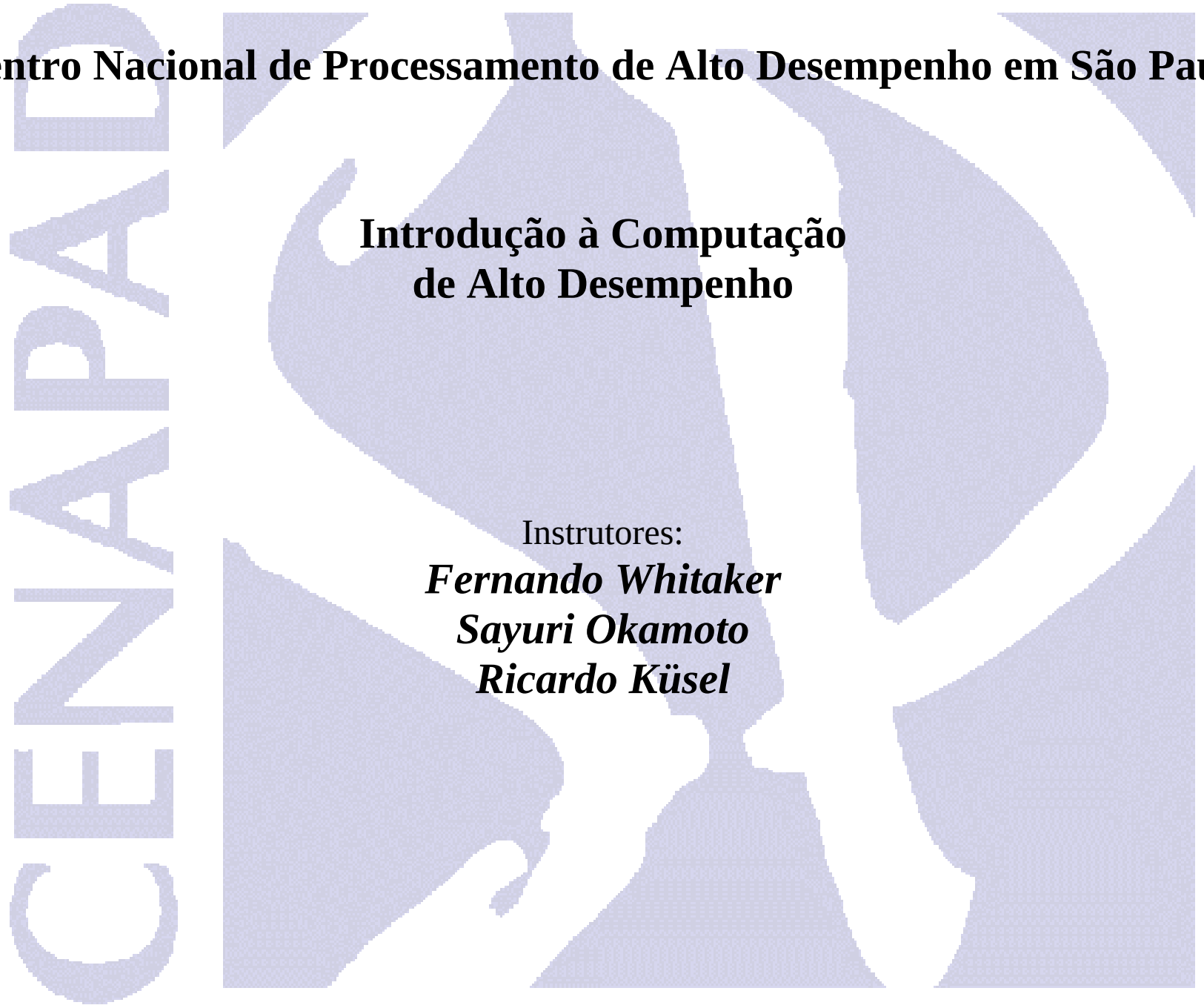


Centro Nacional de Processamento de Alto Desempenho em São Paulo

Introdução à Computação de Alto Desempenho

Instrutores:

Fernando Whitaker

Sayuri Okamoto

Ricardo Küsel

ÍNDICE

CENAPAD-SP

Objetivos

Prestar serviços em computação de alto desempenho, tornando disponível, aos pesquisadores e instituições de ensino e pesquisa, um ambiente computacional poderoso, bem como o suporte necessário para o uso do mesmo.

