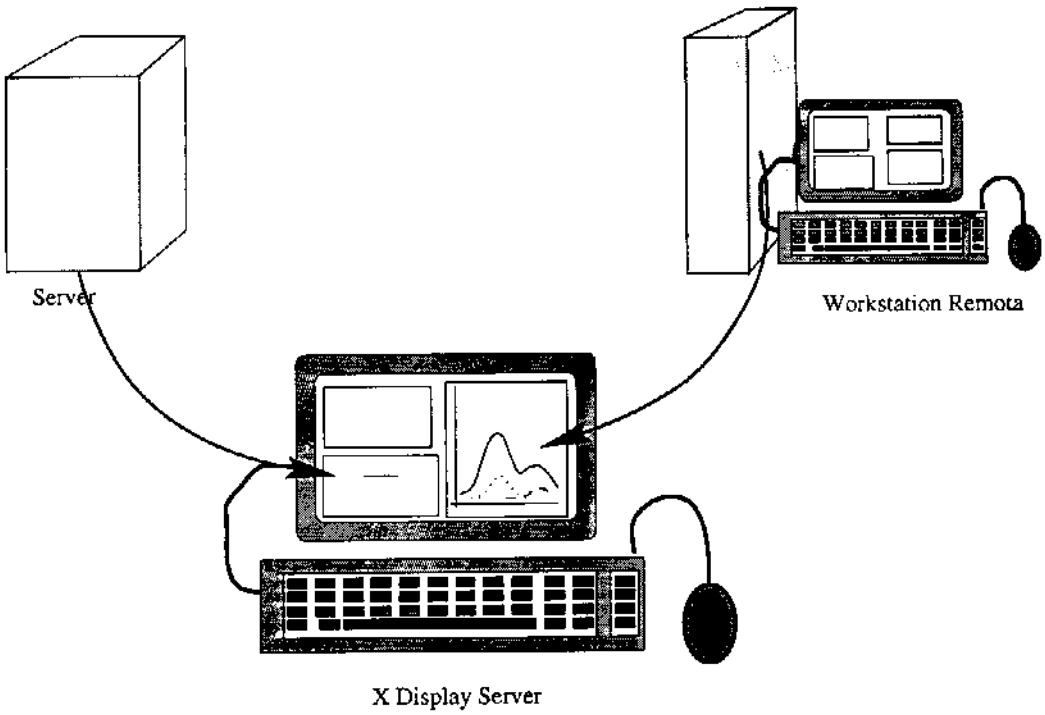


Figura 3.5: Conceito de servidor de tela utilizado pelo X.

acessem recursos gráficos remotos através de uma rede tranformando uma estação de trabalho em um servidor de tela gráfica (*X display server*) capaz de atender simultaneamente diversos clientes (figura 3.5) [10]. Este padrão , originalmente voltado para textos e gráficos bidimensionais, passou a incorporar a partir da versão 11.5 (X11R5) diversas funções do PHIGS-PLUS, através de uma extensão ao servidor de tela, denominada PEX (PHIGS Extensions to X [13]) e de duas bibliotecas de programação, capazes de gerar chamadas através da rede para a produção de objetos tridimensionais, `libphigs.a` e `PEXlib.a` [16].

Enquanto o PHIGS e a biblioteca `libphigs.a` consistem exatamente em um padrão de programação gráfica, com as mesmas funcionalidades da GL, a `PEXlib.a` consiste de uma biblioteca de acesso direto as funções do protocolo de rede PEX sobre a qual podem ser desenvolvidas aplicações ou uma interface de programação na forma de uma biblioteca como a `libphigs.a`.

3.3.3 OpenGL

Um dos pontos fracos da Iris GL é não possuir, como o X, a capacidade de comandar a execução remota de gráficos e depender exclusivamente dos

equipamentos da Silicon Graphics ou IBM. Visando contornar esta situação a Silicon Graphics propôs em 1991 um novo padrão de biblioteca gráfica denominado OpenGL [11]. A OpenGL, em sua versão para Unix, é capaz de interagir remotamente com outros computadores através do protocolo de rede denominado GLX [17]. Este padrão é desenvolvido e mantido pelo OpenGL Advisory Revision Board (ARB). Nos mesmo moldes do X Consortium, a OpenGL e o ARB tentam seguir as características abertas de desenvolvimento e estabelecimento dos padrões do PEX e da PEXlib. Através do fornecimento de bibliotecas auxiliares para a sua integração com ambiente X [18], para o desenvolvimento de aplicações (GLU [19]), interfaces gráficas de interação com o usuário (Glut [20]) além de literatura técnica especializada e abundante [21,22].

Em uma análise detalhada da OpenGL e do PEX/PEXlib realizada por Akin [23] em 1991, a OpenGl mostrou-se superior a este como plataforma genérica para o desenvolvimento de aplicações gráficas, no que diz respeito a desempenho, programabilidade, evolução e adaptação aos diferentes dispositivos gráficos e de rede. O PEX/PEXlib, por sua vez mostrou-se mais adequado para aplicações específicas nas áreas de CAD/CAM de engenharia.

3.3.4 MESA

Paralelamente ao desenvolvimento da OpenGL, Brian Paul, da Universidade de Wisconsin, desenvolveu e distribui livremente uma versão gratuita clone da OpenGL denominada Mesa [24].

Esta biblioteca possui todas as funções da OpenGL e apresenta desempenho igual ou superior a ela em sistemas sem aceleração gráfica em hardware. Por utilizar o ambiente X, ela incorpora o suporte de rede deste além de poder ser compilada e utilizada em qualquer sistema operacional que suporte X.

3.4 Interfaces Gráficas com o Usuário

Embora as bibliotecas gráficas possuam todas as funções necessárias para a geração de objetos sólidos, elas são relativamente pobres com relação ao conjunto de funções (*widgets*) para a produção de interfaces gráficas de interação com o usuário (*GUI, Graphics User Interface*) se comparadas a outros conjuntos de ferramentas de programação em X (OpenLook Toolkit, XView Toolkit [25], Motiff Toolkit [26], glut [20,27]). Isto se deve a um decisão do ARB e do X Consortium que preferiram deixar a cargo de cada programador ou fabricante a escolha do conjunto de ferramentas que julgar mais adequado.

O mesmo ocorre com a GL e com a VOGL (descrita a seguir). Desta forma para a execução deste trabalho foram utilizadas bibliotecas de programação que possuissem apenas os *widgets* necessários e atendesse os requisitos de interoperabilidade entre diversos sistemas operacionais conforme as especificidades de cada programa, nominalmente, XView Toolkit e GLUT.

3.5 VoglZ Desenvolvimento de Uma Biblioteca 3D

Um dos objetivos deste trabalho é desenvolver uma biblioteca gráfica avançada para visualização científica em química e, ao mesmo tempo, obter a maior interoperabilidade entre sistemas computacionais Unix. Para isto foi utilizada como código fonte base a bilioteca VOGL, uma biblioteca gráfica baseada em X11, desenvolvida pelo *Department Of Engineering Computer Resources, Faculty Of Engineering*, da Universidade de Melbourne [28]. A VOGL, como o X11, é distribuída na forma de programa fonte o que permitiu que fosse modificada neste trabalho para incorporar as funções necessárias. As modificações introduzidas na VOGL visaram capacitá-la à geração de imagens tridimensionais com iluminação e efeitos de transparência e remoção de linhas e superfícies ocultas.

A seguir são descritos os detalhes das modificações feitas e as implementações utilizadas para a adição de cada função.

3.5.1 Buffer de Profundidade

A adição de um buffer de profundidade foi realizada pela inserção na estrutura de dados da VOGL de um vetor de inteiros de 32 bits com dimensão igual ao produto `largura * altura` da tela onde cada inteiro corresponde ao valor `z` do pixel situado na posição `x,y` da tela.

A utilização deste buffer é ativada pela função `zbuffer(` *on/off* `)`, onde *on* e *off* criam ou destroem o vetor e pela função `zclear()` utilizada para inicializar o vetor.

Para que cada pixel possa ser testado utilizando o Z-buffer antes de ser, ou não, armazenado no frame buffer, as funções encarregadas pelo processamento de linhas e polígonos foram reescritas.

3.5.2 Processamento de Linhas

A biblioteca Vogl original faz uso do processador de linhas do dispositivo gráfico utilizado para desenhá-las no frame buffer, enviando a este pares

$(x_1, y_1), (x_2, y_2)$. Isto impede que pontos (x_i, y_i) contidos no segmento de reta $(x_1, y_1), (x_2, y_2)$ sejam testados pelo buffer de profundidade. Para contornar este problema foram introduzidas duas rotinas genéricas na biblioteca. A primeira, `bres(x,y)` realiza a decomposição de um segmento de reta definido por $(x_1, y_1), (x_2, y_2)$ em um conjunto de N pares (x_i, y_i) utilizando o algoritmo de Bresenham [6,9] conforme ilustra a figura 3.6 A visibilidade dos

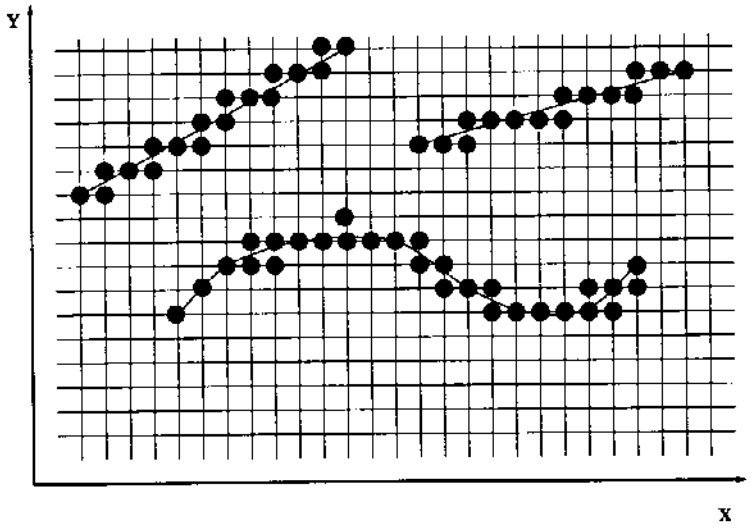


Figura 3.6: Decomposição de retas e curvas em dipositivos de varredura fixa utilizando o algoritmo de Bresenham

pontos (x_i, y_i) resultantes desta decomposição é testada em relação ao buffer de profundidade e, caso sejam visíveis, enviados para o dispositivo gráfico através da segunda função, `Vpix(ix, iy,r,g,b,flag)`.

```
 Vpix(ix, iy,r,g,b,flag)
int ix,iy; /* Coordenadas de tela */
int r,g,b; /* Componentes de cor do pixel */
int flag;  /* -1 ativa a cor (r,g,b) em (ix,iy)
               0 ativa o pixel em (ix,iy) utilizando a cor atual
               1 ativa o pixel em (ix,iy) utilizando a cor (r,g,b)
           */
```

3.5.3 Processamento de Polígonos

O processamento de polígonos na biblioteca Vogl original é feito através do envio de uma lista de N vértices $(x_1, y_1), (x_2, y_2), \cdots (x_N, y_N)$ para o proces-

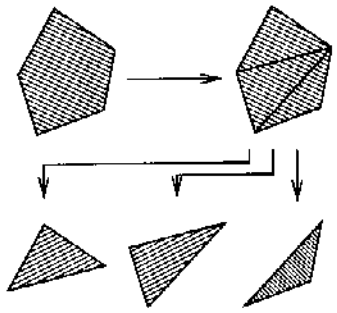


Figura 3.7: Decomposição de um polígono em triângulos

sador de polígonos do dispositivo gráfico. Assim, como no caso do processamento de linhas, isto impede que os pontos internos ao polígono sejam testados quanto a sua visibilidade utilizando o Z-buffer. Para solucionar esta limitação foram adicionadas duas novas funções à biblioteca. A primeira delas, decompõe um polígono de N vértices em $N-2$ triângulos. conforme ilustra figura 3.7. A decomposição do polígono em triângulos simplifica con-

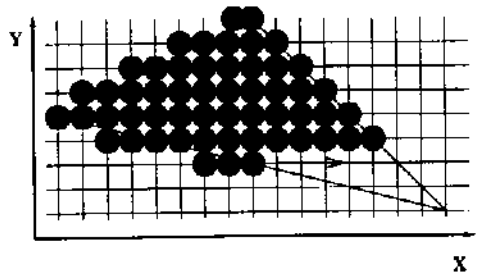


Figura 3.8: Conversão de um triângulo em pixels

sideravelmente o processamento e os cálculos de interpolação das posições e profundidades dos pixels internos e a sua decomposição em linhas adjacentes empregando o algoritmo de Bresenham nas suas arestas como ilustra a figura 3.8.

```
draw_poly (n, v)
     int n;                          /* Numero de vertices no poligono */
     vertex v[];                     /* Vetor de vertices */
```

Através das modificações descritas o processamento de linhas e polígonos passou a possuir testes de visibilidade completos para cada pixel desenhado na tela eliminando as ambiguidades com relação a posição relativa de objetos. A figura 3.9 sumariza estas modificações.

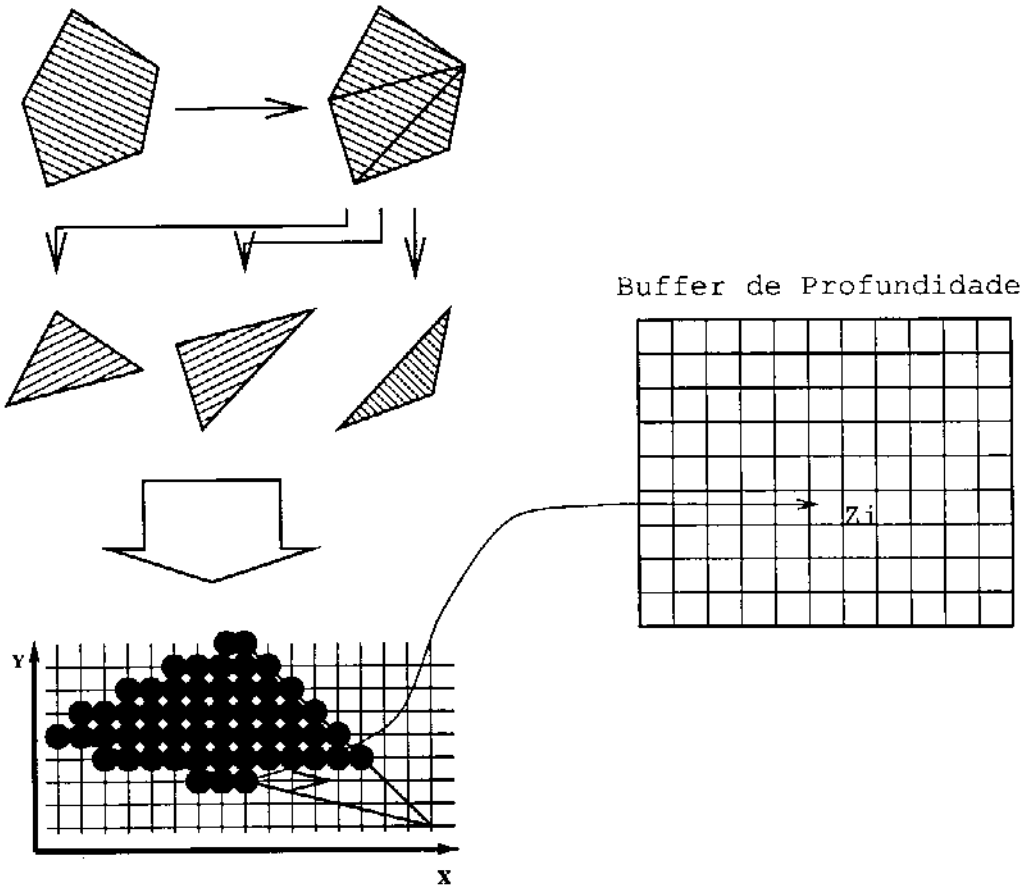


Figura 3.9: Processador de polígonos com buffer de profundidade implementado na biblioteca VoglZ

3.5.4 Cálculos de Iluminação e Reflexão

A remoção de linhas e superfícies ocultas sozinha não é suficiente para a obtenção de gráficos com definição adequada dos objetos. Para isto é necessária a utilização de recursos de iluminação da cena onde estes objetos são desenhados. Esta iluminação é um artifício que busca realçar a percepção tridimensional dos objetos utilizando fontes de luz e propriedades de reflexão artificiais definidas para cada superfície. Em função dos objetivos do trabalho de visualização vários modelos de iluminação podem ser adotados. A figura 3.10 ilustra os diversos resultados possíveis, para um mesmo objeto utilizando quatro modelos que foram implementados na biblioteca neste trabalho. O cálculo de iluminação de um objeto depende das propriedades de reflexão definidas para este e do modelo de iluminação empregado. As

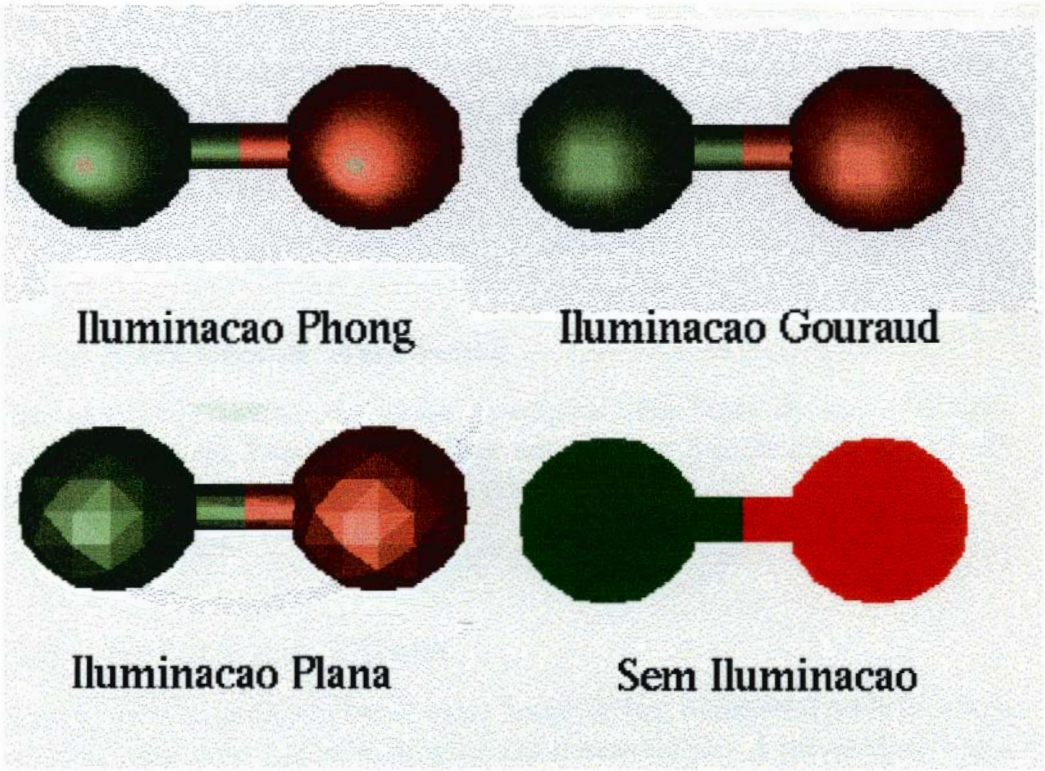


Figura 3.10: Modelos de iluminação implementados na biblioteca VoglZ

propriedades de reflexão de um material são modeladas de acordo com os princípios básicos da física, mas não buscam a exatidão da óptica e sim o realce artificial visando a melhoria da percepção espacial por parte do observador. Os modelos de iluminação aplicados também visam mimetizar a natureza sem, contudo, tentar reproduzi-la de forma exata.

3.5.5 Propriedades de Reflexão dos Materiais

A iluminação e os cálculos de reflexão em computação gráfica são realizados através do cálculo da cor resultante da interação de uma ou mais fontes de luz com um ponto do objeto gráfico em relação a posição do observador. Esta interação depende das propriedades de material deste ponto e da sua orientação no espaço. As propriedades definidas neste trabalho foram:

3.5.6 Emissividade

O material pode atuar como uma fonte luminosa e ser visível na tela mesmo na ausência de fontes de luz, esta propriedade é definida por 3 coeficientes k_e^R, k_e^G, k_e^B para cada componente de cor do material. A intensidade I_e para cada cor do pixel será:

$$I_e^{cor} = k_e^{cor} \tag{3.1}$$

3.5.7 Refletância Ambiente

O material pode refletir de forma isotrópica uma fonte de luz ambiente, também isotrópica. Esta propriedade é definida por 3 coeficientes k_a^R, k_a^G, k_a^B para cada componente de cor do material;

$$I_a^{cor} = k_a^{cor} I_{ambiente} \tag{3.2}$$

3.5.8 Refletância Difusa

Se houver na cena gráfica uma ou mais fontes de luz, o material pode interagir com estas, refletindo de forma difusa a cor destas fontes. A intensidade desta interação depende do ângulo formado entre o vetor $\vec{N}$ normal à sua superfície e o vetor $\vec{L}$ que define a direção da fonte luminosa e dos coeficientes de reflexão difusa do material para cada cor, k_d^R, k_d^G, k_d^B. A intensidade I_d^{cor} de uma das componentes de cor resultante seria:

$$I_d^{cor} = I_L^{cor} k_d^{cor} (\vec{N} \cdot \vec{L}) \tag{3.3}$$

3.5.9 Refletância Especular

A refletância especular depende da quantidade de luz refletida pelo material que alcança o observador. Desta forma, para o seu cálculo, é necessário que a partir do vetor $\vec{R}$ de reflexão (determinado pelo vetor $\vec{N}$ normal à superfície e pelo vetor $\vec{L}$ que define a fonte de luz) seja estimado o ângulo ϕ que este faz com o vetor $\vec{V}$ que define a direção de observação da cena. Esta propriedade é definida pelos coeficientes de reflexão especular k_s^R, k_s^G, k_s^B. A intensidade I_s^{cor} seria dada por

$$I_s^{cor} = k_s^{cor} I_L^{cor} \cos(\phi) = k_s^{cor} I_L^{cor} (\vec{R} \cdot \vec{V}) \tag{3.4}$$

A cor resultante de um pixel na tela será a soma de todas as cores resultantes dos cálculos para cada um dos materiais:

$$\begin{aligned} I_R^{total} &= I_e^R + I_a^R + I_d^R + I_s^R \\ I_G^{total} &= I_e^G + I_a^G + I_d^G + I_s^G \\ I_B^{total} &= I_e^B + I_a^B + I_d^B + I_s^B \end{aligned} \tag{3.5}$$

A figura 3.11 ilustra os diferentes resultados obtidos para um mesmo objeto utilizando diferentes propriedades de material.

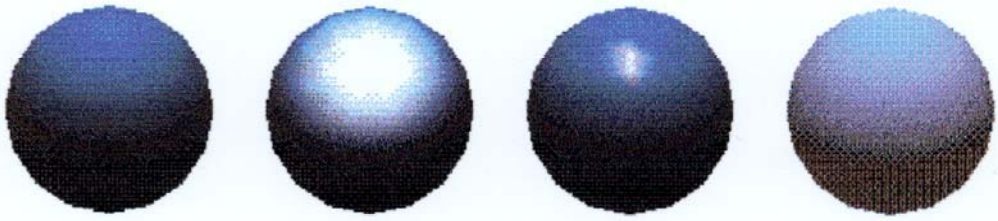


Figura 3.11: Representações gráficas de um mesmo objeto utilizando diferentes propriedades de material

3.5.10 Modelos de Iluminação

A figura 3.10 ilustra os quatro modelos de iluminação implementados neste trabalho. São eles:

3.5.11 Ausência de Iluminação

Neste modelo não são feitos cálculos de iluminação, apenas é realizada a eliminação de linhas e superfícies ocultas utilizando o teste de profundidade. Este modelo é adequado para o desenho de linhas e pontos apenas pois não fornece indicações de relevo ou curvatura de superfícies.

3.5.12 Iluminação Plana

O modelo de iluminação plana [9] utiliza os vértices do triângulo para determinar o vetor $\vec{N}$ normal à superfície deste. Este vetor $\vec{N}$ é utilizado para calcular a cor resultante do triângulo utilizando todas as propriedades definidas para o material que o compõe. Esta cor, então, é utilizada para desenhar

todos os pixels que compõe o triângulo. O uso deste modelo é adequado para o desenho de poliedros como cubos, octaedros, etc mas não é adequado para objetos com superfícies curvas.

3.5.13 Iluminação Gouraud

O modelo de Gouraud [6,9] calcula inicialmente as cores dos três vértices do triângulo utilizando as propriedades de reflexão e os respectivos vetores normais. A seguir os pixels internos ao triângulos são desenhados tendo as suas cores interpoladas a partir das cores dos vértices. Este modelo é o que apresenta a melhor relação entre o seu custo computacional e qualidade gráfica para superfícies curvas, no entanto não reproduz adequadamente os efeitos de reflexão especular.

3.5.14 Iluminação Phong

O modelo de iluminação Phong [6,9] utiliza os vetores normais e as componentes de cor de cada vértice do triângulo para interpolar os valores das componentes de cor e o vetor normal de **cada** pixel interno do triângulo. Os valores interpolados são então utilizados para, juntamente com as propriedades de reflexão do material, calcular a cor resultante da interação com as fontes de luz presentes na cena gráfica.

3.5.15 Implementação Computacional

A adição de iluminação à VOGL exigiu a reescrita completa das estruturas internas de dados para armazenamento dos vetores normais n^x, n^y, n^z e para armazenar as três componentes de cor (vermelho, verde, azul, ou RGB) de cada vértice. Com esta modificação cada vértice passou a ser definido por 3 tripletes $v_i = v(x_i, y_i, z_i, n_i^x, n_i^y, n_i^z, R_i, G_i, B_i)$ onde x_i, y_i, z_i e n_i^x, n_i^y, n_i^z são número reais de precisão simples e R_i, G_i, B_i são inteiros sem sinal de 8 bits.

Além da alteração na definição dos vértices foi criado um buffer interno para o armazenamento em memória da imagem gerada na tela. O armazenamento interno das três componentes de cor é necessário uma vez que este trabalho visa criar uma biblioteca independente de hardware e não há um procedimento padrão para a leitura do **frame buffer** seja ele de 8, 16 ou 24 bits e os cálculos necessitam ter acesso a cor atual de um pixel no **frame buffer**. Nesta implementação estes dados foram armazenados em um vetor de números inteiros de 32 bits `*rgbbuf` na forma:

```
 Alfa   Vermelho  Verde   Azul
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
+-------+-------+-------+-------+
<------------32 bits------------>
```

E o acesso a cada componente de cor é feito através da função `cpack()`

```
void cpack ( i )
int i           /* valor de cor do pixel */
{
short alfa, r, g, b;
  alfa = (short) ((0xff000000 & rgbbuf[i]) >> 24);
     r = (short) ((0x00ff0000 & rgbbuf[i]) >> 16);
     g = (short) ((0x0000ff00 & rgbbuf[i]) >>  8);
     b = (short) ((0x000000ff & rgbbuf[i])) ;
}
```

Para facilitar a conversão de valores de ponto flutuante das componentes de cor (entre 0.0 e 1.0) para a representação interna do `*rggbuf` foi criada função `rgbpak()`

```
int rgbpak(a, r, g, b)
int a;            /* coeficiente de transparencia discutido abaixo */
float r,g,b;      /* componentes de cor r,g,b em ponto flutuante */
{
return (a<<24)|((int)(255.0*r)<<16) |
               ((int)(255.0*g)<< 8) |
               ((int)(255.0*b));
}
```

As propriedades de reflexão dos materiais e das luzes são definidas e controladas pela função `lmdef()`

```
void  lmdef(deftype, index, properties)
int   deftype, index ;
float properties[];
/* Onde

deftype      = DEFMATERIAL ou DEFLIGHT
index        = indice identificador do material ou fonte luminosa
properties[] = lista de coeficientes de reflexao ou
```

```
                transparencia do material ou luzes presentes na cena
*/
```

A ativação ou desativação das fontes de iluminação é feita pela função `lmbind()` descrita a seguir:

```
void lmbind (target, index)
int target, index;
/* Onde

target = LIGHT0 a LIGHT8, uma das 8 fontes de iluminacao possiveis
index  = 0 desativa a fonte de luz, 1 ativa a fonte de luz
*/
```

Para o processamento de cada triângulo foi desenvolvida a função `Renderer()`, esta função é encarregada de fazer a decomposição de cada triângulo e realizar os cálculos de interpolação necessários para o processamento de cada pixel pela função seguinte. A função `Renderer()` tem a seguinte definição:

```
void
Renderer(n, ppx, ppy, pv, pn)
int n;    /* numero de vertices */
int  ppx; /* coordenadas x dos vertices no espaco de projecao */
int  ppy; /* coordenadas y dos vertices no espaco de projecao */
float  pv[][4]; /* coordenadas dos vertices no espaco real */
float  pn[][3]; /* normais dos vertices */
```

A função Renderer(), por sua vez utiliza a função `light_evaluate()` para cada pixel. Esta função é a responsável por todos os cálculos de iluminação e tem como parâmetros:

```
void
light_evaluate(nx, ny, nz, red, green, blue, x, y, z, r, g, b)
float nx,ny,nz;       /* Vetor normal */
float red,green,blue; /* Cor do pixel */
float x,y,z;          /* Coordenadas do pixel */
float *r,*g,*b;       /* Cor calculada do pixel */
```

O cálculo da cor resultante dos efeitos de iluminação para um dado pixel é realizado segundo a equação 3.6 iterada sobre todas as fontes de iluminação presentes na cena:

$$
\begin{aligned}
I_R^{total} &= I_e^R + I_a^R + \sum_{L=1}^{N_L} \left\{ I_d^R + I_s^R \right\} \\
I_G^{total} &= I_e^G + I_a^G + \sum_{L=1}^{N_L} \left\{ I_d^G + I_s^G \right\} \\
I_B^{total} &= I_e^B + I_a^B + \sum_{L=1}^{N_L} \left\{ I_d^B + I_s^B \right\}
\end{aligned} \tag{3.6}
$$

Expandindo a equação acima ficamos com:

$$
\begin{aligned}
I_R^{total} &= k_e^R + k_a^R I_a^R + \sum_{L=1}^{N_L} \left\{ I_L^R k_d^R (\vec{N} \cdot \vec{L_L}) + k_s^R I_L^R (\vec{R_L} \cdot \vec{V}) \right\} \\
I_G^{total} &= k_e^G + k_a^G I_a^G + \sum_{L=1}^{N_L} \left\{ I_L^G k_d^G (\vec{N} \cdot \vec{L_L}) + k_s^G I_L^G (\vec{R_L} \cdot \vec{V}) \right\} \\
I_B^{total} &= k_e^B + k_a^B I_a^B + \sum_{L=1}^{N_L} \left\{ I_L^B k_d^B (\vec{N} \cdot \vec{L_L}) + k_s^B I_L^B (\vec{R_L} \cdot \vec{V}) \right\}
\end{aligned} \tag{3.7}
$$

3.5.16 Efeitos de Transparência

A implementação dos efeitos de transparência foi feita utilizando um modelo linear onde cada vértice possui um valor de opacidade α que será ou não utilizado para realizar a mistura de cores. Este modelo não leva em conta efeitos de refração. O valor de opacidade α varia entre $0,0$ e $1,0$, correspondendo a um pixel completamente opaco ou a um pixel completamente transparente, respectivamente. O cálculo é feito utilizando os pares $P_i \{Z_i, \alpha_i\}$ e $P_j \{Z_j, \alpha_j\}$ correspondentes ao pixel armazenado no buffer RGB (e no *frame buffer*) e ao pixel candidato à esta posição na tela e no buffer respectivamente. Nestas condições as seguintes situações são possíveis:

1. $Z_i < Z_j$, P_i encontra-se mais próximo do observador que P_j

   (a) $\alpha_i = 0$, P_i é completamente opaco e P_j é descartado.

   (b) $0 \leq \alpha_i \leq 1$, P_i é semi transparente e a cor resultante é a combinação linear

   $$I = I_i \cdot \alpha_i + I_j \cdot \alpha_j \tag{3.8}$$

   O valor de Z_i é mantido inalterado e α_i é substituido por

   $$\alpha_i^{novo} = \frac{\alpha_i + \alpha_j}{2} \tag{3.9}$$

   no buffer de transparência.

(c) $\alpha_i = 1$, P_i é completamente transparente, a cor de P_j substitui P_i, Z_i e α_i são mantidos inalterados.

2. $Z_i > Z_j$, P_j encontra-se mais próximo do observador que P_i

(a) $\alpha_j = 0$, P_j é completamente opaco e P_i é substituído por P_j no buffer RGB, Z_j substitui Z_i no *frame buffer* e α_j substitui α_i no buffer de transparência.

(b) $0 \leq \alpha_j \leq 1$, P_j é semi transparente e a cor resultante é a combinação linear

$$I = I_i \cdot \alpha_i + I_j \cdot \alpha_j \tag{3.10}$$

o valor de Z_i é substituido por Z_j e α_i é substituido por

$$\alpha_i^{novo} = \frac{\alpha_i + \alpha_j}{2} \tag{3.11}$$

no buffer de transparência.

(c) $\alpha_j = 1$, P_j é completamente transparente, a cor de P_i permanece no buffer RGB, Z_i é substituido por Z_j e α_i por α_j

Todo o processamento descrito é realizado pela função `transp_evaluate()`:

```
void
transp_evaluate(izb,alpha,r,g,b)
int izb;          /* Valor Zj */
int alpha;        /* Valor de alphaj */
float *r,*g,*b;   /* Cores resultantes */
```

Este modelo de transparência, embora simples, é adequado para a produção de imagens e computacionalmente rápido e eficiente. Vários refinamentos são possíveis através de modificações e adições à função `transp_evaluate()`.

A figura 3.12 sumariza as etapas de processamento implementadas nesta biblioteca descritas acima.

3.5.17 Formatos de Arquivos de Saída

Um dos objetivos da adição de um buffer RGB à biblioteca VoglZ foi a possibilidade de salvar em arquivos as imagens geradas pelos programas. A disponibilidade destas imagens em arquivos permite o seu pos-processamento por outros programas e a impressão em uma variedade de impressoras. Atualmente existe mais de uma centena de formatos de arquivos gráficos [33], neste

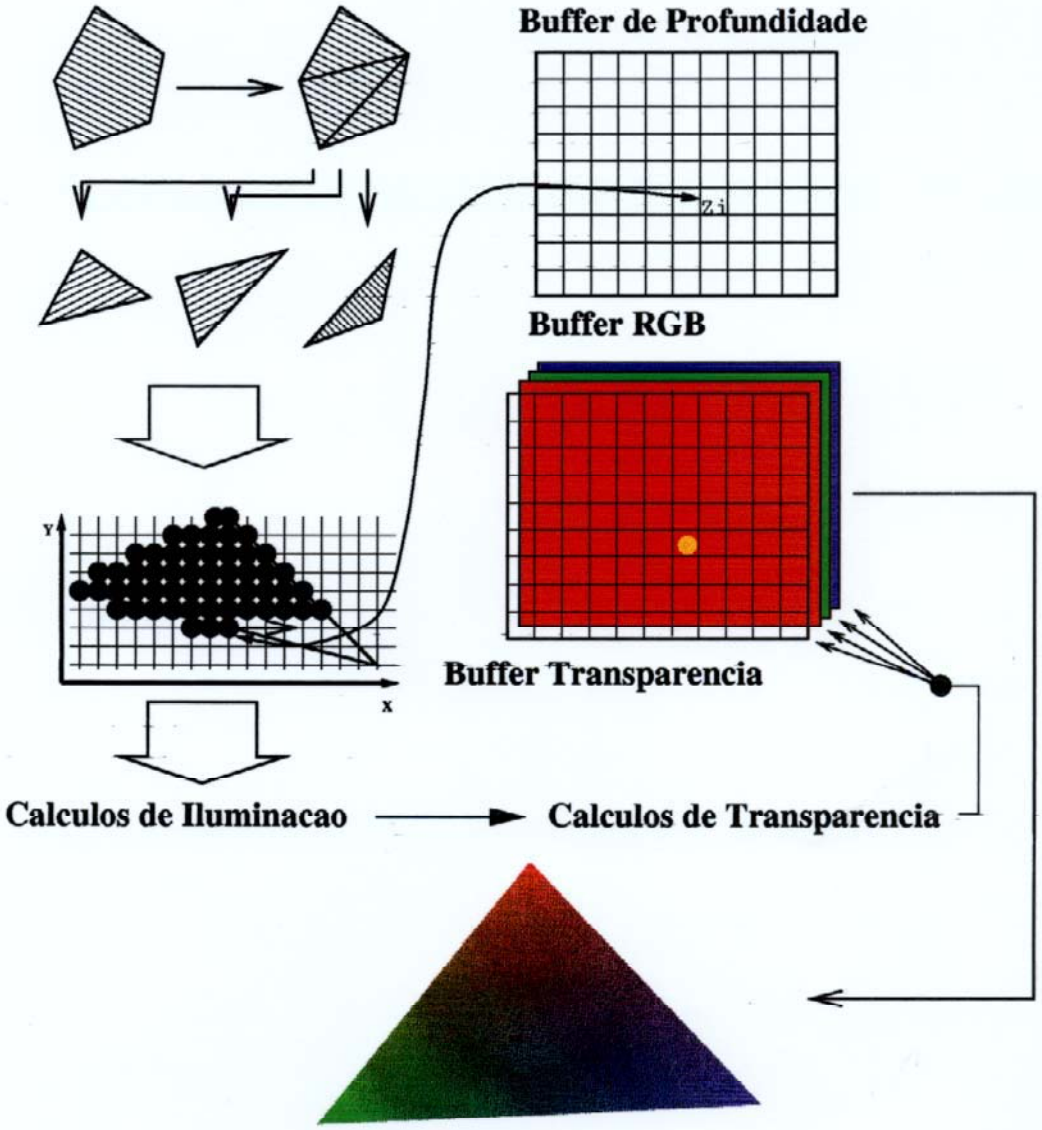


Figura 3.12: Processamento de vértices pela VoglZ

trabalho foram escolhidos os formatos *PPM* (*PortablePixMap*), *AdobePostScript*^{TM} [31,32] e NCSA *HDF* [34].

O formato *PPM* é domínio público e consiste de um arquivo binário com um cabeçalho identificando o formato, a imagem e suas dimensões, seguido de um vetor com os componentes RGB de cada pixel:

```
P6
#Image rendered with VoglZ 3.4\n
DimensaoX DimensaoY ValorMaximo
R1 G1 B1 R2 G2 B2 R3 G3 B3 ...
```

A principal característica do *PPM* é ser um formato não destrutivo (ao contrário de outros formatos como *GIF* e *JPEG*) de 24 bits que conserva integralmente todas as informações da imagem.

O formato *AdobePostScript*^{TM} foi desenvolvido para a criação de páginas a serem impressas. Este formato consiste, na verdade, de uma linguagem de programação (como C e Fortran) extremamente versátil permitindo a combinação de imagens, textos e desenhos em uma mesma página. Por ser uma linguagem de programação, o *AdobePostScript*^{TM} pode ser utilizado livremente e, por ser gravado em formato ASCII, torna simples a sua edição manual, caso necessário e a sua distribuição eletrônica.

O formato *HDF* (*HierarchicalDataFormat*), descrito mais detalhadamente em 4.4, caracteriza-se por ser um formato binário independente da arquitetura de computador com capacidade de armazenar, em um mesmo arquivo, imagens, dados numéricos, tabelas, texto, etc. Neste trabalho ele foi utilizado para geração de arquivos de dados em computadores com um tipo de arquitetura e transferi-los para processamento em outros de arquitetura diversa.

3.6 Programas Desenvolvidos e Resultados Obtidos

3.6.1 Biblioteca de Objetos Químicos

Utilizando a biblioteca VoglZ foi desenvolvida uma biblioteca de objetos químicos, isto é, um conjunto de funções primitivas para a criação de representações moleculares. Estas primitivas consistem de geradoras de objetos sólidos como esferas, cilindros e de objetos variáveis como curvas de contorno, isosuperfícies, planos, superfícies, etc.

A tabela 3.1 sumariza estas funções:

Função	Operação
`boundbox()`	Gera uma caixa contendo todas as moléculas
`contour()`	Gera n curvas de contorno no plano (x, y)
`cramp()`	Gera uma rampa de cores em um intervalo de valores $f(x,y,z)_{min}, f(x,y,z)_{max}$
`cyl()`	Gera um cilindro de raio r, cores c_1, c_2 entre os pontos (x_1, y_1, z_1) e (x_2, y_2, z_2) e os respectivos vetores normais
`gplane()`	Extrai um plano de dados passando por z_i de um cubo de dados $f(x,y,z)$
`isosurf()`	Extrai uma isosuperfície com valor v de um cubo de dados $f(x,y,z)$ de dimensões n_x, n_y, n_z
`makemol()`	Constroi todas as representações $(wireframe, stick, ball\&stick, CPK)$ de uma molécula a partir de suas coordenadas e número atômicos
`readhdf()`	Lê um arquivo HDF de dados contendo coordenadas atômicas, densidades eletrônicas, etc
`readmpc()`	Lê arquivos em formato mopac6
`readpsi88()`	Lê arquivos binários em formato PSI88
`readxyz()`	Lê arquivos texto de coordenadas atômicas em formato x, y, z
`sphere()`	Gera tetraedros, octaedros e esferas de raio r, cor c e os respectivos vetores normais
`sph_gen()`	Gera uma esfera de raio r, cor c com os respectivos vetores normais utilizando subdivisão angular
`triedrum()`	Gera um triedro centrado na origem do sistema molecular
`z2name()`	Obtém o símbolo de um elemento a partir de seu número atômico

Tabela 3.1: Biblioteca de Objetos Químicos

Bloco de Memória	Bytes/elemento	Consumo de Memória (*kbytes*)
Vértices	54000	527 kbytes
Z buffer	4	1914 kbytes
RGB & α buffer	4	1914 kbytes

Tabela 3.2: Consumo de memória para uma imagem 700 por 700 pixels com 5000 triângulos

3.6.2 Considerações Sobre Desempenho

Três fatores afetam o desempenho da VoglZ, a velocidade do processador, a velocidade do sistema de memória (VM, RAM e cache) e a velocidade do dispositivo gráfico de saída empregado.

A equação 3.7 ilustra o número de operações de ponto flutuante necessárias para a obtenção das três componentes de cor de cada pixel, a estas operações devem ser adicionadas as operações de interpolação e normalização de vetores. Estas operações exigem a utilização de sistemas computacionais dotados de unidades aritméticas de ponto flutuante.

A análise da figura 3.12 mostra que o desenho de cada pixel na tela implica em três acessos à memória principal, um para a manipulação aritmética dos vértices, outro para o teste do buffer de profundidade e outro para acesso ao buffer de transparência e de cores. Considerando uma imagem típica de dimensões 700 por 700 pixels contendo em torno de 5000 triângulos o consumo de memória é expresso pela tabela 3.2

Em função dos tamanhos dos blocos de memória e da frequência com que são acessados o desempenho da biblioteca VoglZ depende fortemente da quantidade de memória física disponível, da implementação de hardware do sistema de memória (RAM e cache) e do suporte do sistema operacional à alocação dinâmica de memória e sua manipulação. Para obtenção de um desempenho razoável é necessária a utilização de computadores com no mínimo 16 Mbytes de memória física disponível, um sistema de memória virtual avançado e um sistema de controle de cache e memória física eficiente.