Serviços

- Equipamentos de alto desempenho computacional
- Suporte técnico
- Treinamentos
- Equipamentos para visualização
- Equipamentos para confecção de apresentações

Anotações

Hierarquia

- FINEP / MCT
- SINAPAD
- CENAPAD's →
  - CENAPAD-SP
  - CENAPAD-RJ
  - CENAPAD-RS
  - CENAPAD-NE
  - CENAPAD-MG

Estrutura Organizacional

- Conselho Diretor
- Diretoria
- Conselho Técnico
- Gerência Técnica
- Grupo de Suporte de Sistemas
  (suporte@cenapad.unicamp.br)
- Grupo de Suporte a Usuários
  (consult@cenapad.unicamp.br)

Anotações

Equipamentos

Ambiente Central

- Máquinas interativas e de visualização — uso interativo
- Processamento serial — batch
- Processamento paralelo — batch

Ambientes Departamentais (IQ e IFGW)

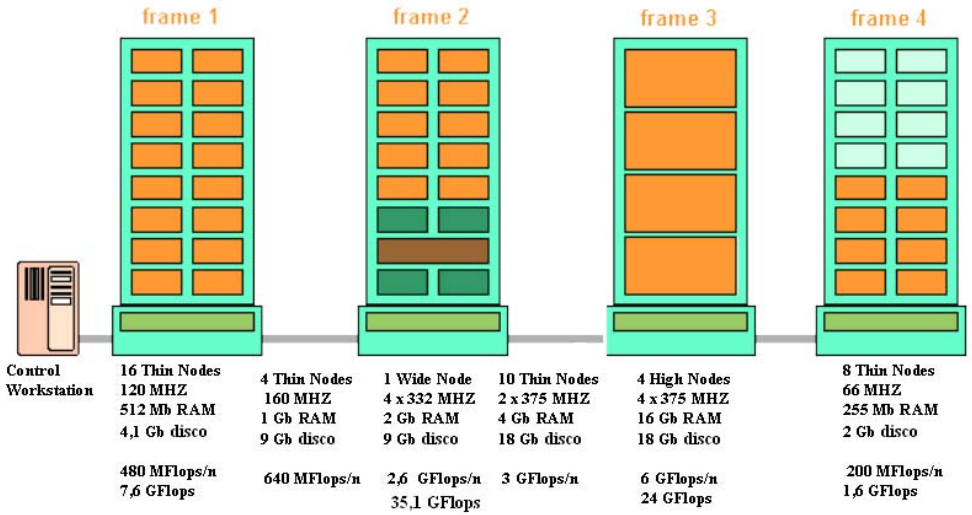
- Máquinas interativas e de visualização — uso interativo
- Processamento serial — batch

Anotações

As máquinas reservadas para uso interativo são as únicas que permitem *logon* remoto e uso interativo. Ao conectar-se ao CENAPAD-SP, o sistema automaticamente abre a sessão na máquina que estiver mais livre, distribuindo assim a carga do sistema entre as máquinas disponíveis.

As máquinas *batch* somente podem ser utilizadas através do sistema de filas, como veremos mais adiante.

Ambiente Paralelo

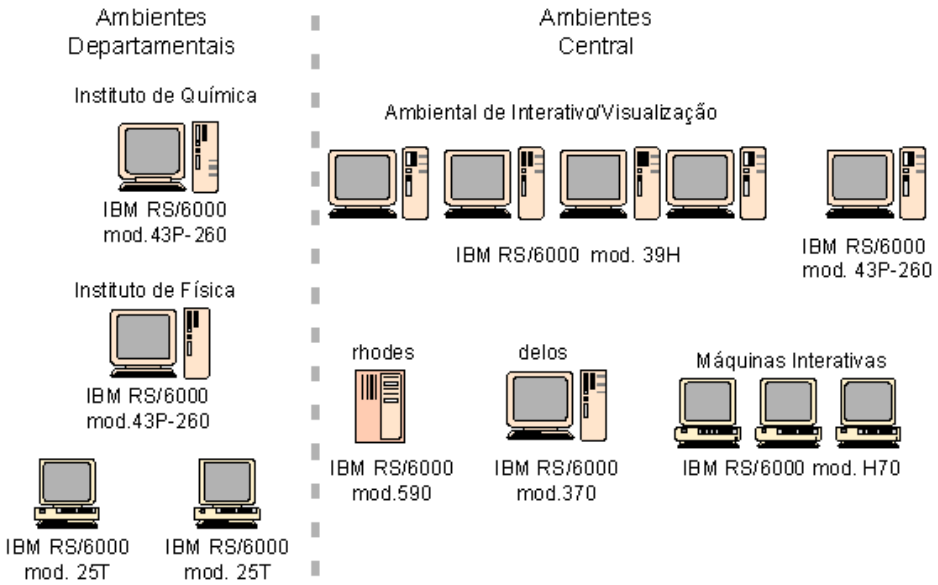


IBM 9076 SP

- *RS/6000*
- *43 nós*
- *70 GFlops*
- *330 GBytes em discos externos*
- *SP Switch: 150 MBytes / s*