O último fator influenciando o desempenho está relacionado à tecnologia utilizada pelo dispositivo gráfico de saída. Independentemente das suas caraterísticas (8, 16 ou 24 bits de profundidade) o dispositivo gráfico será acessado constantemente para o envio dos pixels calculados para o frame buffer, a velocidade com que isto é realizado depende da tecnologia de barramento empregada. Neste trabalho foram utilizadas as tecnologias ISA, SBUS e PCI. A tecnologia ISA emprega um barramento de 16 bits operando a uma velocidade de 4,77 MHz que mostrou-se inaceitável para este tipo de trabalho. Os barramentos SBUS (32 bits a 25 MHz) e PCI (32 bits a

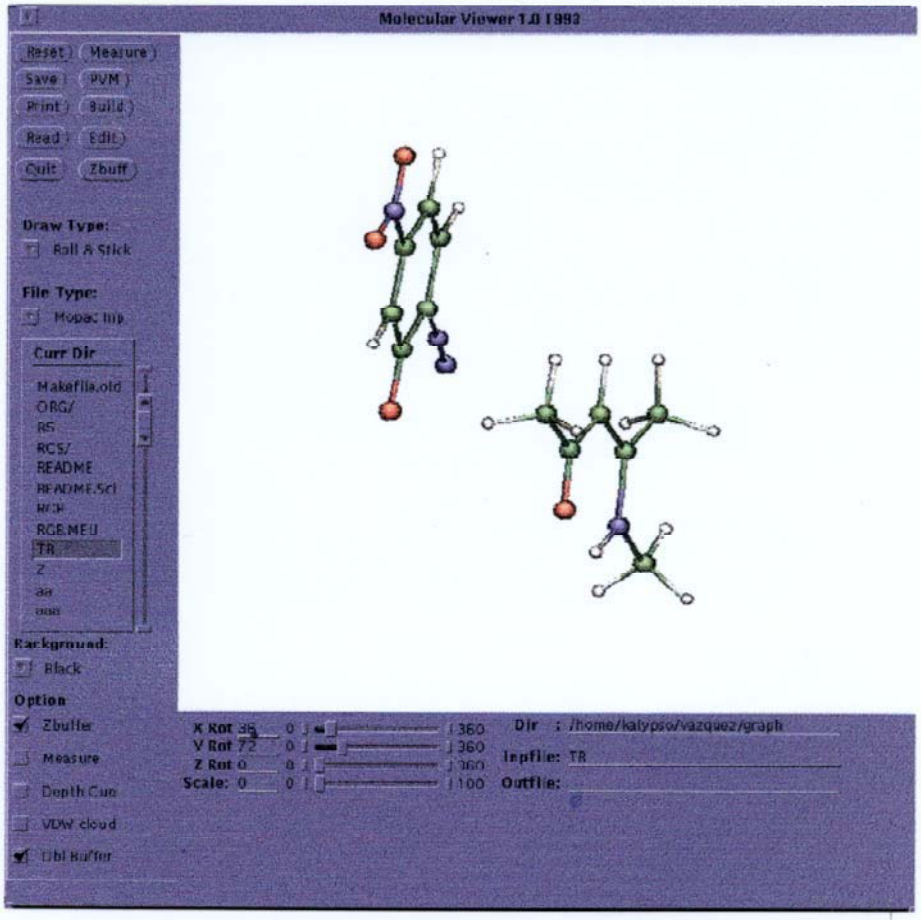


Figura 3.13: Programa de visualização molecular **MV** desenvolvido com a biblioteca VoglZ

33MHz), por outro lado, encontram-se dentro da largura de banda exigida pelos processadores atuais.

3.6.3 MV, a simple Molecular Viewer

Utilizando a biblioteca de objetos químicos juntamente com a biblioteca VoglZ, foi desenvolvido o programa **MV** (*A simple Molecular Viewer*) com uma interface de usuário baseada no XView Toolkit 25. As figuras a seguir ilustram os resultados obtidos com o **MV** durante os estudos de alguns pesquisadores do Instituto de Química da Unicamp. A figura 3.13 ilustra a aparência geral do programa **MV** e os principais recursos por ele disponibilizados [29]. A figura 3.14 mostra um estudo de estado de transição realizado

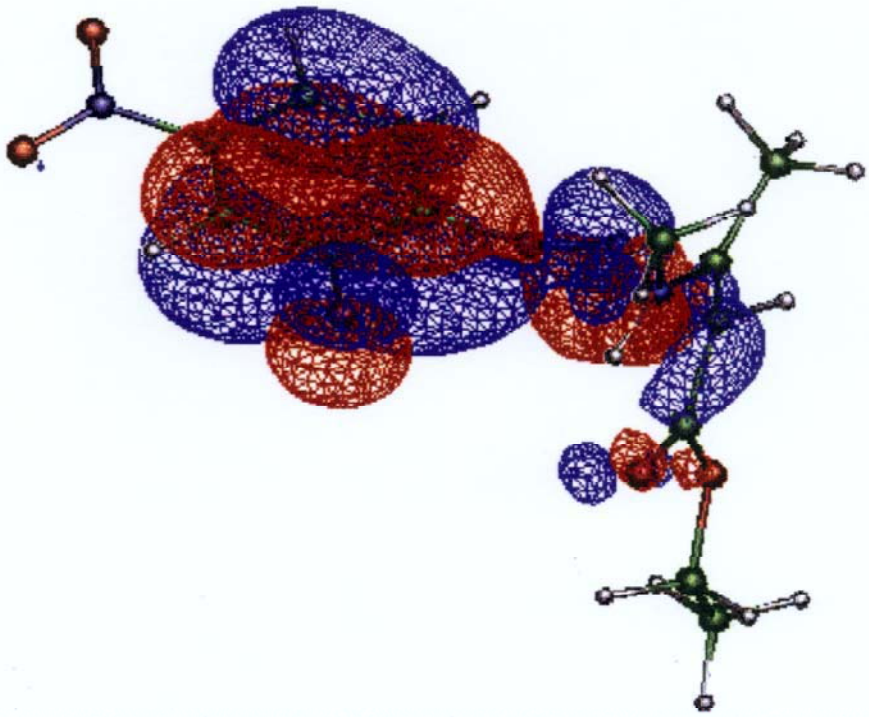


Figura 3.14: Estudo de um estado de transição visualizado com auxílio da biblioteca VoglZ

por Figueiredo [30,35]. Esta figura ilustra a utilização da função `isosurf` para a visualização dos cubos de dados gerados pelo programa PSI88 [36], bem como a utilização intensiva do modelo de iluminação Phong e do buffer de profundidade para a remoção de linhas e superfícies ocultas.

Na figura 3.15, um estudo de reatividade conduzido por Santos [37,38], podem ser observados os efeitos obtidos quando se utiliza o buffer de transparência para a visualização de imagens complexas. É importante notar na figura 3.15 que, apesar da simplicidade do modelo de transparência utilizado, os resultados obtidos realçam fortemente a percepção espacial sem introduzir artifícios na imagem.

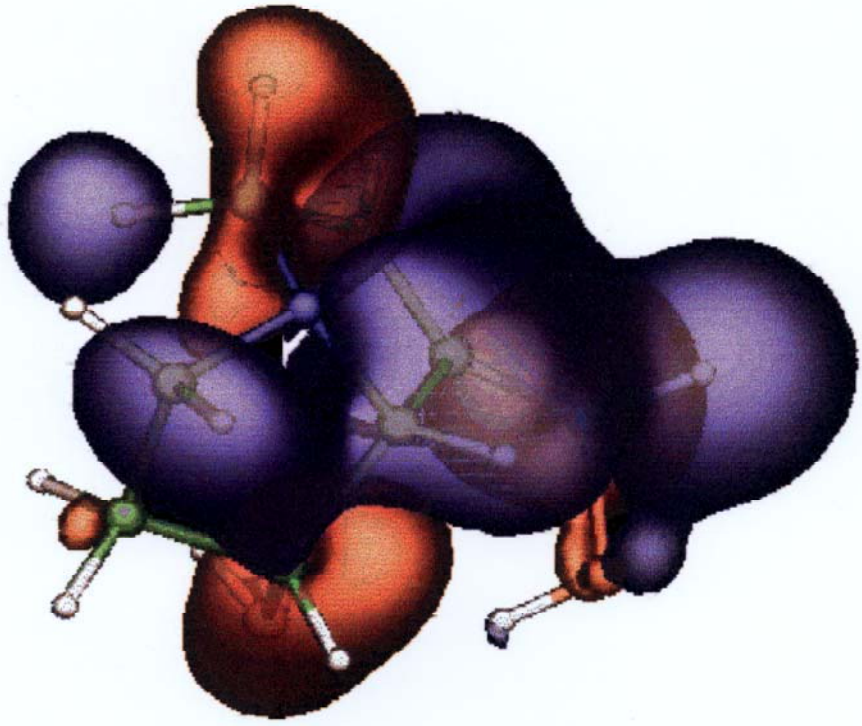


Figura 3.15: Efeitos de transparência aplicados a uma isosuperfície no estudo de reatividade

3.6.4 Programa GLPSI

O programa **GLPSI** foi desenvolvido a partir de 1994 como uma evolução do **MV**. Este programa abandona o XView Toolkit em troca do GLUT [20] e substitui a biblioteca VoglZ pela biblioteca Mesa [24] à qual foram adicionadas as extensões desenvolvidas na VoglZ. Um dos objetivos desta migração foi incorporar a interface de programação da OpenGL para poder ampliar o número de plataformas computacionais suportadas. A figura 3.16 ilustra a aparência geral do **GLPSI** e os principais recursos oferecidos.

Entre as capacidades adicionadas ao **GLPSI** destacam-se

1. A codificação de propriedades de interesse utilizando escala de cores, como ilustra a figura 3.17

2. A análise interativa de geometrias moleculares, (medidas de distâncias, ângulos, etc);

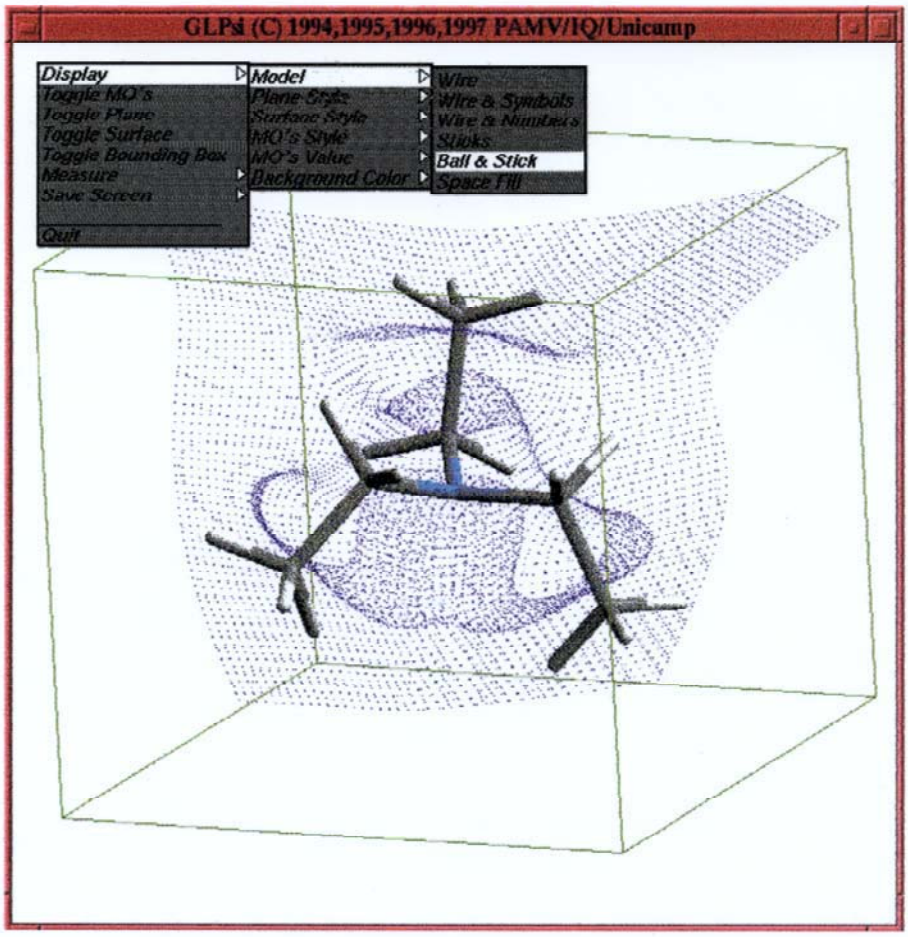


Figura 3.16: Programa **GLPSI**

3. A geração interativa de planos e superfícies associados a propriedades moleculares calculadas

4. A geração de arquivos em formato PPM e PostScript com os resultados para armazenamento ou impressão.

5. A adição do algoritmo de Goodsell & Olson [39] para a geração de gravuras preto e branco utilizando o buffer RGB e o buffer de profundidade (figura 3.18);

6. Suporte à OpenGL e ao protocolo de rede GLX quando disponíveis.

Figura 3.17: Visualização de isosuperfícies, codificação de cores e desenho de superfícies utilizando o GLPSI.

3.7 Discussão e Conclusões

Os resultados obtidos neste trabalho mostram que, utilizando os recursos computacionais de software e hardware disponíveis atualmente, é possível o desenvolvimento de técnicas e programas avançados de visualização molecular. A utilização de linguagens e padrões bem definidos de programação é essencial para que se obtenha a maior independência possível de hardware. Neste caso isto foi obtido através da adoção de padrões como o compilador GNU C, o ambiente X11 e o sistema operacional Unix.

O desempenho computacional obtido depende fortemente de fatores tecnológicos ainda em evolução, mesmo assim, considerando que este trabalho iniciou seu desenvolvimento com desempenho aceitável [29] em equipamentos modestos como PC 386 DX 40MHz com 8 Mbytes de RAM, não há limitação tecnológica (de software ou hardware) atualmente para a sua continuidade ou aprimoramento, ao contrário, a evolução dos dispositivos gráficos de saída nos últimos anos tem convergido na direção de prover suporte mais eficiente às técnicas aqui desenvolvidas.

No que tange a tradição histórica do uso da linguagem Fortran pelos químicos, no entanto, o desenvolvimento da VoglZ mostrou a sua total inadequação às necessidades de programação da visualização molecular. Mesmo

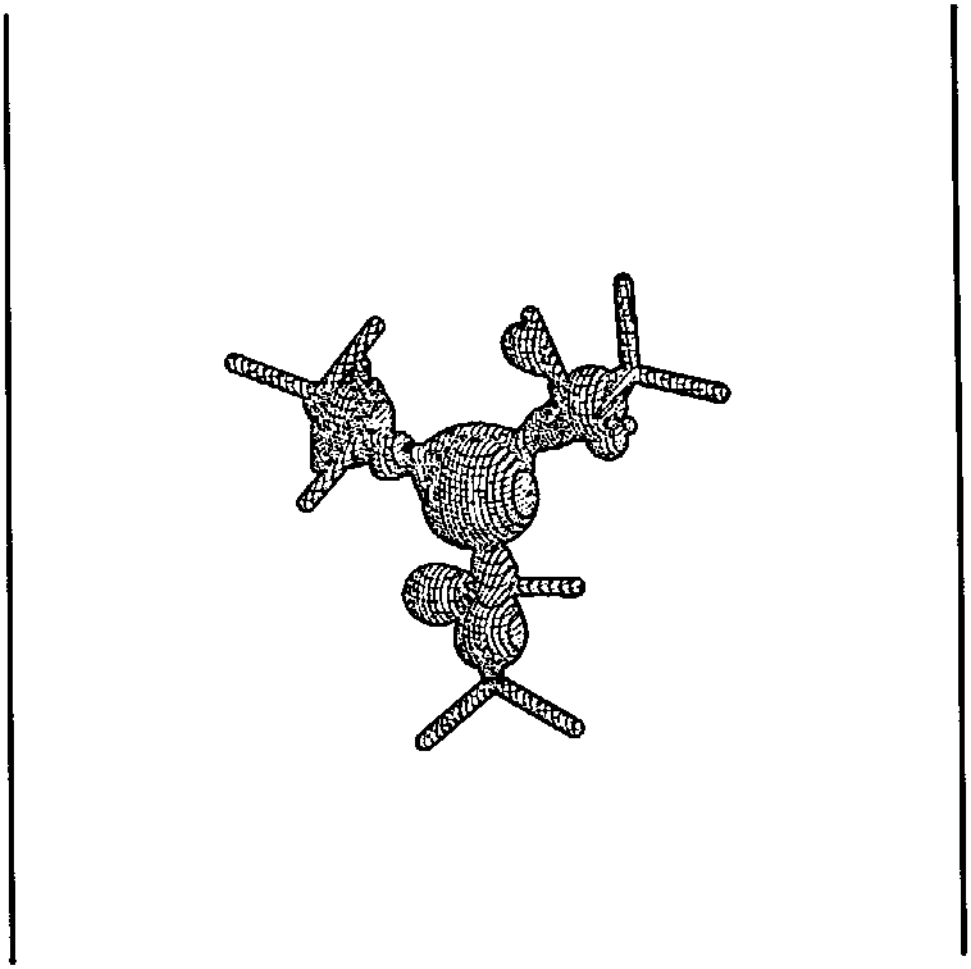


Figura 3.18: Exemplo de gravura preto e branco produzida pelo programa GLPSI

em suas revisões mais recentes como Fortran90 ou Fortran95, esta linguagem não dispõe, de forma eficiente, dos recursos da linguagem C para a definição e manipulação de estruturas e dados relativamente complexos que surgem na visualização molecular e nem da sua transportabilidade entre processadores e sistemas operacionais diferentes. Em outras palavras, a visualização científica molecular pressupõe o domínio e o uso da linguagem C como base de desenvolvimento, da mesma forma que, em outras áreas da química como instrumentação e quimiometria as linguagens *assembly* e $Matlab^{TM}$ tem sido apontadas como mais apropriadas.

Do ponto de vista dos objetivos inicialmente propostos para este trabalho, a criação de um sistema de visualização científica molecular pode-se concluir

que os mesmos foram atingidos. A biblioteca de objetos moleculares e a biblioteca VoglZ com suas extensões adicionadas à Mesa fornecem um ambiente de programação e desenvolvimento para aplicações de visualização científica em química, a sua transportabilidade entre diversos sistemas computacionais é garantida pelo uso de padrões bem definidos e disponíveis gratuitamente.

3.8 Bibliografia

1. *Angell, I. O.*, **A Pratical Introduction to Computer Graphics**, McMillan Publishers Ltd, 1981, Londres.

2. *Vazquez, P. A. M., Guadagnini, P. H.*, **Biblioteca Calcomp Para Workstations Sun Sparc**, 14ª Reunião Anual da Sociedade Brasileira de Química, Caxambu, MG, 1991

3. *Vazquez, P. A. M., Garcia, M., Custódio, R.*, **MindTool for X Window System**, 17ª Reunião Anual da SBQ, Caxambu, maio de 1994.

4. *Jonhson, C. K.*, **Report ORNL-5138**, Oak Ridge National Laboratory, Oak Ridge, Tenesse, 1976,EUA.

5. *Clark, J.*, **Roots And Branches of 3-D**,Byte, 153, **17** (1992) 5

6. *Glassner, A. S.*, **Graphics Gems**, Academic Press, Boston, 1990.

7. *Rogers, D. F., Adams, J. A.*, **Mathematical Elements for Computer Graphics**, McGraw-Hill, 1990.

8. *Rogers, D. F.*, **Procedural Elements for Computer Graphics**, McGraw-Hill, 1985.

9. *Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F.*, **Computer Graphics, Principle and Practice**, Addison-Wesley, Reading, MA, EUA, 2ed, 1992.

10. *Nye A.*, **Xlib Reference Manual for Version 11**, O'Reilly & Associates,Inc., Sebastopol, 1990, EUA.

11. *Segal, M., Akeley, K*, **The OpenGL Graphics System: A Specification (Version 1.0)**, Silicon Graphics, Inc. 1993.

12. *MIT X Consortium Staff*, **X Window System, Version 11, Release 5, Release Notes**, MIT Laboratory for Computer Science, Cambridge, 1991, EUA

13. *Hess M. et all.*, **PEX-SI Porting Guide and Implementation Details**, Sun Microsystems, Inc., 1991.

14. *Hopgood, F. R.A., Duce, D. A.*, **A Primer For PHIGS**, John Wiley & Sons, Chichester, 1991, Inglaterra.

15. *Gaskins T.*, **PHIGS Programming Manual**, O'Reilly & Associates,Inc., Sebastopol, 1991, EUA.

16. *Hess M. et all.*, **PEX-SI User's Guide**,Sun Microsystems, Inc., 1991.

17. *Hui S.*, **GLX Extension For OpenGL, Protocol Specification 1.2**, Silicon Graphics, Inc. 1995.

18. *Karlton P.*, **OpenGL Graphics with the X Window System**, Silicon Graphics, Inc. 1993.

19. *Smith, K. P.*, **The OpenGL Graphics System Utility Library**, Silicon Graphics, Inc. 1993.

20. *Kilgard M. J.*, **The OpenGL Utility Toolkit (GLUT) Programming Interface**, Silicon Graphics, Inc. 1995.

21. *Neider J., Davis, T., Woo M.*, **OpenGL Programming Guide**, Addison-Wesley, 1995.

22. *OpenGl Architeture Review Board*, **OpenGL Reference Manual**, Addison-Wesley, 1995.

23. *Akin, A.*, **Analysis of PEX5.1 and OpenGL 1.0**, Silicon Graphics, Inc. 1992.

24. **Mesa** Brian Paul, *brianp@ssec.wisc.edu* , ©1996,1997, disponível em *ftp://iris.ssec.wisc.edu/pub/Mesa*

25. *Heller, D.*, **XView Programming Manual**, O'Reilly & Associates, Inc., Sebastopol, 1991, EUA.

26. *Heller, D., Ferguson, P. M.* , **Motif Programming Manual**, O'Reilly & Associates, Inc., Sebastopol,1994,EUA.

27. *Kilgard, M. J.*, **OpenGL Programming for the X Window System**, Addison Wesley, Reading, MA, EUA, 1996.

28. **VOGL:** Department Of Engineering Computer Resources, Faculty Of Engineering, University of Melbourne, ©1991,1997, disponível em *ftp://gondwana.ecr.mu.OZ.AU*

29. *Vazquez, P. A. M.*, **Desenvolvimento de Algumas Extensões à Biblioteca VOGL Para Aplicação em Gráficos Moleculares**, V SIBIGRAPI, Recife, PE, 1992.

30. *Vazquez, P. A. M., Guadagnini, P. H., Figueiredo, L. J.*, **Algumas Aplicações do Processamento de Imagens na Visualização de Dados Químicos**, 1ª Mostra de Trabalhos da UNICAMP em Computação de Imagens, Universidade Estadual de Campinas, Campinas, SP, 1992.

31. *Adobe Systems, Inc.*, **Postscript Language Tutorial and Cookbook**, Addison-Wesley, 1985.

32. *Adobe Systems, Inc.*, **Postscript Language Reference Manual**, Addison-Wesley, 1985.

33. *Murray, J. D., van Ryper, W.*, **Encyclopedia of Graphics File Formats**, O'Reilly & Associates, Inc., Sebastopol, 1994, EUA.

34. *Software Development Group*, **NCSA HDF Specifications**, NCSA, University of Illinois at Urbana-Champaign, EUA, 1989.

35. *Figueiredo, L. J.* **Estudo**, Tese de Doutoramento, IQ Unicamp, 1994.

36. *Jorgensen* PSI 88

37. *Santos, M. G.* **Estudo**, Tese de Doutoramento, IQ Unicamp, 1995.

38. *Vazquez, P. A. M.*, **Visualization Improvement in VOGL By The Addition of Better Surface Description Through the Use of Phong Shading**, 17ª Reunião Anual da SBQ, Caxambu, MG, 1994.

39. *Goodsell, D. S., Olson, A. J.*, **Molecular Illustration in Black and White**, *J. Mol. Graphics*, 1992, Vol. 10, December.

Capítulo 4

Computação Química Distribuída

4.1 Introdução

Neste capítulo procuraremos analisar soluções que permitam que programas em geral, em C ou Fortran, possam executar distribuidamente de forma paralela ou cooperativa visando a otimização do uso de recursos e/ou a redução do tempo das tarefas computacionais.

As tentativas de obter alto desempenho computacional podem ser classificadas em dois grandes grupos [1,2], o desenvolvimento de processadores mais rápidos, com ciclos de processamento pequenos (300MHz) e arquiteturas mais eficientes ou o desenvolvimento de sistemas multiprocessados onde, utilizando um grande número de processadores com ciclos maiores de processamento (100-300 MHz), a aceleração é obtida através da divisão do trabalho, isto é, através da distribuição entre vários processadores (no mesmo computador ou não) de partes da tarefa global a ser realizada. Exemplos do primeiro caso são os supercomputadores de alta performance (Cray , NEC) que se caracterizam por serem de preço extremamente elevado, exigirem uma infraestrura sofisticada e se basearem em um modelo centralizado de computação.

No segundo caso é necessário que se faça uma classificação segundo três correntes tecnológicas: aquela que se baseia em um grande número de processadores operando de forma síncrona a mesma instrução sobre um conjunto igualmente grande de dados (*SIMD Single Instruction Multiple Data*); a que se baseia em um número grande de processadores independentes, operando de forma síncrona ou não, executando diferentes instruções sobre o conjunto de dados (*MIMD Multiple Instruction Multiple Data*) e, por último, aquela que faz uso de um número médio (16 a 64) de processadores fortemente aco-

plados, e operando de forma simétrica compartilhando periféricos, memória e recursos locais (*SMP Symetric Multiprocessing*). Mais recentemente graças ao desenvovimento notável das arquiteturas RISC (Reduced Instruction Set Computer) utilizadas em estações de trabalho e a popularização e amadurecimento das tecnologias de compiladores e de redes (acompanhadas de equivalente redução de custos) uma outra solução tornou-se viável para a aceleração de cálculos computacionais.

Através do uso de funções intrínsecas ao sistema operacional Unix (embora gradativamente estejam tornando-se disponíveis em outros sistemas) uma série de bibliotecas de comunicação foram criadas para transformar uma rede de estações de trabalho em um cluster computacional com um desempenho numérico aproximadamente igual a soma dos desempenhos individuais de cada componente da rede. O modelo de programação em clusters pode seguir um conceito extremamente semelhante àquele utilizado em arquiteturas MIMD, bastando apenas que o programa seja desenvolvido tendo em mente as limitações ou vantagens de uma rede de estações em relação aos sistemas multiprocessados. Da mesma forma pode-se optar por um modelo SMP/MIMD simulando, em um cluster, um computador SMP.

Entre as limitações impostas a utilização de clusters de estações de trabalho como opção a sistemas MIMD ou SMP destacam-se principalmente [2]:

- A elevada latência e baixa velocidade de transmissão dos meios físicos utilizados para montar as redes (Ethernet 10 Mbits/s, Token-Ring 16 Mbits/s, FastEthernet 100 Mbits/s FDDI 100Mbits/s latência típica 5 ms) quando comparados a latências da ordem de nanosegundos usuais em sistemas SMP ou MIMD.

- Da mesma forma, a natureza multiusuário destes equipamentos pode produzir desequilíbrios imprevisíveis em cada equipamento participante em função da sua utilização para outras tarefas.

As principais vantagens, por sua vez, são:

- A memória local maior (tipicamente 64 a 128 Mbytes contra 8 a32 Mbytes em sistemas MIMD);

- A existência de discos locais em cada processador com , teoricamente, a capacidade de armazenamento que se fizer necessária ao trabalho;

- A disponibilidade de periféricos especiais (aceleradores gráficos, processadores vetoriais, digitalizadores de vídeo, etc) permite adicionar

funcionalidades extras a um ou mais computadores do cluster, de acordo com as especifidades de um dado projeto, de forma aditiva ao cluster existente;

- Interoperabilidade, uma característica implícita do sistema Unix, que consiste na capacidade de se poder conectar, para a execução de uma mesma tarefa, equipamentos de fabricantes diferentes procurando explorar em cada um os recursos que dispõe (capacidade de armazenamento em disco, visualização, capacidade de memória, etc);

- Escalabilidade, isto é, um cluster de computadores pode ser dinamicamente configurado e dimensionado através da adição ou remoção de computadores de acordo com as necessidades do problema a ser analisado.

Os mecanismos disponíveis para a computação distribuída basciam-se nas formas possíveis de comunicação entre processos. Estas formas são descritas a seguir.

4.2 Comunicação Interprocessos no Unix

A comunicação interprocessos é a base da computação distribuída. É através dela que um processo pode iniciar ou encerrar outros processos no mesmo computador ou ao longo de uma rede e realizar a troca de dados necessária à execução de uma tarefa. Exemplos de comunicação entre processos executando em computadores diferentes interconectados por uma rede podem ser encontrados rotineiramente em serviços como correio eletrônico, compartilhamento de discos e impressoras. Os mecanismos básicos pelos quais os processos se comunicam no Unix são: pipes, FIFOs, mecanismos de memória compartilhada utilizando SysV IPC ou BSD *mmap* e sockets BSD. Destes mecanismos apenas o último contempla a comunicação entre processos executando em um mesmo computador ou executando em um grupo de computadores remotos.

4.2.1 Pipes

Pipes [3,4,5] consistem em canais unidirecionais de comunicação entre processos que executam em um mesmo computador e que guardam entre si uma relação hierárquica pai-filho. Para comunicação entre dois processos utilizando pipes, é necessário que o processo pai solicite a criação de um pipe e a seguir faça uma chamada à função `fork()` para iniciar a execução do

processo filho. A lógica do programa definirá a direção de comunicação a ser utilizada[4]. Pipes são utilizados amplamente em diversos sistemas operacionais em comandos de sistema (ex. `ls -l | more`) e devido as suas características não apresentam a flexibilidade necessária para aplicações científicas mais complexas.

4.2.2 FIFO

FIFOs, [3,4,5] ou pipes *First In, First Out*, são canais de comunicação aos quais foi associado um nome, da mesma forma que um nome de arquivo, e que permitem a comunicação entre processos sem que haja, entre os mesmos, uma relação hierárquica. Mais especificamente, um dado processo cria um FIFO e associa um nome a ele, a seguir este mesmo processo abre este FIFO como se estivesse abrindo um arquivo e passa a realizar operações de leitura/escrita. Um segundo processo que deseje comunicar-se com o primeiro realiza, também a operação de abertura de arquivo, e através de operações recíprocas de leitura/escrita realiza a troca de dados desejada. Exemplos de comunicação entre processos utilizando FIFOS podem ser encontrados no sistema `lpr/lpd` de impressão dos Unix baseados na linhagem de Berkeley. Embora mais flexíveis que os pipes, os FIFOS carecem da flexibilidade e generalidade necessárias à computação científica distribuída.

4.2.3 Memória compartilhada

Uma outra forma de comunicação entre processos executando em sistemas Unix é através de memória compartilhada (*shared memory*). Neste método um bloco de memória RAM pode ser acessado por dois ou mais programas executando em um mesmo computador. Duas implementações tem sido amplamente empregadas: SysV SHM [3,4] e BSD `mmap()` [5,6,7]

A implementação SysV faz uso de páginas da memória virtual do sistema operacional que podem ser acessadas de forma coordenada por vários processos compartilhando um mesmo conjunto de dados. Apesar de eficiente este método não possui a padronização e suporte universal entre plataformas computacionais diferentes e visa o compartilhamento local de memória entre processos e nao o compartilhamento em rede.

4.2.4 mmaping

O método de comunicação entre processos utilizando **`mmap()`** consiste em mapear, no espaço de variáveis de um ou mais processos, um dispositivo ou arquivo [7]:

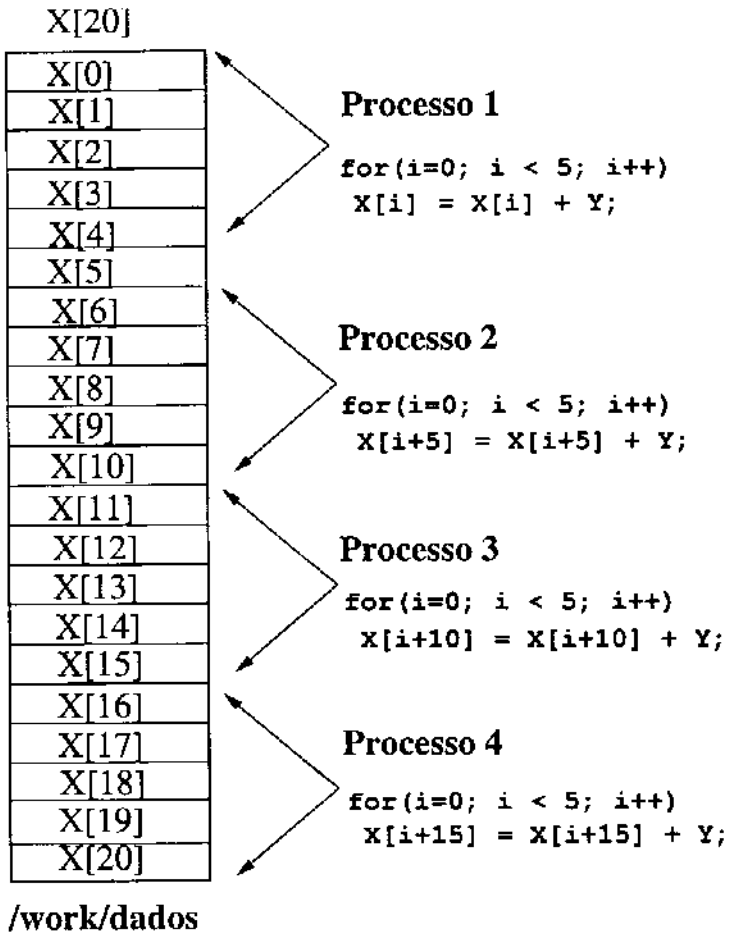


Figura 4.1: Programação distribuída utilizando `mmap`

Em outras palavras, uma matriz `x[1000]` usualmente mapeada no espaço de endereçamento de memória do programa passa a residir em um arquivo cujas dimensões dependem apenas dos recursos de armazenamento disponíveis. Este arquivo fica sob controle do sistema operacional podendo aproveitar em maior ou menor grau as suas características de gerenciamento. Uma vez que os elementos `x[i]` não residem mais no espaço privativo de um dos processos em execução os mesmos passam a estar disponíveis para que outros processos possam acessá-los e/ou alterá-los, permitindo a distribuição do processamento.

Caso este arquivo esteja disponível à demais máquinas de um cluster utilizando um sistema de arquivos distribuídos em rede como o NFS [8] o processamento poderá ser realizado de forma cooperativa por todos os componentes

do cluster. É importante frisar que o `mmaping` não consiste em operações de leitura e escrita (I/O) simultâneas por vários processos mas realmente no mapeamento de um dispositivo ou arquivo **diretamente** no mapa de memória de um ou mais processos. Esta característica acarreta dificuldades na interoperação de processos quando o arquivo `mmaped` não encontra-se situado localmente mas, por exemplo, montado via NFS uma vez que a especificação inicial deste protocolo não garante [8] o sincronismo perfeito entre servidor e clientes. Visando contornar esta situação Minich [9] integrou fortemente o protocolo NFS ao sistema de memória virtual , neste trabalho utilizamos [10] o MNFS proposto por Minich para a execução de processos distribuídos entre um cluster de computadores PC 486 e comparamos a sua eficiência e viabilidade em química em relação a outros métodos conforme será descrito e discutido posteriormente.

O uso de `mmaping` mostra-se vantajoso, no entanto, pela sua simplicidade de programação, após a operação de mapeamento do arquivo ou dispositivo em memória a operação sobre as variáveis é feita de forma usual:

```
x[i,j] = a*y[i,j]+b
```

A sua eficiência, no entanto, só é realizada em computadores monoprocessados onde os processos compartilhando o mesmo conjunto de dados não concorram por recursos semelhantes do sistema (ex. unidade de processamento de ponto flutuante), ou em sistemas SMP, isto é, em computadores com vários processadores compartilhando entre si a mesma memória física.

Implementações via redes, como o MNFS, embora específicas e em alguns casos de uso restrito a alguns Unix têm atraido o interesse de pesquisadores conforme descrevem Souto e Stark [11].

4.2.5 Sockets BSD

Desenvolvidos pela Universidade da Califórnia, Berkeley, e disponíveis a partir da versão 4.1cBSD [6,12] para computadores VAX por volta de 1982, os sockets consistem numa abstração baseada em funções primitivas de comunicação implementadas no sistema operacional e de uma interface de programação de aplicações (ou API, *Application Programming Interface*) acessível ao usuário. Os sockets BSD foram criados com os seguintes objetivos [12,13]:

- Transparência, a comunicação independe dos processos residirem ou não no mesmo computador;
- Compatibilidade e interoperabilidade entre diferentes sistemas operacionais;

- Independência das características físicas da rede;
- Independência do protocolo de comunicação entre os computadores que compõe a rede;

Um socket é uma das extremidades de um canal virtual de comunicação entre dois ou mais processos, executando ou não no mesmo computador, que pode ser criado e manipulado para leitura e escrita da mesma forma que um arquivo de dados. Ao contrário destes, no entanto, um socket só existe durante o período de intercomunicação dos processos. Posteriormente à versão inicial, liberada em 1983, a implementação de sockets sofreu revisões a cada nova versão do Unix de Berkeley: 4.2BSD em 1984, 4.3BSD em 1986, 4.3BSD-Tahoe em 1988, 4.3BSD-Reno em 1990 [7] Em 1991, Berkeley disponibilizou abertamente uma versão de seu Unix que tornou-se conhecida como Net/2. Esta versão, assim como o 4.3BSD-Tahoe de 1988, foram utilizadas amplamente por vários fabricantes de sistemas operacionais como base e referência para a sua propria implementação de sockets e de comunicação via rede utilizando o protocolo IP . Posteriormente, Berkeley ainda lançou mais duas versões do seu Unix, denominadas 4.4BSD-Lite (1994) e 4.4BSD-Lite2 (1995) [7,14]. Esta evolução histórica de implementações fez com que nem todos os fabricantes tenham evoluido ou aprimorado suas implementações de sockets e comunicação de rede na mesma escala que Berkeley. Desta forma neste trabalho, e visando a transferibilidade de programas entre sistemas, segue-se, conforme recomendado por Stevens [14,15], as interfaces de programação, os padrões e implementação descritos e utilizados para a versão Net/2 de sockets e comunicação via rede.

4.2.6 Interface de Programação de Aplicações

As funções disponíveis para programação utilizando sockets são [13,16,17] sumarizadas na tabela 4.1.

A existência das funções `listen()`, `accept()` e `connect()` evidenciam as características de um modelo de programação cliente-servidor para a utilização de sockets em computação distribuída [18]. A figura 4.2 ilustra melhor esta caracterítica e as etapas envolvidas no estabelecimento de um canal ou circuito virtual de comunicação entre dois processos utilizando sockets.

Em linguagem C, um socket é criado através da função:

```
s=socket(domínio, tipo, protocolo)
```

onde *domínio* define domínio de comunicação a ser empregado , conforme a tabela 4.2 [13,16].

Operação	Função	Variante 1	Variante 2
Criação	socket()		
Associação de Nome	bind()		
Seleciona Conexão	listen()		
Accita Conexão	accept()		
Realiza Conexão	connect()		
Leitura de dados	read()	recv()	recvfrom()
Escrita de dados	write()	send()	sendto()
Encerra Conexão	close()	shutdown()	

Tabela 4.1: Interface de Programação de Aplicações com sockets BSD

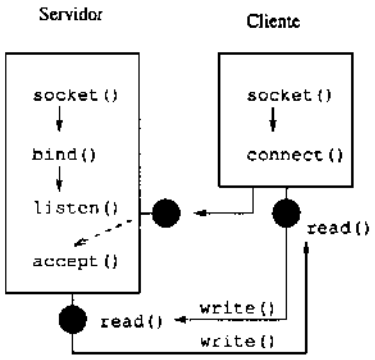


Figura 4.2: Sequência de eventos no estabelecimento de um circuito virtual utilizando BSD sockets.

O *tipo* de socket a ser criado define a semântica de comunicação a ser empregada para comunicação entre os processos conforme ilustrado na tabela 4.3 [7]

Por último, *protocolo* define qual protocolo do domínio e semântica escolhidos deve ser empregado. Via de regra apenas um protocolo existirá para uma dada seleção embora seja possível que mais de um esteja disponível. A definição de todos os parâmetros envolvidos na criação de um socket é bastante dependente da implementação de Unix empregada, nem todas as versões do Unix suportam todas as variantes possíveis. Do ponto de vista da computação distribuída em química apenas os domínios PF_LOCAL e PF_INET, que permitem a comunicação entre processos executando em um mesmo computador ou processos entre computadores conectados à Internet, respectivamente, é que são de interesse prático.

Conforme ilustrado na figura 4.2 o modelo cliente-servidor utilizado pelos

Domínio	Família de protocolos
PF_LOCAL	Protocolos internos (locais) ao computador
PF_INET	Protocolos Internet (ARPA)
PF_ISO	Protocolos ISO
PF_CCITT	Protocolos ITU-T (X.25, etc)
PF_NS	Protocols XNS da Xerox

Tabela 4.2: Famílias de Protocolos definidas para sockets BSD

Tipo	Semântica de comunicação do canal
SOCK_STREAM	bidirecional, conectado,sequencial,confiável
SOCK_DGRAM	não conectado, mensagens de tamanho fixo
SOCK_RAW	transferencia de dados brutos, uso reservado
SOCK_SEQPACKET	sequenciamento definido, conectado, tamanho fixo
SOCK_RDM	entrega confirmada

Tabela 4.3: Semânticas de Comunicação para sockets BSD

sockets exige que, após a sua criação, este tenha associado a si um nome que permita a sua localização processos clientes, isto é realizado através da função [16] :

```
bind(s, struct sockaddr endereço,tamanho)
```

Na função acima `s` é o descritor retornado pelo sistema operacional, quando da chamada da função `socket()`, que identifica o socket para o processo servidor e a estrutura `endereço` definida como:

```
struct sockaddr {
        u_char  sa_len;         /* tamanho total */
        u_char  sa_family;      /* familia */
        char    sa_data[14];    /* valor do endereco */
};
```

depende da família de protocolos empregada [16,17]. No caso da família PF_LOCAL, `endereço` definirá um nome de um socket que estará localizado na hierarquia do sistema de arquivos do Unix da mesma forma que um arquivo convencional. No caso da família PF_INET, definirá os parâmetros de identificação do protocolo IP conforme será descrito posteriormente. Em suma a função `bind()` cria uma associação do tipo:

{*protocolo,endereço-local,processo-local,endereço-remoto,processo-remoto*}

A partir deste instante, o processo criador do socket poderá operar no papel de servidor observando o socket a espera de conexões, esta condição é ativada através da função abaixo.

```
listen(s,max)
```

Onde `s` define o socket e `max` define o número máximo de conexões pendentes aceitadas pelo servidor. Ou seja, caso o valor de `max` seja excedido durante a execução do processor o cliente receberá uma notificação de conexão rejeitada.

A aceitação de uma conexão solicitada por um cliente é realizada através da função

```
accept(s, struct sockaddr endereço,tamanho)
```

esta função remove a solicitação do cliente da fila de pedidos de conexão e solicita ao sistema operacional que a estrutura de dados definida por `endereço` seja preenchida identificando a outra extremidade da conexão que foi criada e um novo socket, com as mesmas propriedades de `s` seja criado para atender o cliente.

Na extremidade oposta, do cliente, a sequência necessária à criação de uma conexão consiste na criação do socket, usando `socket()`, seguida da conexão ao servidor através de

```
connect(s, struct sockaddr endereço,tamanho)
```

Onde `s` é o identificador do socket criado pelo cliente, `endereço` especifica a outra extremidade da conexão, ou seja o socket associado ao servidor.

Uma vez estabelecida a conexão entre cliente e servidor ambos os processos podem trocar dados entre si utilizando as funções `read()` e `write()`, `send()` e `recv()` ou `recvfrom()`, `sendto()`.

As diferenças entre sockets conectados ou desconectados tornam-se mais claras tendo em contas os mecanismos de comunicação disponíveis no Protocolo Internet ARPA (IP) mais conhecido como TCP/IP.