Anotações

Ambiente Serial



Equipamentos para Confecção de Apresentações

- Impressora de Cera Sólida Tektronix Phaser 350
- Impressora de Slides Polaroid
- Vídeo Cassete Panasonic AG-1980P

Anotações

Softwares Disponíveis

Sistemas

- AIX 4.3.3 e OSF/1.3.2
- LoadLeveler 2.1
- ADSM6000 3.1

Programação

- IBM – HPF, xlFortran, xlc, xlC, xlPascal, xdb, xldb
- DEC – Fortran 77, C 3.11
- Domínio Público – gcc, g++, libg++, xwpe, wxWindows, Java

Processamento Paralelo

- Ambiente Paralelo (MPI, MPL, PVMe, VT, PEDB)
- PESSL, OSLP
- Domínio Público – mpich, pvm, xpvm, tcgmsg

Anotações

Softwares Disponíveis

Visualização

- Licenciados – Data Explorer, OpenGL, Spartan
- Domínio Público – ansys2dx, xmgr, gnuplot, RasMol, Xmol

Matemática / Engenharia

- Licenciados – BLAS, NAG, Mathematica, MASS, OSL, DXML, ESSL, Ansys
- Domínio Público – LAPACK, LAPACK++, LASPACK, Octave

Física / Química

- Licenciados – CERNLib, Gaussian98, Mulliken, Spartan 5
- Domínio Público – Gamess2000, Mopac 6 e 7, Argus 1.1, Babel 1.1

Anotações

Perfil dos Usuários

Por Área

- Química 40%
- Física 25%
- Engenharias 25%
- Matemática 4%
- Computação 3%
- Outros 3%

Por Instituição

- UNICAMP 65%
- USP 10%
- UNESP 6%
- UFPR 3%
- UFSCar 3%
- CTA 3%
- UFPE 3%
- UERJ 2%
- UNB 2%
- Outros 3%

Anotações

Uso do Ambiente

Alto Desempenho

A cada ano, a indústria de hardware investe alguns bilhões de dólares em busca de computadores mais poderosos. Basicamente, o investimento em novas tecnologias atua em duas frentes:

1. expansão dos limites físicos que determinam a velocidade máxima em que os circuitos elétricos conseguem operar
2. desenvolvimento de novas arquiteturas que permitam obter maior desempenho, dentro dos limites físicos atuais

As máquinas lançadas a cada ano sempre trazem avanços nestes dois aspectos. No primeiro, vemos comumente o lançamento de processadores já conhecidos, porém com um *clock* um pouco mais elevado.

O segundo aspecto refere-se a diferenças estruturais, que podem ir desde mudanças na estrutura interna do processador, passando por mudanças na montagem e conexão deste aos demais dispositivos do computador, ou até mesmo em relação ao uso de vários computadores.

Anotações

Alto Desempenho (continuação)

Uma das principais estratégias utilizadas para obtenção de maior desempenho é o paralelismo. Este termo pode ser aplicado a vários níveis:

- vários computadores trabalhando em conjunto para realizar uma determinada tarefa;
- dentro de um mesmo computador, que pode ter vários processadores;[1]
- dentro de cada processador, que pode ter várias unidades de execução para executar instruções em paralelo.[2]

Paralelismo:
- Solução — *ultrapassa o limite do processador individual*
- Tendência — *Escalabilidade é um fator importante devido à flexibilização de custos*
- SMP x MPP — *simétrico X massivo*

Quanto à forma de utilização de memória, há dois "tipos" de paralelismo: SMP (*Symmetric Multi-Processing*) e MPP (*Massive Parallel Processing*).

[1] Não confundir este conceito com *multitasking* e *multithreading*!

[2] Neste texto, trataremos apenas de paralelismo em nível "macro", ou seja, vários computadores ou um computador com vários processadores.