4.3 IP - O Protocolo Internet

O protocolo IP é o resultado de um projeto de pesquisa iniciado no final da década de 60 pela *Advanced Research Projects Agency*, ARPA, do governo norte-americano [12,6]. Por volta de 1977-79, a arquitetura desta rede e os seus protocolos já estavam definidos na forma atual. Por volta de 1983, visando encorajar as universidades a adotarem e utilizarem estes padrões, a

ARPA financiou a implementação dos mesmos no sistema operacional Unix e financiou a Universidade de Berkeley a integração dos mesmos em sua versão do Unix, 4BSD.

O protocolo IP caracteriza-se basicamente por fornecer dois tipos de serviços [12,15]:

- Serviço de envio de pacotes de dados não conectados, isto é, o protocolo IP é capaz de enviar um pacote de dados de uma máquina para qualquer outra conectada à rede baseado apenas em informações de endereço contidas dentro do pacote de dados.

- Serviço de transporte de dados confiável e orientado à conexão, isto é, o protocolo IP possui uma abstração que permite um computador criar uma conexão com um serviço localizado em outro computador e realizar o envio e recebimento de grandes volumes de dados como se houvesse uma conexão permanente, via hardware, entre ambos computadores.

Embora existam diversos protocolos de interconexão de redes que forneçam esses serviços, o protocolo IP destaca-se dos demais pelas seguintes características [12] :

- Independência da tecnologia de rede utilizada, o protocolo abstrai totalmente a natureza dos meios físicos (modems seriais, rádios, Ethernet, FDDI, ATM, etc) utilizados para interligar os equipamentos.

- Interconexão universal, em uma rede utilizando o protocolo IP cada computador é identificado por um endereço próprio, e único, que o localiza em toda a rede. Cada pacote carrega no seu interior os endereços de origem e de destino permitindo que cada computador intermediário, pelo qual atravesse durante o percurso, determine o caminho a ser percorrido pelo pacote em ambas as direções.

- Comunicação e confirmação ponto a ponto, as comunicações entre dois computadores são diretas; os computadores intermediários utilizados para o transporte de dados não interferem na troca de dados atuando apenas na transferência dos pacotes de uma rede para outra e permitindo que rotas alternativas entre ambos os pontos sejam utilizadas de forma dinâmica.

- Protocolos padronizados para aplicações, adicionalmente aos mecanismos e protocolos de transporte de dados entre computadores, o protocolo IP define padrões bem definidos para as aplicações mais comuns como correio eletrônico, transferência de arquivos, login remoto, etc.

O mecanismo básico de comunicação do protocolo IP consiste de um pacote de dados conforme ilustra a figura 4.3 [15]

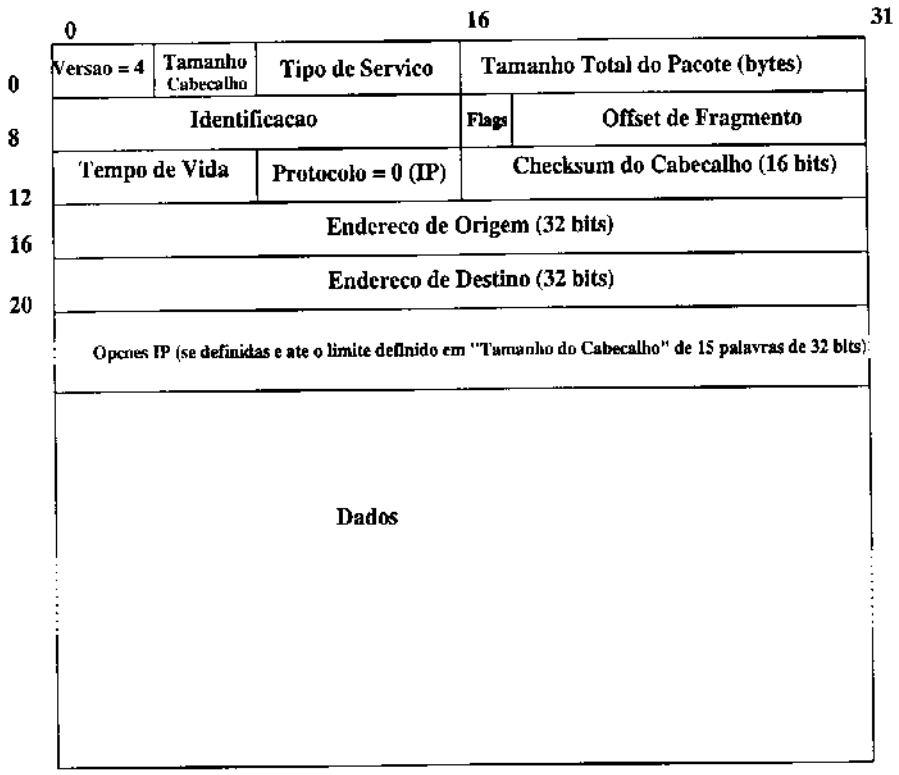


Figura 4.3: Pacote de dados IP ou datagrama IP

Sucintamente um pacote IP consiste de um cabeçalho, onde todas as suas características são definidas e de um bloco de dados. Este cabeçalho possui uma dimensão mínima de 20 bytes e máxima de 60 bytes dependendo do uso de opções IP ou não. Uma vez que o tamanho total do pacote é definido por um numero de 16 bits no cabeçalho o bloco de dados terá um valor máximo de 2^{15} bytes menos o tamanho do cabeçalho. Além disso o cabeçalho define:

- O número de versão para o protocolo IP (atualmente versão 4 e com a versão 6 em etapa final de padronização [12]);
- O tempo de vida do pacote contado em termos de quantos computadores ele já atravessou, isto é, quantas redes ele é permitido percorrer antes de ser abandonado;
- Um protocolo de comunicação com relação ao bloco de dados.

- Adicionalmente campos como *Flags*, *Checksum*, *Identificação*, *Fragmento* definem parâmetros que são utilizados pelo sistema operacional para tomar ações e decisões quanto a integridade e unicidade do pacote e dos dados e a sua manipulação.

Do ponto de vista da química computacional os campos do cabeçalho mais importantes são aqueles que definem o *Protocolo* contido no pacote de dados e os endereços de origem e destino do pacote. Estes endereços, utilizados pelo sistema operacional para definir o roteamento do pacote de sua origem até seu destino final, consistem de números de 32 bits únicos em uma dada rede IP. Quando o sistema operacional precisa enviar um pacote IP ele utiliza unicamente o endereço de destino do pacote para definir suas ações. A figura 4.4 ilustra esta ação em uma rede local simples para dois casos típicos. No primeiro caso o computador (A), com endereço 143.106.13.10, deseja enviar um datagrama IP para o computador (B), com endereço 143.106.13.12. Como ambos encontram-se diretamente conectados à mesma rede física o envio é feito diretamente de (A) para (B). No segundo caso (A) deseja enviar um pacote para o computador (D), localizado em outra rede física e com o endereço 143.106.51.5. Como não há conexão física direta entre ambos, o computador (A) localiza em sua tabela de rotas o computador (C), com endereços 143.106.13.1 e 143.106.51.37 conectado a ambas redes físicas e envia para ele o datagrama. O computador (C) baseado no endereço de destino do pacote recebido realiza a etapa final entregando o pacote ao computador (D). Neste caso o computador (C) agiu como *gateway* ou roteador entre duas redes distintas.

Este mecanismo de roteamento pode ser extendido de uma rede local para uma rede internacional baseada no protocolo IP de forma totalmente transparente conforme ilustrado na figura 4.3 tornando possível a conexão de vários computadores localizados em locais geograficamente distintos e distantes de forma transparente para um programa.

Na figura 4.3 a comunicação entre um computador qualquer conectado à Internet com endereço A1.B1.C1.D1 e outro localizado em qualquer outro local e com endereço A2.B2.C2.D2 é realizada através de sucessivos *gateways*, representados pela nuvem, os quais, baseados unicamente no endereço de destino do datagrama, tomarão decisões de roteamento passando o pacote de dados para o roteador seguinte e, sucessivamente, o datagrama encaminhar-se-á ao seu destino final. O fato do datagrama possuir os endereços de origem e destino do pacote permite que o caminho reverso, não necessariamente o mesmo, seja percorrido em caso de resposta do destino final ou qualquer evento no percurso que necessite o envio de uma mensagem ao computador que gerou o datagrama originalmente.

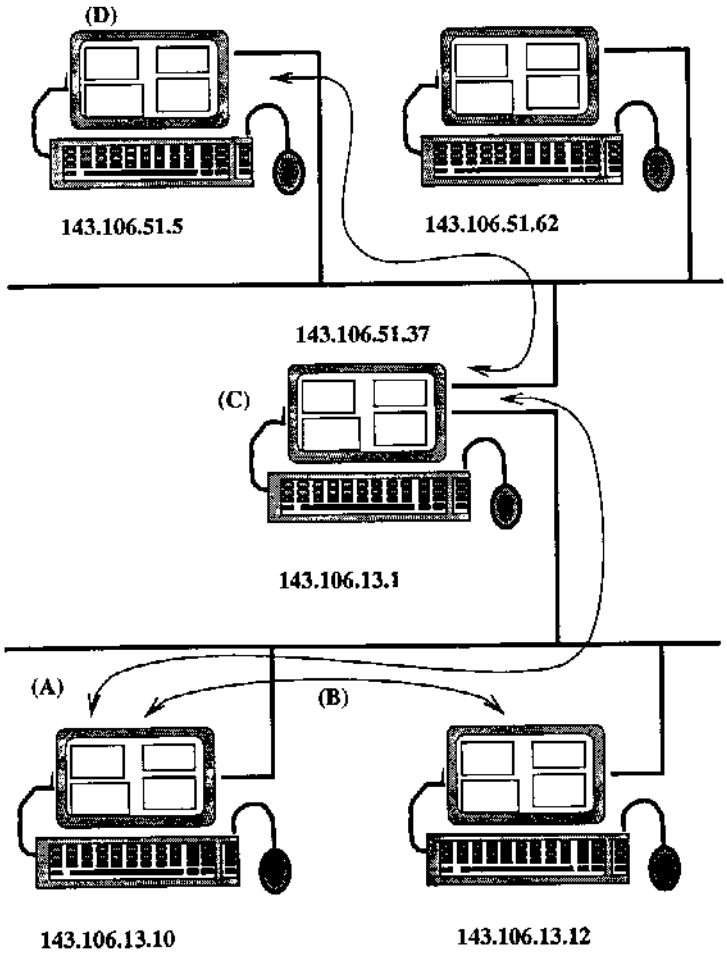


Figura 4.4: Mecanismos de roteamento de pacotes em uma rede local IP

Do ponto de vista de intercomunicação entre dois computadores o protocolo IP fornece os mecanismos de envio e entrega de datagramas mas não provê nenhum suporte que permita a utilização direta dos dados em um ou mais processos computacionais, ou seja, o pacote não carrega consigo nenhuma informação que identifique que processo o originou e a que processo se destina. Da mesma forma o protocolo IP apenas garante o roteamento do pacote entre dois pontos mas não provê nenhum mecanismo de confirmação de entrega dos datagramas ou de verificação de integridade (além do mecanismo `Checksum`) ou sequenciamento correto,ou seja, dois datagrams podem atingir o destino final em ordem inversa daquela em que foram enviados dependendo da rota que cada um utilizou. Para isto o protocolo IP faz uso de dois outros protocolos de comunicação. Utilizando o campo `Protocolo` do

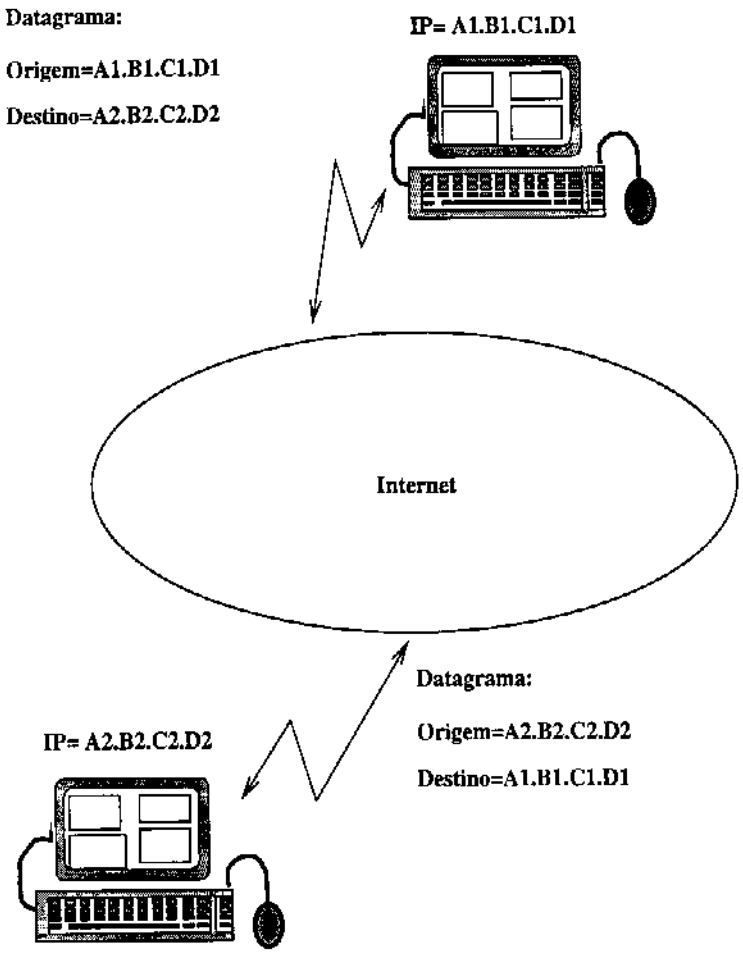


Figura 4.5: Mecanismos de roteamento de pacotes em uma rede mundial IP

cabeçalho para identifica-los e encapsulando-os ou envelopando-os no campo de dados. Estes protocolos são o UDP (*User Datagram Protocol*) e o TCP (*Transmission Control Protocol*) descritos a seguir.

4.3.1 Protocolo UDP

O protocolo UDP utiliza um mecanismo similar ao IP puro no sentido que ele faz o envio dos datagramas não provendo, em si, nenhum mecanismo de garantia que o pacote chegou ao seu destino final. A integridade dos dados é definida apenas por um mecanismo de `Checksum` como no IP e fica ao encargo dos processos que o utilizam verificação deste assim como a ordenação sequenciada dos dados. O que o UDP adiciona ao protocolo IP é o conceito

de portas de comunicação, números de 16 bits, que permitem identificar em ambos os computadores, já identificados pelos respectivos endereços IP, os processos envolvidos na troca de dados. A figura 4.6 ilustra o formato de um datagrama UDP encapsulado em um pacote IP.

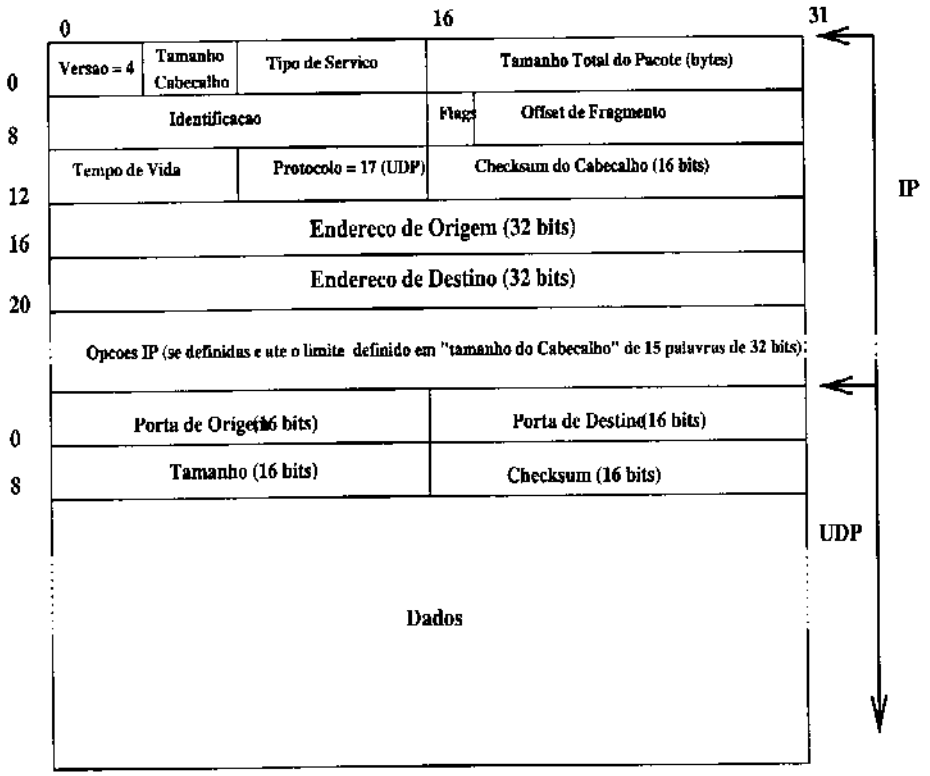


Figura 4.6: Formato de um datagrama UDP encapsulado em um pacote IP.

A utilização de portas numeradas permite que os sistemas operacionais, ao receberem um datagrama, identifiquem na sua tabela de processos qual deles é o destinatário do mesmo. Em outras palavras os pares *endereço1.porta1* e *endereço2.porta2* constituem ponto inicial e final que identificam, de forma unívoca, o canal de comunicação conforme ilustrado na figura 4.3.1.

Este mecanismo de identificação {*computador:processo*} adicional do protocolo UDP aumenta apenas em 4 bytes o espaço gasto para a especificação de um datagrama, adiciona uma sobrecarga adicional ao processador associada a mais um cálculo de Checksum mas elimina completamente a necessidade, por parte do programador, de envolver-se com detalhes internos do sistema operacional necessários à identificação dos datagramas deixando a cargo deste o roteamento completo:

{*computador1:processo1,computador2:processo2*}

Cabe aqui chamar a atenção que o UDP, da mesma forma que o protocolo IP, é de natureza não conectada (*connectionless*), isto é, o envio de um datagram UDP não implica em confirmação ou qualquer outra ação por parte do computador que o recebe e, portanto, a troca de pacotes não estabelece uma ligação permanente entre ambos computadores como se estivessem diretamente conectados fisicamente. Isto implica que, na sua forma nativa, o protocolo UDP não possui meios de obter, dinamicamente, informações sobre o tráfego na rede e as condições de recepção remota. Tanto o protocolo IP quanto o protocolo UDP possuem analogia com os sistemas de rádio e TV neste aspecto, o transmissor não tem como determinar as condições com que os dados chegam ao receptor. Este tipo de canal de comunicação corresponde à semântica SOCK_DGRAM definida na tabela 4.3. Apesar desta característica o UDP tem servido de base para o desenvolvimento de programas, protocolos e serviços amplamente utilizados na Internet. Entre eles pode-se destacar os serviços de DNS (*Domain Name Service*) de natureza global e o protocolo de compartilhamento de discos NFS desenvolvido pela Sun. As principais características positivas do protocolo UDP são a sua simplicidade e eficiência (desempenho) para efetuar o envio de dados entre dois ou mais computadores.

4.3.2 Protocolo TCP

O suporte à criação de circuitos virtuais e o estabelecimento de comunicações totalmente confiáveis entre dois computadores é fornecido através do uso do protocolo TCP encapsulado em um datagrama IP. Conforme ilustra a figura 4.8

O protocolo TCP adiciona ao processo de comunicação além dos pares {`endereço:porta`} dois novos contadores [15,12,19,17]

- Um número sequencial que identifica a ordem em que este pacote foi gerado em relação a todos os demais pacotes enviados pelo `processo-local` que está utilizando a `porta-local` para falar com o `processo-remoto` utilizando a `porta-remota`;
- Um número de confirmação de recebimento que indica ao computador remoto qual o número sequencial do último pacote recebido integralmente.

Além disto o protocolo TCP força a utilização do `Checksum` para verificação da integridade dos dados, provê mecanismos para priorização de um

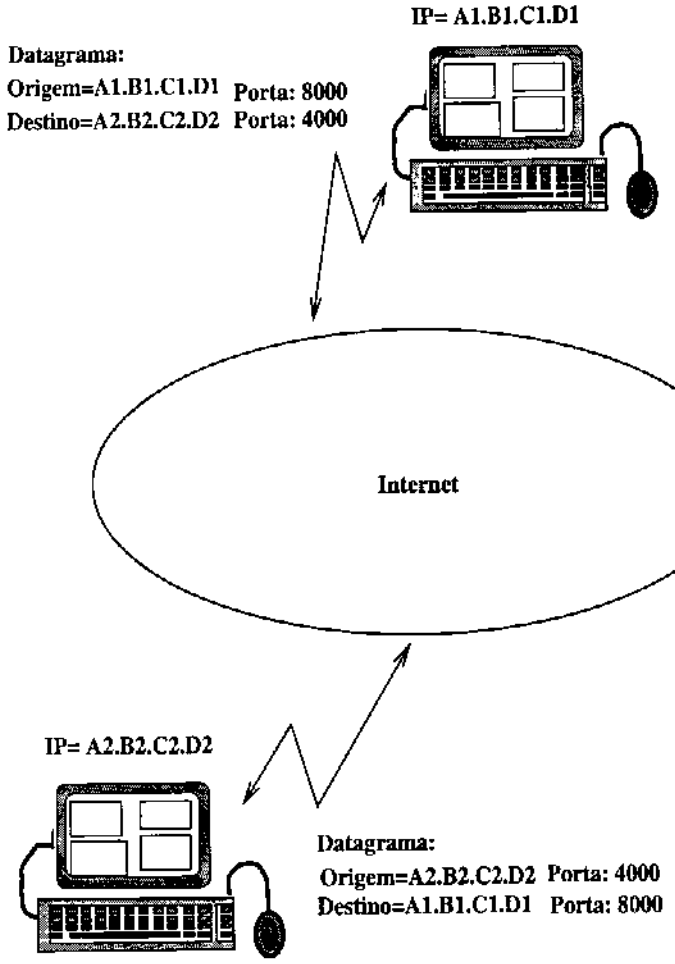


Figura 4.7: Roteamento de um pacote UDP entre dois computadores.

pacote em relação aos demais (`Flags` e apontador de `urgência`) que fornecem ao computador remoto indicações, via `Tamanho da Janela de Dados`, das suas condições de recepção de dados. Desta forma o roteamento de um pacote TCP é realizado dentro dos parâmetros descritos na figura 4.9

As principais vantagens do protocolo TCP derivam das características descritas acima. Esta semântica de comunicação corresponde à semântica SOCK_STREAM definida na tabela 4.3 e possui analogia com uma ligação telefônica em contraste a analogia feita com os protocolos IP e UDP.Utilizando um mecanismo que cria um canal direto entre dois pontos da Internet, como se os mesmos estivessem fisicamente interligados, e provendo mecanismos de sequenciamento correto dos dados e de verificação de sua integridade assim como provendo adaptação dinâmica às necessidades do sistema operacional

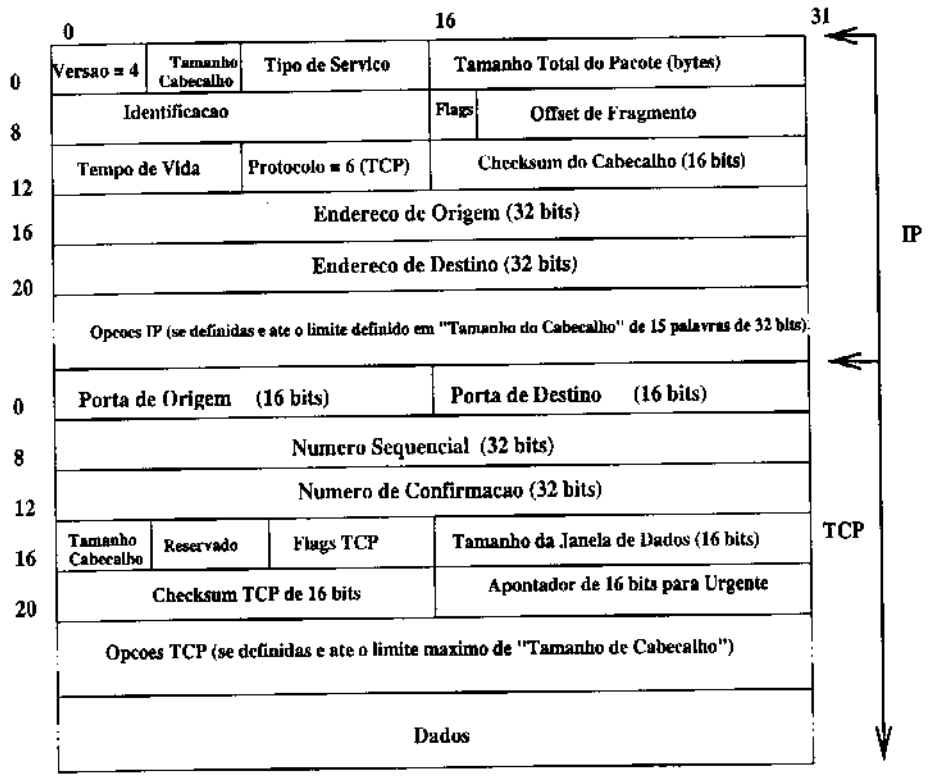


Figura 4.8: Formato de um pacote TCP encapsulado em um pacote IP.

e do processo que o utiliza além das condições de tráfego da rede, o TCP encarrega-se das principais variáveis de comunicação deixando o programador livre para concentrar-se no desenvolvimento de seu programa. Esta versatilidade resultou na utilização do TCP para o desenvolvimento de inúmeros programas de uso rotineiro como Telnet, FTP, rlogin, etc além sua utilização em aplicações como o ambiente gráfico X11 ou versões recentes do protocolo NFS.

Estas características, no entanto, refletem-se de forma negativa no desempenho do canal de comunicação estabelecido. A adição do cabeçalho TCP ao cabeçalho IP incrementa em 20 bytes, no mínimo, a fração de um pacote destinada ao roteamento e controle, duplicando a quantidade de dados necessária para o envio do mesmo. Considerando uma sessão `telnet`, por exemplo, isto significa que cada caracter de 1 byte digitado implica na transmissão de 41 bytes seguida de uma resposta do computador remoto, de 40 bytes no mínimo, confirmando o recebimento do caracter. Da mesma forma os mecanismos de confirmação da integridade dos dados e seu sequenciamento correto implicam em uma sobrecarga computacional para o

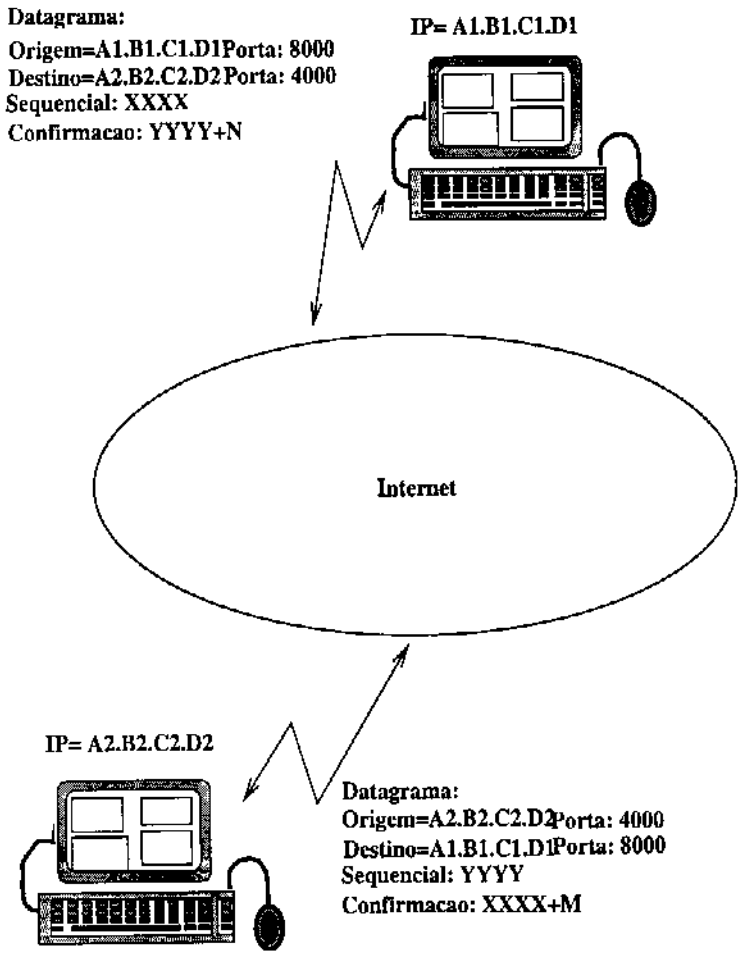


Figura 4.9: Roteamento de um pacote TCP entre dois computadores.

sistema operacional em ambas as extremidades do canal de comunicação. Visando otimizar estes aspectos de desempenho do TCP foi desenvolvido o padrão T/TCP, ou TCP para transações (*TCP for Transactions*) [20] em 1993 e tornado público em 1994. Infelizmente poucos sistemas operacionais atuais (basicamente aqueles baseados no 4.4BSD e SunOS4) suportam estas extensões.

Da óptica da química computacional o protocolo TCP mostra-se o mais apropriado dentre os protocolos Internet uma vez que ele encarrega-se da maior parte dos procedimentos necessários para o estabelecimento de uma conexão e posterior intercâmbio de dados. Entre as aplicações da química computacional que tornam-se viáveis através do TCP encontram-se a visualização científica usando um modelo cliente-servidor, a criação de máquinas virtuais através da interligação de um grande número de computadores e aplicações em geral que necessitem do suporte confiável fornecido por este protocolo.

4.3.3 Outros protocolos Internet

Além dos protocolos UDP e TCP, o protocolo IP possui uma série de outros protocolos voltados para tarefas específicas. A tabela 4.4 sumariza os principais protocolos utilizados atualmente [21,7].

Número	Sigla	Finalidade
0	IP	Internet Protocol
1	ICMP	Internet Control Message Protocol
2	IGMP	Internet Group Management
3	GGP	Gateway-Gateway Protocol
4	IP-ENCAP	IP Encapsulated in IP
6	TCP	Transmission Control Protocol
8	EGP	Exterior Gateway Protocol
12	PUP	PARC Universal Packet Protocol
17	UDP	User Datagram Protocol
20	HMP	Most Monitoring Protocol
46	RSVP	Resource ReSerVation Protocol
51	IPSEC	IP Secure (encrypted)
81	VMTP	Versatile Message Transport
89	OSPFIGP	Open Shortest Path First IGP

Tabela 4.4: Protocolos definidos para encapsulamento IP

Via de regra esses protocolos voltam-se para os aspectos administrativos e de gerenciamento de uma Internet e não encontram-se acessíveis ao pesquisador, pois o sistema operacional exige privilégios de acesso, ou não possuem aplicação direta em computação e programação científica, com excessão talvez de IP-ENCAP e RSVP que permitem a criação de canais de áudio e vídeo com qualidade definida. Em virtude disto não serão abordados neste trabalho.

4.4 Computação Distribuída Utilizando Sockets

Do que foi exposto torna-se evidente a estreita relação entre o modelo de programação fornecido pelos sockets BSD e a arquitetura e mecanismos de comunicação providos pelo protocolo IP e sua integração com o sistema operacional Unix.

Desta forma os *sockets* apresentam-se como a escolha mais adequada para o desenvolvimento de programas distribuídos em rede ou para execução local do ponto de vista de compatibilidade e transportabilidade entre diversos sistemas Unix.

No entanto, o TCP/IP, nome genérico utilizado para designar o protocolo Internet, e os protocolos nele embutidos, apesar de robusto e eficiente, ainda deixa sob a responsabilidade dos processos que o utilizam, a solução de alguns problemas resultantes das características dos processadores e sistemas operacionais. Estes problemas dizem respeito às diferentes arquiteturas de hardware existentes, aos diferentes padrões utilizados para aritmética de ponto flutuante e aos mecanismos de identificação de usuário e processo usados pelos sistemas operacionais.

Atualmente os processadores podem ser classificados em duas categorias de acordo com a maneira com que armazenam e representam palavras de 32 bits: *big endians* e *little endians*. Processadores *big endian* armazenam os 4 bytes que compõe uma palavra na ordem 1234 enquanto que processadores *little endian* a armazenam na ordem 4321. Esta diferença é suficiente para causar problemas, inviabilizando a comunicação entre processos que estejam executando em maquinas com ordenação diversa de bytes. Exemplos de processadores *little endian* podem ser encontrados na linha Intel enquanto que processadores Sparc são *big endian*. Embora a biblioteca padrão do Unix forneça funções para ajustes relacionados a ordem dos bytes (ex. `ntoh()`, `hton()`) elas não se adequam, por exemplo, ao processamento de variáveis de ponto flutuante ou caracteres alfabéticos.

Outro problema está associado à convenção utilizada pelo sistema operacional para representar variáveis de ponto flutuante de precisão simples ou dupla. Apesar de existirem padrões para isto, como o IEEE 754, nem todas bibliotecas matemáticas dos sistemas operacionais os adotam [2]. Por último, é possível que, em um mesmo cluster de computadores, existam processadores com tamanho de palavra diferentes, por exemplo, processadores Alpha ou UltraSparc de 64bits interoperando com processadores Intel de 32bits.

Visando contornar os problemas associados às diversas formas de representação interna, algumas soluções foram propostas. Neste trabalho foram investigadas as seguintes, a codificação dos dados no Formato Hieráquico de Dados (HDF Hierarchical Data Format) do NCSA [22], adequado à criação, armazenamento e troca de arquivos e o padrão Representação Externa de Dados (XDR eXternal Data Representation) desenvolvido pela Sun Microsystems [23,24]

4.4.1 Codificação HDF

A codificação HDF foi desenvolvida para permitir que equipamentos totalmente diversos quanto a representação de dados possam intercambiar arquivos entre si. Esta codificação permite não apenas o intercâmbio de dados numéricos de qualquer precisão ou representação, como também de imagens, textos, etc. Embora a codificação HDF contemple praticamente todas as variações possíveis no que diz respeito a ordenação de bytes, tamanho de palavra e precisão numérica ela não é viável para utilização em tempo real pois utiliza mecanismos de cabeçalhos hierárquicos de identificação de blocos de dados. Esta característica torna-a bastante adequada para a troca de arquivos completos de dados entre sistemas diferentes como visto no capítulo 2, mas é inapropriada para comunicação em rede.

4.4.2 Codificação XDR

A solução para o problema de troca de dados em tempo real foi desenvolvida pela Sun MicroSystems na forma de uma especificação e posta em domínio público [23,24] com o nome de XDR, tendo sido adotada em todas as versões do UNIX e demais sistemas operacionais. Esta especifica a forma de codificar os dados a serem transmitidos de uma forma bem definida para os todos computadores utilizando o formato XDR. Este formato, por exemplo, define os diversos formatos conforme a figura 4.10.

Fica a cargo do programa detectar se o computador remoto difere com relação a representação de dados e em função disto codificar os dados da sua representação interna para o padrão XDR de representação externa antes de

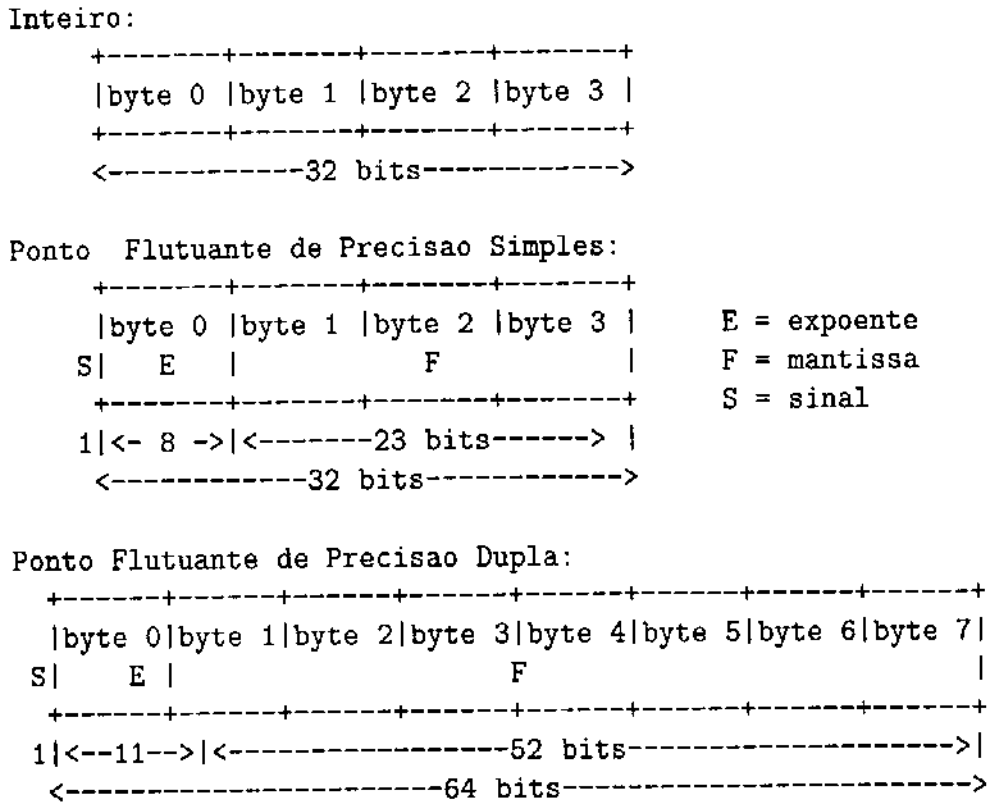


Figura 4.10: Exemplos de representação externa XDR

envia-los. Fica a encargo do computador remoto identificar esta situação e realizar a conversão dos dados recebidos para a sua representação interna.

Evidentemente a conversão entre a representação interna e a representação externa, quando ocorre, consiste em mais um encapsulamento de dados, como no caso dos protocolos UDP e TCP e insere um atraso adicional no tempo de transmissão e recepção dos dados entre máquinas com diferentes representações internas. O padrão XDR é amplamente adotado e empregado em várias aplicações de rede como por exemplo no sistema de compartilhamento de discos e arquivos via NFS. Esta ampla adoção assim como a sua disponibilidade pública faz com que o XDR seja encontrado em todos os ambientes Unix.

4.4.3 Identificação, Autenticação e Segurança

Um último obstáculo na programação via sockets diz respeito a maneira como um sistema operacional organiza-se internamente para identificação de seus processos. Sistemas monousuário como Windows95, MacOS, MS/DOS não possuem o conceito de credenciais de usuário e direitos de acesso correspondentes. Sistemas multiusuários como OpenVMS, WindowsNT e Unix, por sua vez, atribuem uma crendecial de usuário a cada processo em execução e utilizam esta credencial para determinar os privilégios de acesso aos recursos do sistema operacional correspondentes a essas credenciais.

Conforme descrito em 4.2.6 um datagrama IP transporta apenas as credenciais relativas a identificação de ambas as extremidades de uma conexão não identificando, de forma alguma, o usuário associado ao datagrama.

A criação de um socket, vista em 4.2.5, seja ele como um cliente ou como um servidor é totalmente independente da identidade do usuário dono do processo que a realiza. Em outras palavras, em princípio qualquer usuário válido pode criar um ou mais sockets ficando a cargo do sistema operacional a autorização ou não baseado em critérios de sua arquitetura.

Esta flexibilidade do protocolo Internet da interface de programação sockets BSD permite que sistemas monousuários, ao quais falta o conceito de credenciais de usuário, por exemplo, conectem-se com sistemas multiusuários ou que estes últimos, mesmo diferindo com relação aos critérios identificação de processos e usuários, interoperem de forma transparente. Infelizmente esta flexibilidade acarreta na fragilização da segurança de dados e do sistema como um todo permitindo que, a menos que se adotem mecanismos de identificação e autenticação, um usuário voluntariamente personifique outro com todas as consequências associadas a isto. Isto torna necessário a adição de mais uma etapa no processo de estabelecimento de uma conexão entre dois computadores que identifique de forma única o usuário remoto para que am-

bos os sistemas operacionais possam exercer o controle de acesso necessário sobre os processos.

4.4.4 Bibliotecas de Comunicação

É evidente que, embora os sockets e o protocolo IP forneçam todos os mecanismos básicos de comunicação entre processos, uma série de problemas devem ser contornados no que diz respeito a interoperabilidade entre arquiteturas e sistemas operacionais diversos. A utilização das diversas soluções para estes problemas no entanto pode conduzir a programas complexos onde grande parte da codificação está voltada para a comunicação em si ocultando os algoritmos computacionais empregados para a solução do problema. Ao mesmo tempo a diversidade de soluções individuais pode criar códigos extremamente difíceis de transportar para outros programas.

Visando simplificar e uniformizar a programação de programas distribuídos uma série de sistemas e bibliotecas de comunicação foram desenvolvidos. A seguir são descritos as principais linguagens e bibliotecas de interesse à química computacional.

As linguagens e bibliotecas voltadas para comunicação entre processos buscam simplificar e reduzir o tempo de desenvolvimento de programas além de propor uma padronização para os programadores. Na área de linguagens interpretadas a linguagem Perl desenvolvida por Larry Wall [25] possui todos os recursos necessários para o desenvolvimento de aplicações distribuídas simples de forma rápida, portátil e eficiente.

Como toda linguagem interpretada Perl peca pelo elevado tempo de inicialização, o baixo desempenho, se comparada a linguagens compiladas e as dificuldades de programação para projetos de grandes dimensões, e prima pela facilidade e velocidade de desenvolvimento e teste de protótipos de programas mais complexos a serem escritos em C e por este motivo foi utilizada neste trabalho.

No campo das linguagens compiladas o fortran-m, desenvolvido por Foster e Chandy [26] ou o Pfortran, desenvolvido por Clark e colaboradores [27], são exemplos de extensões introduzidas na linguagem Fortran77 para incorporar rotinas de comunicação. Outras soluções na mesma direção levam aos compiladores proprietários de fabricantes de computadores como IBM ou Silicon Graphics que restringem a sua utilização a um dado modelo de equipamento.

Do ponto de vista do desenvolvimento de novos padrões de linguagem, tanto o Fortran90 quanto o HPF (High Performance Fortran) tem sido propostos por consórcios de fabricantes e universidades [2] num esforço de propriciar aos programadores ambientes de desenvolvimento totalmente independentes da natureza do equipamento ou arquitetura onde os programas serão

executados. Estes esforços, no entanto, não tem mostrado o sucesso esperado em virtude da evolução rápida do mercado tornando obsoletos os padrões ou demasiado específicos a uma fatia de mercado. Um exemplo disto é a obsolescência do Fortran90, forçando a definição do Fortran95, em virtude de IBM e DEC terem implementado extensões àquela linguagem incompatíveis entre si.

Do ponto de vista de programação genérica de sistemas utilizando linguagem C, a Sun desenvolveu, e colocou em domínio público, em 1988 [28] uma biblioteca que incorpora em si os componentes básicos de programação utilizando sockets, codificação XDR e a capacidade de execução remota de subrotinas. Esta biblioteca denominada *Sun RPC (Remote Procedure Call) Library*. Este padrão sofreu uma revisão em 1995 [29] e tem sido amplamente utilizado no desenvolvimento de aplicações cliente/servidor de sistemas (*NFS, rstatd, rusersd*) e serviu como base e modelo para o desenvolvimento de outros sistemas comerciais baseados em *remote procedure call* como o DCE/OSF cuja disponibilidade é restrita a alguns vendedores. Embora a *Sun RPC* ofereça recursos genéricos de programação em rede ela mostra-se mais apropriada ao desenvolvimento de programas voltados para o atendimento de transações cliente/servidor.

Simultaneamente ao desenvolvimento de linguagens de programação, projetos voltados ao desenvolvimento de bibliotecas científicas de programação que forneçam os elementos básicos de comunicação entre processos tem se mostrado mais bem sucedidos entre a comunidade científica. Estas bibliotecas, denominadas MPL (*Message Passing Libraries*), provém as primitivas básicas de comunicação necessárias à programação distribuída, contornando todos os problemas descritos acima, mantendo-se, no entanto, suficientemente portáteis e adaptáveis aos mais diversos problemas computacionais. Esta granularidade permite, dentro de um certo grau, o desenvolvimento de algoritmos e programas para os diversos modelos e arquiteturas disponíveis para computação distribuída.