Anotações

SMP x MPP

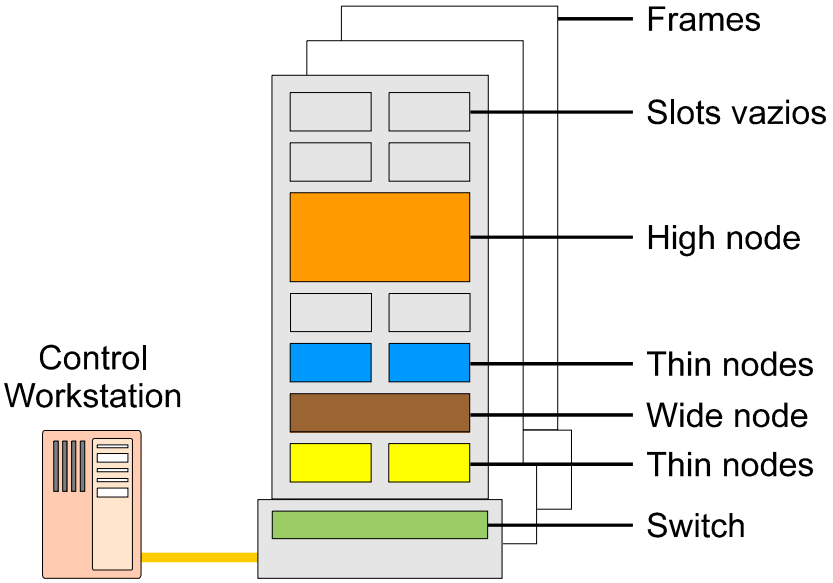
Sistemas SMP (*Symmetric Multi Processing*) possuem mais de um processador em um mesmo computador. Todos eles compartilham os recursos de memória e disco existentes, segundo uma política de controle de concorrência adotada pelo sistema operacional. Esta complexa arquitetura, entretanto, é bastante transparente para o desenvolvimento de aplicações, pois a maior parte da complexidade fica a cargo do sistema operacional.

Em sistemas MPP (*Massive Parallel Processing*), os processadores possuem maior independência entre si, havendo pouco ou nenhum compartilhamento de recursos. Tipicamente, cada *node* de um sistema MPP é um computador independente, com memória e disco próprios. O controle do paralelismo é feito pela aplicação, que deve coordenar a distribuição de tarefas e a coerência entre os diversos *nodes*.

Cada alternativa apresenta vantagens e desvantagens. A complexidade de ambos cresce à medida que aumenta o número de *nodes*/processadores. Em SMP, um elevado número de processadores resulta em menor desempenho por processador, pois os mesmos recursos precisam ser compartilhados por todos eles. Em MPP, cada *node* trabalha com memória e disco próprios, o que permite a utilização de muitos processadores. Por isso mesmo, a aplicação tem que manter a coerência entre todos e garantir a distribuição eficiente do trabalho – caso contrário, pode haver desperdício do poder de processamento total.

Anotações

IBM SP – Arquitetura



O IBM 9076 SP (*Scalable POWERParallel*) utiliza uma combinação das arquiteturas SMP e MPP. Sua configuração pode variar de 1 até 512 *nodes* (arquitetura MPP), sendo que cada um deles é uma máquina RISC completa, podendo inclusive ter mais de um processador (arquitetura SMP).

Há vários modelos de nodes e configurações possíveis. Os nodes são montados em gabinetes específicos, os *frames*. Cada *frame* suporta até 16 *nodes*, e a ligação de vários *frames* permite atingir um elevado número de máquinas.

Os *nodes* são conectados através de uma rede de altíssima velocidade, o *SP Switch*, que atinge uma taxa de transferência de 150 MBytes/s.

Anotações

O custo de gerenciamento não é proporcional ao número de *nodes*, pois praticamente toda a administração do sistem apode ser feita em paralelo, a partir da *Control Workstation*.

O SP suporta o uso de *nodes* com configurações diferentes em um mesmo sistema (MPP). Isto é um ponto bastante positivo, pois permite que sejam incluídos modelos novos sem exigir a atualização dos já existentes.

Sistema de Filas

Para utilizar as máquinas do ambiente de processamento batch, que são as mais poderosas do CENAPAD-SP, cada usuário precisa submeter seus jobs ao Sistema de Filas. Este sistema aloca os processos de acordo com a disponibilidade e características de cada máquina, evitando que um número excessivo de processos sobrecarregue uma máquina e cause perda de desempenho.