Entre as bibliotecas de comunicação disponíveis para o desenvolvimento de programas adequados para execução em clusters de estações de trabalho, três mostram-se especialmente interessantes para a área de química.

- A biblioteca comercial Linda desenvolvida por Gelerntner [30], utilizada pelo pacote Gaussian94 e disponível apenas para alguns sistemas operacionais.

- A biblioteca TCGMSG (*Theoretical Chemistry Group MesSaGe passing library*) desenvolvida por Robert J. Harrison no grupo de química teórica do Argonne National Laboratory [33]

- A biblioteca PVM (*Parallel Virtual Machine*) desenvolvida pelos grupos de pesquisa em ciência da computação do Oak Ridge National Laboratory, Emory University e University of Tennessee [34,35,36].

As bibliotecas TCGMSG e PVM por serem públicas foram analisadas neste trabalho. Ambas podem ser chamadas utilizando as linguagens C e Fortran, utilizam o protocolo XDR, realizam a autenticação de usuários utilizando recursos e programas do sistema operacional e possuem versões operacionais para uma ampla gama de equipamentos desde microcomputadores PC até supercomputadores Cray, podendo, também, ser utilizadas em sistemas multiprocessados.

Uma vantagem adicional das bibliotecas PVM e TCGMSG é que ambas são distribuídas eletronicamente na forma de programa fonte. Isto permite a análise e modificação das mesmas e adaptação às condições e sistemas utilizados (como foi feito neste trabalho para os sistemas baseados no 4.4BSD) o que é impossível com bibliotecas comerciais.

Embora semelhantes em sua funcionalidade estas duas bibliotecas diferenciam-se nos seus paradigmas de programação. A biblioteca TCGMSG é orientada para um modelo onde um mesmo programa é executando simétrica e simultaneamente em todas as máquinas do cluster sendo que cada uma opera sobre uma parte do conjunto de dados de acordo com algum critério definido pelo programador (*SPMD Single Program Multiple Data*). Esta foi, por exemplo, a solução adotada na paralelização do programa de cálculos abinitio GAMESS por Windus [37]. Já a biblioteca PVM é orientada para a decomposição de um programa em um conjunto de outros programas que podem ou não ser executados simultaneamente realizando ou não o mesmo tipo de tarefa (*MPMD Multiple Program Multiple Data*), não descartando a execução SPMD se esta mostrar-se mais adequada.

Visando reduzir as diferenças entre as diferentes bibliotecas MPL, recentemente foi proposto o padrão MPI (Message Passing Interface) coordenado pelo MPI Forum [38] que visa tornar transparente ao programador qual MPL será utilizada em um dado equipamento ou cluster. Basicamente o MPI adiciona mais uma API uniforme sob a qual podem ser empregadas diferentes MPLs. E como toda interface de programação que envelopa outras interfaces está sujeito ao impacto de desempenho e as limitações de todas elas simultaneamente.

Ao contrário da biblioteca TCGMSG, a biblioteca PVM tem obtido penetração expressiva na comunidade científica, tendo sido utilizada como base para o desenvolvimento de sistemas computacionais distribuidos como a biblioteca de matrizes distribuídas DALIB [39] e o tradutor Xadaptor do GDE [40] para a conversão de programas escritos em HPF para Fortran77. A sua

Num. CPU	1	2	4	8	16	32
RISC6000	8847	4511	2345			
Y-MP	6482					
Delta	33309	16941	8652	4647	2497	1424

Tabela 4.5: Tempos de execução para o cálculo SCF

versatilidade também permitiu a adaptação do PFortran [27] desenvolvido para arquiteturas hipercubo Intel, neste trabalho, para geração de código baseado em sockets BSD. Outros trabalhos utilizando PVM incluem a biblioteca PIM para métodos iterativos desenvolvida por Cunha e Hopkins [41].

A título de ilustração da aceleração possível em cálculos moleculares a tabela 4.5 mostra os resultados obtidos para a molécula $(CH_3)_3AlPH_2$ com simetria C_s utilizando 169 funções de base. Os cálculos foram feitos em um supercomputador (Cray Y-MP), em um computador multiprocessado (Delta) e em um cluster de estações IBM RISC6000 modelo 560 [37].

A análise da tabela 4.5 mostra a viabilidade dos clusters de estações de trabalho em relação a equipamentos mais dispendiosos e sofisticados.

4.4.5 Biblioteca PVM

A biblioteca PVM foi desenvolvida visando a construção de *máquinas virtuais* na Internet através da interconexão de um grande número de computadores. Esta interconexão é realizada através de um processo iniciado em cada computador que compõe a máquina virtual denominado *pvmd* ou *daemon pvm*. Inicialmente um processo *pvmd* é posto em execução manualmente pelo usuário. Este processo, em seguida, encarrega-se de criar os demais *pvmd* nas máquinas restantes que comporão a máquina virtual utilizando os comandos *rsh* ou *rexec* do Unix para realizar a autenticação e identificação de usuário. Estes processos *pvmd* comunicam-se entre si utilizando o protocolo UDP e são responsáveis pela execução e término de outros programas na máquina virtual.

A execução de processos distribuídos no PVM é feita através de processos denominados *Tasks*, ou tarefas, que serão criados a partir de um *Task* mestre, inicialmente, podendo também ser criados por outros *Tasks*. Estes processos utilizam a biblioteca *libpvm* para comunicação, quando eles comunicam-se com os *pvmds* utilizam o protocolo TCP ou sockets no domínio Unix *pvmd* local. A comunicação entre os *Tasks* utiliza o protocolo TCP, caso residam em computadores distintos, ou sockets no domínio Unix caso residam no

mesmo computador.

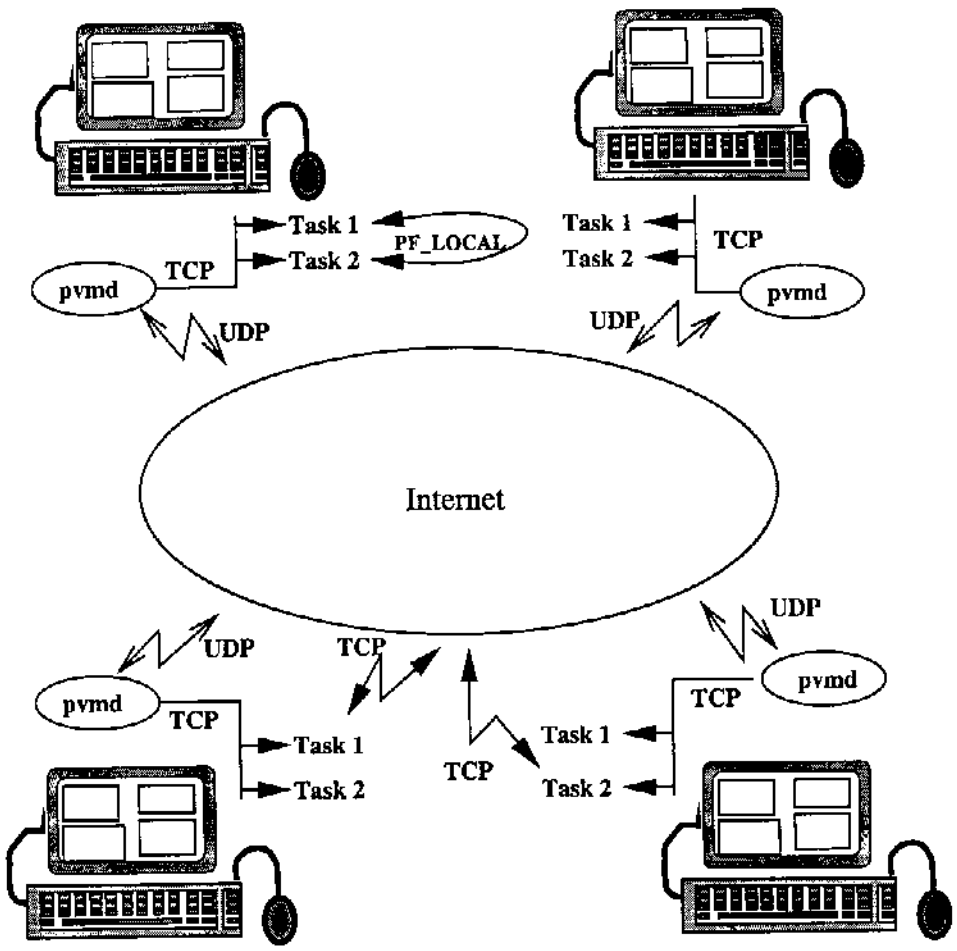


Figura 4.11: Modelo de comunicações do PVM

O roteamento de mensagens entre *Tasks* é feito através de pacotes que mimetizam o roteamento IP. Cada processo é identificado unicamente na máquina virtual por um número de 32 bits denominado **TID** *Task IDentifier*. Os *pvmd* mantém o controle da atribuição do TID e a relação entre TID e computadores da máquina virtual. As mensagens enviadas pelos processos são, analogamente ao protocolo IP, identificadas por um TID de destino e por um TID de origem. Cada mensagem possui um cabeçalho que identifica a sua natureza (dados, controle, etc) e é codificada via XDR pelos pvmd ou pela *libpvm* caso as arquiteturas dos computadores difiram entre si. A figura 4.11 ilustra o modelo de comunicação descrito.

Tecnologia	Velocidade (Mb/s)
Ethernet	10
Fast Ethernet	100
FDDI	100
ATM	155
ATM	655
Giga Ethernet	1000

Tabela 4.6: Principais tecnologias de rede disponíveis

4.5 Desempenho - Algumas Considerações

Até este ponto do trabalho foram definidos uma série de fatores necessários a utilização de uma rede de computadores interconectados em rede para que se possa desenvolver e utilizar programas que operem de forma cooperativa para:

- Reduzir o tempo de cálculo de um dado problema de química computacional, e/ou;
- Abordar problemas de química computacional cujas dimensões não poderiam ser manipuladas em um único computador, e/ou;
- Agrupar de forma aditiva computadores com funcionalidades específicas de forma a obter uma nova funcionalidade.

Embora a computação distribuída contemple todos os objetivos citados, nem sempre ela poderá ser a forma mais eficiente de alcança-los quando a análise for feita única e exclusivamente sob a óptica do desempenho puro. As tecnologias de rede disponíveis atualmente para uso em computação distribuída basicamente são:

4.5.1 Ethernet 10Mb/s - Desempenho Teórico

A tecnologia Ethernet possui um limite de 1538 bytes no tamanho máximo que um pacote de dados deve ter para que seja transmitido. Deste limite, denominado MTU (Maximum Transmission Unit) do meio físico, apenas 1460 dados podem ser utilizados por um processo e os restantes 78 bytes são utilizados pelo meio físico (placa de comunicação) para controle e endereçamento na rede local. Desta forma apenas 95% da banda de 10 Mbits/s estão disponíveis para o usuário, supondo que o meio físico esteja dedicado

Tecnologia	Vel. Teórica Mb/s	Vel. Observada Mb/s	%
Ethernet	9,5	8,8	92,6
Fast Ethernet	95	90	94,7
FDDI	100	97	87,6

Tabela 4.7: Desempenho experimental de redes Ethernet e FDDI

exclusivamente a comunicação entre dois computadores isto resulta em 9.5 Mbits/s. Considerando que 1 byte = 8 bits a velocidade teórica máxima esperada é de 1187500 bytes/s, ou 1159.6 kbytes/s. Em termos de dados numéricos utilizados em química computacional onde um número inteiro e um número real são representados por 32 bits, ou 4 bytes, e um número de precisão dupla por 64 bits ou 8 bytes isto representa, respectivamente, a capacidade de transferir 290 kbytes ou 145 kbytes de dados em cada transação.

4.5.2 Desempenhos Observados

Os resultados teóricos previstos para Ethernet 10 não levam em conta fatores associados ao processador, placa de comunicação utilizada, congestionamento do meio físico, etc. Neste trabalho uma série de medidas experimentais foi realizada para as redes de tecnologia Ethernet, FastEthernet e FDDI utilizando os programas `tcpblast`, `bing` e `netperf` considerados padrões para medidas desta natureza. Os resultados obtidos encontram-se resumidos na tabela 4.7.

A tabela 4.7 reflete apenas os resultados obtidos para as redes e os computadores em que foram realizadas as medidas e os resultados dela concordam aqueles relatados por Stevens[12] que mostram um desempenho experimental médio em torno de 80 a 90% do valor nominal da tecnologia. Considerando os resultados analisados para uma Ethernet 10 Mb/s pode-se esperar para as demais tecnologia uma capacidade de transferência da ordem de 2.8Mbytes/s (Fast Ether e FDDI).

4.5.3 Desempenho - redes vs. outras tecnologias

A análise destes valores torna-se relevante quando comparada com os valores obtidos para medidas de velocidade de acesso a disco e memória física local e confrontados com valores de velocidade de cálculo do processador. Neste trabalho foram realizadas medidas de velocidade de acesso a disco utilizando o programa de *benchmark* `bonie` para discos utilizando as tecnologias SCSI e IDE. Também foram realizadas medidas de velocidade de acesso a memória

empregando o programa de *benchmark* `stream` e medidas de velocidade de cálculo de ponto flutuante utilizando o programa de *benchmark* `flops`. Os resultados obtidos para vários processadores encontram-se sumarizados na tabela 4.8

Tecnologia	Velocidade MB/s	Processador	Chipset
Ethernet 10	0,28	Pentium133	
Ethernet 100	2,8	Pentium133	
FDDI	2,8	Pentium133	
Disco/Tecnologia			
IDE/ATA (leitura)	6,0	Pentium133	PCI/VX
IDE/ATA (escrita)	6,0	Pentium133	PCI/VX
SCSI2 (leitura)	8,0	Pentium133	PCI/VX
SCSI2 (escrita)	8,0	Pentium133	PCI/VX
Processador	Memória MB/s	Vel. (MHz)	Chipset
Pentium166	70,0	166	PCI/Triton 1
Pentium133	150,0	133	PCI/VX
PentiumPro	150,0	200	PCI/Natoma
Alpha	150,0	266	DEC
UltraSparc2	200,0	200	Sun
Processador	MFLOPS	Vel. (MHz)	Fabricante
Pentium166	30,0	166	Intel
PentiumPro	50,0	200	Intel
UltraSparc2	100,0	200	Sun
Alpha	300,0	266	DEC

Tabela 4.8: Resultados de Benchmark

A análise da tabela 4.8 deve ser realizada dentro do mesmo escopo da análise feita para a tabela 4.7, isto é, ela deve servir como indicativo da velocidades relativas de transferência de dados em computação utilizando uma série de tecnologias. Conforme discutido em Dowd[2], o desempenho real de um computador depende de uma série de fatores associados ao ambiente de execução e à natureza das tarefas realizadas nele que enfatizarão discrepâncias entre os índices absolutos isolados, como é o caso do computador Alpha medido, onde não há correspondência entre o elevado poder de cálculo e a velocidade de acesso à memória.

A tabela 4.8 mostra, por outro lado, que as atuais tecnologias de rede possuem uma velocidade de transferência de dados inferior às demais tecnologias, mesmo considerando uma transição rápida para GigaEthernet nos

Processador	MFLOPS	Vel. Leitura	%	Vel. Escrita	%	Vel. Obs.
Pentium166	30,0	240,0	29,1	120,0	58,3	70,0
PentiumPro	50,0	400,0	37,5	200,0	75,0	150,0
UltraSparc	100,0	800,0	25,0	400,0	50,0	200,0
Alpha	300,0	1200,0	12,5	600,0	25,0	150,0

Tabela 4.9: Desempenho de memória para alguns processadores

próximos anos isto colocaria as taxas de transferência de dados (em torno de 30MB/s) ainda abaixo de tecnologias ultrapassadas de gerenciamento de acesso a memória como a Triton 1 e na ordem de grandeza de tecnologias de acesso a disco (não medidas neste trabalho) mais avançadas que IDE/ATA ou SCSI2 já disponíveis como *disk stripping*.

Do mesmo modo os números da tabela 4.8 mostram atualmente uma disparidade entre os desempenhos obtidos pelos processadores para cálculos de ponto flutuante e a velocidade máxima com que conseguem acessar a memória para realização desses cálculos. Considerando a definição elementar de uma operação de ponto flutuante (FLOP) como sendo

- leitura de dois números de 4 bytes (precisão simples) da memória do computador;
- uma operação aritmética elementar (soma, multiplicação, subtração ou divisão);
- armazenamento do resultado (4 bytes) na memória do computador

isso implica na transferência de 8 bytes de dados (leitura) seguida da transferência de 4 bytes (escrita). A tabela 4.9 resume as velocidades de memória necessárias para a manutenção dos resultados na tabéla 4.8

Estes resultados mostram claramente que nenhum dos computadores medidos possui um sistema de memória com desempenho compatível com os processador utilizado. Da lista acima o processador PentiumPro é aquele que mostra o melhor equilíbrio entre a sua capacidade de cálculo e a sua velocidade de acesso à memória. Nos demais casos as diferenças observadas apenas indicam que a capacidade de cálculo (MFLOPS) medida foi obtida através do sistema de *cache* de nível 2 dos computadores em virtude do programa utilizado `flops` ser pequeno o suficiente para não necessitar acessar a memória principal, ao contrário do programa `stream` projetado para forçar este acesso.

4.5.4 Desempenho - Fatores a serem considerados

Os resultados obtidos apontam uma séric de fatores a serem levados em conta para o desenvolvimento de programas que explorem os recursos oferecidos por uma rede IP. É possível constatar que apesar das taxas de transferência inferiores às demais tecnologias, os desequilíbrios de desempenho entre as varias formas de transferência de dados de e para o processador exigem que os seguintes fatores sejam levados em conta em qualquer algoritmo computacional que vise obter desempenho:

- Os algoritmos computacionais devem levar em conta as grandes diferenças de velocidade do processador e memória principal explorando ao máximo o mecanismo de cache e evitando acesso contínuo à memória principal. Isto aplica-se tanto a algoritmos locais quanto a troca de dados via rede;
- Blocos de dados médios, que podem residir em memória sem prejuízo do item anterior devem ser intercambiados entre os membros participantes de um cluster; e
- Blocos de dados de grandes dimensões devem residir na forma de cópia nos discos locais de cada equipamento do cluster ao longo de uma tarefa computacional

4.6 Aplicações

A partir do que foi exposto uma série de aplicações foi desenvolvida no presente trabalho visando avaliar a viabilidade das técnicas disponíveis para computação distribuída conforme descrito a seguir.

4.6.1 Paralelização do Programa FORCES

Neste trabalho a foi investigada a paralelização das partes mais intensivas computacionalmente do programa FORCES de mecânica molecular desenvolvido neste grupo [42]. Neste programa a energia potencial molecular é expandida em série de Taylor e expressa pela equação 4.1

$$U = U_0 + \sum_{i=}^{N} A_i^{\alpha} \Delta X_i^{\alpha} + \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} C_{i,j}^{\alpha,\beta} \Delta X_i^{\alpha} \Delta X_j^{\beta} \qquad i,j = 1, \cdots N \qquad \alpha, \beta = x, y, z \tag{4.1}$$

Onde A_i^α representa os elementos do vetor gradiente e $C_{i,j}^{\alpha,\beta}$ representa os elementos da matriz hessiana de segundas derivadas. Aplicando a condição de mínimo de energia potencial U a 4.1 temos 4.2:

$$\frac{\partial U}{\partial \Delta X_i^\alpha} = 0 \quad (i = 1, \cdots N, \alpha = x, y, z) \tag{4.2}$$

da qual obtém-se um conjunto de $3N$ equações algébricas lineares para os elementos de X_i^α, dados pela seguinte equação:

$$\begin{aligned} -A_i^\alpha = \sum_{i=1}^{N}\sum_{j=1}^{N} C_{i,j}^{\alpha,\beta}\Delta X_j^\beta \\ i = 1, \cdots N \\ \alpha, \beta = x, y, z \end{aligned} \tag{4.3}$$

Em virtude das aproximações realizadas para a construção da equação 4.1 a equação 4.3 é resolvida iterativamente gerando um novo conjunto de coordenadas X_i^α na forma:

$$X_{i_{novas}}^\alpha = X_{i_{antigas}}^\alpha + \Delta X_i^\alpha \quad (i = 1, \cdots N) \tag{4.4}$$

Os valores de X_i^α obtidos em 4.4 são utilizados para atualizar os coeficientes de 4.3 e o processo érepetido até que os valores de ΔX_i^α em 4.4 sejam considerados suficientemente pequenos e que os termos A_i^α possam ser considerados iguais a zero e as primeiras derivadas nulas. Neste ponto a energia potencial pode ser expressa por 4.5

$$U = U_0 + \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} C_{i,j}^{\alpha,\beta}\Delta X_i^\alpha \Delta X_i^\beta \tag{4.5}$$

A matriz **C** pode então ser utilizada para o cálculo dos números de onda vibracionais após sua multiplicação pelos recíprocos das raizes quadradas das massas atômicas Mi para formar a matriz **CM**:

$$CM_{i,j}^{\alpha,\beta} = \frac{C_{i,j}^{\alpha,\beta}}{\sqrt{M_i M_j}} \quad (i, j = 1, \cdots N,\ \alpha, \beta = x, y, z) \tag{4.6}$$

Os autovalores de **CM** correspondem aos números de onda vibracionais e os autovetores aos modos normais vibracionais em coordenadas cartesianas de deslocamento.

As etapas com maior demanda computacional foram determinadas com base nos critérios empregados anteriormente por Clark e colaboradores [43,45] e através da utilização do programa *gprof* [2,44] para análise do perfil de execução.

- A formação da matriz hessiana de segundas derivadas onde são calculados um número muito grande de termos envolvendo 2 ou mais átomos
- A solução do sistema de equações lineares resultante para obtenção do vetor de deslocamentos atômicos.
- A diagonalização da matriz hessiana ponderada em massas atômicas para obtenção das frequências vibracionais.

Embora para moléculas de tamanho médio (50 átomos) o tempo de cálculo em cada uma das etapas seja pequeno, para sistemas maiores (ex. 200 ou mais átomos) a solução do sistema de equações pode demorar cerca de 25 minutos em uma estação de trabalho com um desempenho nominal de 2 Mflops. Desta forma torna-se interessante a distribuição, entre as estações disponíveis na rede, destas etapas de cálculo. O desenvolvimento do programa foi realizado em uma rede de 5 estações Sun Sparcstation1+ e uma servidora SparcServer330. Devido a limitações de memória alguns testes foram realizados em um cluster de 8 estações IBM RISC6000/560.

Na construção do vetor gradiente através da distribuição entre diversos processadores não se mostrou eficiente no modelo MPMD de programação pois o tempo necessário para a inicialização dos processos escravos e a distribuição de tarefas entre os processadores é, até 600 átomos, comparável ou maior do que o tempo necessário para a execução dos mesmos. Desta forma o cálculo do vetor gradiente foi atribuido ao processo mestre. A construção da matriz hessiana foi utilizado um critério simples de divisão de tarefas baseado no número de processadores presentes na máquina virtual.

Para a solução do sistema de equações lineares foi desenvolvida uma subrotina baseada no algoritmo de Jacobi no processador mestre e no algoritmo de Gauss-Seidel nos processadores escravos. Como a hessiana utilizada é obtida em coordenadas cartesianas, com número de linhas e colunas múltiplos de 3, o algoritmo fez uso deste fato para simplificar a divisão entre os processadores. Foi observado que este método diverge facilmente em função da magnitude dos elementos diagonais. Por isto foi utilizado um critério de pivoteamento através da troca de índices entre os processadores para a escolha do pivot.

A diagonalização da matriz hessiana ponderada em massas atômicas foi paralelizada inicialmente através do algoritmo proposto por Carbo e colaboradores [46]. Este método, entretanto, mostrou-se instável e extremamente dependente da escolha inicial dos autovetores aproximados. Por isso foi substituido por um método convencional do tipo Householder (HQR) onde as operações mais intensivas são realizadas de forma distribuída mas o processo todo é conduzido pelo programa mestre.

Num. CPU	1	3	6
Total	153,5	65,3	48,7
Hessiana	3,0	1,5	1,1
Sistema linear	25,8	9,6	7,3
Diagonalização	40,7	21,2	15,3

Tabela 4.10: Tempo total em segundos de execução em segundos e tempos parciais gastos para a otimização (4 ciclos) de um agrupamento de 210 átomos do sal $LiOH.2H_2O$ utilizando PVM.

A tabela 4.10 ilustra as melhorias obtidas pela reestruturação do programa para a otimização de geometria e cálculo de frequências de um agrupamento de unidades monômeras de hidróxido de lítio monohidratado com um total de 210 átomos. Os parâmetros moleculares para este cálculo foram determinados a partir das geometrias experimentais obtidas por Raio X , cálculo abinitio para um dímero utilizando base 4-31G e os dados experimentais do espectro infravermelho e Raman.

A partir dos resultados obtidos pode-se concluir que a paralelização do programa resultou em uma redução significativa no tempo total de cálculo. A maior redução foi observada na solução do sistema de equações lineares. Para esta dimensão do problema e da máquina virtual o algoritmo de construção da matriz hessiana não mostrou-se eficiente e o de diagonalização, embora mais efetivo, deve ser reformulado.

4.6.2 Paralelização do Programa FORCES: *shared memory*

Um dos problemas na paralelização do programa FORCES utilizando PVM está relacionada a perda de visibilidade global dos dados que cada processo sofre após a distribuição dos mesmos entre os vários parceiros. Cada atualização envolve um ciclo de sincronização e troca de mensagens cuja granularidade implica diretamente no desempenho. Em outras palavras a atualização constante pode acarretar um dispêndio razoável de tempo no processo de comunicação, uma atualização mais espaçada implica no não aproveitamento total dos recursos disponíveis em virtude do conhecimento reduzido que cada processo possui sobre o andamento do trabalho nos seus demais parceiros. Este problema pode ser removido ou reduzido se todos os parceiros tiverem acesso simultâneo ao conjunto de dados, em outras palavras, se puderem compartilhar o mesmo espaço de memória.

Num. Proc.	1	2	4
Total	40,2	26,3	15,8
Hessiana	3,0	1,5	1,1
Sistema linear	6,5	4,0	2,1
Diagonalização	10,5	11,2	11,3

Tabela 4.11: Tempo total de execução em segundos e tempos parciais gastos para a otimização (4 ciclos) de um agrupamento de 210 átomos do sal $LiOH.2H_2O$ utilizando shared memory.

Visando avaliar a potencialidade desta técnica e para a computação distribuída e tendo em vista a disponibildade cada vez maior de sistemas SMP, o programa FORCES foi reescrito utilizando `mmap()` como mecanismo de compartilhamento de memória. Os dados agora passam a residir na memória virtual do sistema operacional e podem ser acessados por todos os processos simultâneamente. A execução do programa é iniciada por um processo master que se encarrega de inicializar as variáveis compartilhadas e gera seus parceiros através de uma chamada de sistema `fork()`. Cada parceiro pode operar sobre todo o conjunto de dados simultâneamente não necessitando trocar mensagens com os demais dentro de um modelo MIMD. O desenvolvimento do programa foi realizado em uma estação Pentium100 com 32M de RAM rodando FreeBSD2.0.5. A simulação de multiprocessadores foi efetuada levando em conta os tempos individuais de cada processo. Para utilização da funções associadas a `mmap()`, disponíveis apenas em linguagem C, uma pequena biblioteca de *wrappers* para a linguagem Fortran foi desenvolvida.

As principais mudanças no programa dizem respeito à introdução de flags (sinalizadores) de blocos de memória e variáveis compartilhados para evitar a sobreposição de trabalhos e a reescrita dos algoritmos, mais similares a sua forma original, visando o acesso global aos dados. Ao contrário do modelo MPMD utilizando biblioteca de troca de mensagens o modelo MIMD mostrou-se eficiente para a construção do vetor gradiente uma vez que a troca de dados via message passing foi eliminada totalmente e não há inicialização de novos processos para cada tarefa. Da mesma forma a construção da hessiana foi acelerada uma vez que cada processo pode operar sobre os elementos ainda não processados pelos demais. A solução do sistema de equações foi mantida idêntica com aceleração significativa devido a velocidade de intercâmbio dos índices via memória virtual. A diagonalização foi mantida no processo mestre como na técnica de MPMD.

A tabela 4.11 não permite a comparação absoluta com a tabela 4.10 devido as diferenças entre os processadores utilizados, no entanto é possível

constatar a partir dela a otimização de recursos e a redução significativa, por processo, da utilização de memória compartilhada em relação ao PVM. Grande parte desta aceleração pode ser atribuida a remoção completa do tempo gasto na inicialização de processos para cada etapa do cálculo aos procedimentos de empacotamento e transmissão via rede, substituidos aqui pelo acesso direto a memória virtual do sistema. Uma comparação exata só seria possível através da utilização de um sistema SMP verdadeiro, mas os resultados obtidos indicam algoritmos baseados em mmaping como extremamente eficientes, para este tipo de cálculo, em relação àqueles baseados em message passing.

4.6.3 Paralelização do Programa FORCES: MNFS

Conforme já descrito o MNFS é um sistema de shared memory distribuido em clusters utilizando uma interface entre memoria virtual e NFS desenvolvido por Minich[17]. O MNFS utiliza uma interface de programação similar a `mmap()` com a diferença fundamental que a memória de um processo é mapeada em um disco de rede acessível via NFS. O MNFS elimina os efeitos de buffering do servidor e do cliente decorrentes do protocolo NFS permitindo uma granularidade de página de memoria (4k ou 8k) nos acessos aos dados distribuídos entre os parceiros de trabalho. Estudos preliminares realizados neste trabalho[18] mostraram a viabiliadade do MNFS como um ambiente de emulação SMP utilizando clusters Unix. Para utilização do MNFS no programa FORCES foi desenvolvida uma biblioteca de *wrappers* entre a interface C de Minich e a linguagem Fortran77 mantendo-se a mesma estrutura utilizada anteriormente para `mmap()` e utilizando um cluster de 4 estações 486DX33 com 16M de RAM rodando FreeBSD2.0.5. Embora disponível para as plataformas Sun e IBM não foi possível utilizar o MNFS nas mesmas devido a necessidade de uma licensa de código fonte para os sistemas operacionais SunOS e AIX.

Os resultados obtidos mostram que, para 4 processadores, é possivel alcançar-se tempos de sincronização da ordem de 10 milisegundos com relação a alteração de dados globalmente. O acesso e a movimentação de páginas de memória de 4k bytes (page faults) mostrou-se da ordem de 35 milisegundos. Estes valores mostram-se dentro dos valores obtidos para a interface nativa em linguagem C[18]. Por outro lado o mecanismo de inicialização do mmaping mostrou-se até 30% mais lento em relação a implementação em C, não foi possível determinar se isto deve-se aos *wrappers* ou a natureza do compilador Fortran77 empregado (tradutor f2c da AT&T e gcc-2.6.3). Os tempos de comunicação, no entanto, são inferiores àqueles obtidos com PVM para este tamanho de mensagens sendo que este mostra-se adequado para blocos

Num. Proc.	1	2	4
Total	381,2	169,78	126,1
Hessiana	5,3	3,9	2,8
Sistema linear	60,1	32,3	19,2
Diagonalização	101,3	102.1	97,3

Tabela 4.12: Tempo total de execução em segundos e tempos parciais gastos para a otimização (4 ciclos) de um agrupamento de 210 átomos do sal $LiOH.2H_2O$ utilizando MNFS.

de maior tamanho[37].

A adaptação do programa FORCES ao MNFS consistiu basicamente na introdução dos mecanismos de inicialização do MNFS e dos processos escravos, o restante dos algoritmos foram utilizados de forma inalterada. A tabela 4.12 sumariza os resultados obtidos.

Os resultados obtidos expressam um resultado intermediário entre um sistema shared memory e um sistema message passing. A distribuição do entre diversos processadores reduz as penalidades de paginação local e conflitos de cache existentes no programa FORCES com shared memory mas introduz, novamente, a latência de comunicação via rede. Da forma como foi projetado o MNFS, cada *page fault* induz o sistema operacional à leitura de uma nova página de 4k, ou seja, um novo acesso a rede caso esta página não esteja disponível localmente. Da mesma forma cada página escrita por um processo é imediatamente invalidada em todos os demais processadores forçando o mnfs a reconstruir a coerência do espaço de memória compartilhada enviando esta página a todos os participantes da tarefa. Considerando que uma página de 4k corresponde a 1000 variáveis de ponto flutuante ou 512 de precisão dupla é de se esperar que processadores mais rápidos que os utilizados gerem uma alta taxa de invalidação de páginas acarretando uma sobrecarga de sincronização entre os processos. Para evitar isto é necessario que o algoritmo busque adequar a velocidade de processamento e escrita de páginas ao tamanho do cache de dados local de cada participante.

4.7 Discussão e Conclusões

Os resultados obtidos tanto com o PVM com modelo MPMD quanto com técnicas de shared memory e modelo MIMD emulando SMP mostram que a distribuição de uma tarefa computacional entre diversos processos permite produzir uma aceleração limitada pela lei de Amdhal[20]. Os recursos

necessários para exploração da computação distribuída em clusters estão disponíveis de forma aberta e padronizada aos pesquisadores e, na maioria das vezes, gratuitamente. Apenas alguns sistemas de desenvolvimento de aplicações distribuídas encontram-se presos a licenças pelo fabricante (ex. Linda) ou por terceiros (ex. MNFS).

A utilização de computação distribuída em química tem se limitado nos últimos anos a área de química teórica em virtude da sua grande demanda por poder computacional para processamento numérico[38,39] e este trabalho reflete em parte esta tendência. No entanto todas as áreas da química que hoje empregam métodos computacionais podem ser extremamente beneficiadas pelas técnicas descritas não apenas visando a aceleração computacional mas muito mais explorando modelos computacionais cliente-servidor que permitam a operação cooperativa entre vários pesquisadores e vários computadores.

A principal barreira a ser apontada para a utilização mais ampla de computação distribuída em química é a evolução muito rápida nos modelos e ferramentas de programação disponíveis. Isto impede que estas ferramentas atinjam o mercado comercial de microcomputadores pessoais restringindo-as às áreas estritamente ligadas a programação e computação em rede. A estabilização de padrões como MPI, HPF e técnicas como `mmaping` está em andamento e deve reverter ou minimizar esta situação.

No caso específico deste trabalho, a análise dos resultados obtidos e técnicas empregadas apontam o emprego de padrões abertos e bem estabelecidos como Berkeley sockets sobre protocolo TCP/IP, PVM e shared memory como a direção a seguir sempre que se desejar atingir valores ótimos de desempenho e transportabilidade de programas. Sistemas baseados em message passing como o PVM mostraram-se versáteis mas exigem modificações drásticas tanto no modelo de programação quanto nos algoritmos empregados. O desempenho de programas baseados em PVM mostrou-se dependente do ajuste entre as dimensões do problema tratado e do número de processadores disponíveis. A utilização de um modelo SMP por outro lado mostra-se menos intrusiva no que diz respeito as modificações necessárias ao programa e ao algoritmo mas sua eficácia e aplicabilidade depende da disponibilidade de sistemas multiprocessados simétricos com memória compartilhada. Embora estes sistemas já estejam disponíveis comercialmente (ex. Sun, IBM, DEC, etc) o seu uso em ambiente de computação pessoal não será imediato.

4.8 Bibliografia

1. *Pountain, D., Bryan, J.*, **All Systems Go**, Byte, 112, **17**, 8,1992.

2. *Dowd, K.*, **High Performance Computing**, O'Reilly & Associates, Inc., Sebastopol, EUA, 1993.

3. *Bach, M. J., The Design of The UNIX Operating System*, Prentice Hall, EUA, 1990.

4. *Goodheart, B., Cox, J.*, **The Magic Garden Explained, The Internals of UNIX System V Release 4, an Open Systems Design**, Prentice-Hall, Austrália, 1994.

5. *Stevens, W. R.*, **Advanced Programming in the Unix Environment**, Addison-Wesley, EUA, 1993.

6. *Leffler, S. J., McKusick, M. K., Karels, M. J., Quarterman, J. S.*, **The Design and Implementation of the 4.3BSD Unix Operating System**, Addison-Wesley, EUA, 1988.

7. *McKusick, M. K., Bostic, K., Karels, M. J., Quarterman, J. S.*, **The Design and Implementation of the 4.4BSD Operating System**, Addison-Wesley , EUA, 1996.

8. *Sun Microsystems Inc.*, **NFS: Network File System Protocol Version 2 Specification**, RFC1094, Network Information Center, SRI International, Stanford, EUA, 1989.

9. *Minnich, R. G.*, **Mether-NFS: A Modified NFS Which Supports Virtual Shared Memory**, David Sarnof Supercomputing Research Center, SRI International, Stanford, EUA, 1994.

10. *Vazquez, P. A. M.*, **MNFS: programação FORTRAN utilizando memória compartilhada em ambientes de cluster**, 19ª Reunião Anual da SBQ, Poços de Caldas, 1996.

11. *Souto, P., Stark, E. W.*, **A Distributed Shared Memory Facility for FreeBSD**, Proceedings of the USENIX 1997,Technical Conference, Anaheim, California, January 6-10, 1997.

12. *Comer, D. E.*, **Internetworking With TCP/IP, Vol 1: Principles, Protocols, and Architeture** , 3rd ed., Prentice Hall, EUA, 1995.

13. *Sechrest, S.*, **An Introductory 4.4BSD Interprocess Communication Tutorial**, *in* 4.4BSD Programmer's Supplementary Documents, PSD:20, O'Reilly & Associates, Inc. Usenix, Sebastopol, EUA, 1994.

14. *Stevens, W. R.*, **Unix Network Programming**, Prentice Hall, EUA, 1990.

15. *Stevens, W. R.*, **TCP/IP Illustrated: the protocols**, Addison Wesley, EUA, 1994.

16. *Leffler, S. J., Fabry, R. S., Joy, W. N., Lapsley, P.*, **An Advanced 4.4BSD Interprocess Communication Tutorial**, *in* 4.4BSD Programmer's Supplementary Documents, PSD:21, O'Reilly & Associates, Inc. Usenix, Sebastopol, EUA, 1994.

17. *Leffler, S. J., Joy, W. N., Fabry, R. S., M. K., Karels*, **Networking Implementation Notes - 4.4BSD Edition**, *in* 4.4BSD System Manager's Manual, SMM:18, O'Reilly & Associates, Inc. Usenix, Sebastopol, EUA, 1994.

18. *Comer, D. E., Stevens, D. L.*, **Internetworking With TCP/IP, Vol 3: Client-Server Programming And Applications, BSD Socket Version**, 2nd ed., Prentice Hall, EUA, 1996.

19. *Wright, G. R., Stevens, W. R.*, **TCP/IP Illustrated: the implementation**, Addison-Wesley, EUA, 1994.

20. *Stevens, W. R.*, **TCP/IP Illustrated: TCP for transactions, HTTP, NNTP and the Unix Domain Protocols**, Addison-Wesley, EUA, 1994.

21. *Comer, D. E., Stevens, D. L.*, **Internetworking With TCP/IP, Vol 2: Design, Implementation, and Internals**, 2nd ed., Prentice Hall, EUA, 1994.

22. *Software Development Group*, **NCSA HDF Specifications**, NCSA, University of Illinois at Urbana-Champaign, EUA, 1989.

23. *Sun Microsystems Inc.*, **XDR: External Data Representation Standard, RFC1014**, Network Information Center, SRI International, Stanford, EUA, 1987.

24. *Sun Microsystems Inc.*, **XDR: External Data Representation Standard, RFC1832**, Network Information Center, SRI International, Stanford, EUA, 1995.

25. *Wall, L., Christiansen, T, Shwartz, R.*, **Programming Perl**, O'Reilly & Associates, Inc. Usenix, Sebastopol, EUA, 2ed, 1996. disponível em: `http://www.perl.com/src/latest.tar.gz`

26. *Foster, Chandy* , disponível em : *http://www.netlib.org/fortran-m/*

27. *Clark, T. W., Scott, L. R., Bagheri, B.*, **Pfortran a Parallel Extension of Fortran, Reference Manual**, Report UHMD124, University of Houston, Houston, Texas, 1991, EUA.

28. *Sun Microsystems Inc.*, **RPC: Remote Procedure Call Protocol Specification Version 2**, RFC1057, Network Information Center, SRI International, Stanford, EUA, 1988.

29. *Sun Microsystems Inc.*, **RPC: Remote Procedure Call Protocol Specification Version 2, revision**, RFC1831, Network Information Center, SRI International, Stanford, EUA, 1995.

30. *Carriero, N., Gelernter, D.*, **How to Write Parallel Programs**, The MIT Press, EUA, 1990.

31. *Sun Microsystems Inc.*, **NFS: Network File System Protocol Version 3 Specification**, RFC1813, Network Information Center, SRI International, Stanford, EUA, 1995.

32. *Macklem, Rick*, **The 4.4BSD NFS Implementation**, *in* 4.4BSD System Manager's Manual, SMM:6, O'Reilly & Associates, Inc. Usenix, Sebastopol, EUA, 1994.

33. *Harrison, R. J.*, **TCGMSG send/receive routines - version 4.02. User's Manual**, Pacific Northwest Laboratory, 1993.

34. *Beguelin, A., Dongarra, J. J., Geist, G. A., Mancheck, R., Sunderam, V. S.*, **The PVM and HeNCe Projects**, The PVM Distribution Papers, netlib@ornl.gov, Oak Ridge National Laboratory, Oak Ridge, Tennessee,1993,EUA.

35. *Geist, A. et all.*, **PVM 3.0 User's Guide and Reference Manual**, Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1993, EUA.

36. *Geist, A., Beguelin, A., Dongarra, J. J.,et all.*, **PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Parallel Computing**, MIT Press,1993.

37. *Theresa Windus*, Comunicação privada, 1992.

38. MPI Forum. Ver *http://www.mcs.anl.gov/mpi/*

39. *Brandes, T.*, **ADAPTOR, The Distributed Array Library**, High Performance Computing Center, German National Research Institute for Computer Science,GDE, Internal Report No. Adaptor 4, Outubro de 1992.

40. *Brandes, T.*, **ADAPTOR, Users Guide**, High Performance Computing Center, German National Research Institute for Computer Science,GDE, Internal Report No. Adaptor 2, Outubro de 1992.

41. *Cunha, R. D., Hopkins, T.*, **PIM 1.0 The Parallel Iterative Methods Package for Systems of Linear Equations**, University of Kent at Canterbury, Inglaterra, 1992.

42. *Machuca-Herrera, J. O.*, **Mecânica Molecular O Estudo do Campo de Força e sua Aplicação aos Complexos de $SbCl_5$ Com Alguns ligantes $P = O$, $C = O$ e $S = O$**, Tese de Doutoramento, Instituto de Química, UNICAMP, 1988.

43. *Clark, T. W., Scott, L. R., Hanxleden, R. V.*, **Computers and Chemistry** , 219,**14** (1990) 3

44. *gprof - display call graph profile data* 4.4BSD User's Reference Manual, URM:1, O'Reilly & Associates, Inc. Usenix, Sebastopol, EUA, 1994.

45. *Clark, T. W., Kennedy, K., Scott, L. R.*, **Evaluating Parallel Languages For Molecular Dynamics Calculations**, Report UHMD79, University of Houston, Houston, Texas, 1992, EUA.

46. *Carbo, R. , Molino, L, Calabuig, B.*, **Journal of Comp Chem**, 155, 13(1992)2

47. *Vazquez, P. A. M., Morgon, N. H.*, **Velocidade de Transmissão de dados: NFS vs. Paralelismo**, VII Simpósio Brasileiro de Química Teórica, Caxambu, MG, novembro de 1993.

48. *Vazquez, P. A. M., Morgon, N. H.*, **Computação de Alto Desempenho no Ensino de Química** 17ª Reunião Anual da SBQ, Caxambu, maio de 1994.

49. *Vazquez, P. A. M.*,**Paralelization in Chemistry**, 16ª Reunião Anual da SBQ, Caxambu, maio de 1993.

Índice Remissivo