O sistema está dividido em quatro **classes** de filas:

Classe	Prioridade	Limite de CPU
grande	20	60 dias
média	50	24 horas
pequena	100	1 hora
paralela	200	7 dias

A classe de cada job deve ser escolhida estimando-se o tempo de CPU necessário para concluí-lo.

Todo job que ultrapassar o limite de CPU de sua respectiva fila é automaticamente cancelado pelo sistema.

Esta configuração visa priorizar os jobs paralelos, que aproveitam melhor a arquitetura do IBM 9076 SP, e os jobs menores.

Anotações

Sistema de Filas (continuação)

Para cada job, também podem-se especificar os seguintes requisitos:

Feature	Tipo	Descrição
workssa	disco	Somente máquinas com acesso aos discos externos SSA (50 GB)
sp	máquina	Somente nodes do SP
iq		Máquinas do ambiente departamental do IQ
power2		Máquinas com arquitetura POWER2 ou superior
smp		Máquinas multiprocessadas (SMP)
gaussian	software	Uso do software Gaussian98
spartan(*)		Uso do software Spartan
oslp		Uso da biblioteca OSL

(*): Os requisitos de software são obrigatórios caso se deseje executar estes programas.

Anotações

Cada máquina *batch* possui uma área de trabalho local (diretório /work) com 1 a 4 GBytes, que é utilizada por default pelos programas. Como este diretório é local, seu acesso é mais rápido que, por exemplo, os diretórios *home*. Entretanto, jobs que necessitem de mais espaço devem especificar o diretório /workssa, que é compartilhado por todas as máquinas do Ambiente Central e tem 50 GB de espaço total.

Alguns softwares utilizam licenças de número limitado e, por isso, não estão disponíveis em todas as máquinas do ambiente. Neste caso, o software deve ser especificado na submissão do job, para que o sistema de filas o aloque para uma máquina adequada.

Sistema de Filas (continuação)

O sistema de filas controla o andamento dos jobs, armazenando o estado de cada um deles:

Status	Descrição
NQ (Not Queued)	Não passível de iniciar a execução
I (Idle)	Passível de iniciar a execução
ST (Starting)	Iniciando a execução
R (Running)	Em execução
C (Completed)	Terminado, aguardando remoção

Assim que um job é submetido, ele pode entrar em estado “NQ” ou “I”, dependendo do número de jobs que o usuário já possui. Quando um job é concluído, ele é removido do sistema de filas.

Os arquivos produzidos pelos jobs são tratados da seguinte maneira:

- arquivos gravados nos diretórios home dos usuários são mantidos
- arquivos gravados nos diretórios de trabalho temporários (/work e /workssa) são mantidos por 7 dias, contados a partir da data+hora da última atualização
- arquivos gravados nos diretórios /tmp são mantidos por apenas 1 dia

Anotações

É sempre útil fazer com que os jobs mais demorados gravem em um arquivo de saída algum tipo de informação que permita acompanhar seu progresso. Isto evita que um erro de lógica ou de sistema faça com que um job fique “parado” em uma máquina sem dar nenhuma indicação de que houve algum problema.

Antes de submeter jobs longos, é interessante, se possível, submetê-los para a classe “pequena’, apenas para testar a lógica do programa e a sintaxe do arquivo de comandos. Se o job começar a rodar corretamente, você pode submetê-lo com todos os dados para uma das classes maiores.

Sistema de Filas (continuação)

Os critérios utilizados pelo sistema de filas para alocação de jobs são os seguintes:

- não há limite para o número de jobs submetidos
- máximo de 4 jobs em “R” ou “I” por usuário
- classe do job – jobs menores têm prioridade
- data da submissão
- quantidade de jobs do usuário em “R” ou “I”
- máximo de 20 jobs grandes em todo o ambiente

Após cerca de dois dias na fila, a data de submissão passa a ter peso maior em relação aos demais critérios, para evitar que um job em condição desfavorável nunca inicie a execução.

Os limites estabelecidos podem ser aumentados ou eliminados conforme aumentar a disponibilidade de recursos – por exemplo, com a instalação de novas máquinas ou aquisição de novas licenças de softwares.

Em resumo, o sistema procura priorizar jobs paralelos, jobs menores, jobs antigos, e usuários que possuem menos jobs em execução.

Anotações

Uso Remoto

O acesso remoto ao ambiente CENAPAD-SP pode ser feito através dos seguintes serviços:

ssh – "*secure shell*": Conexão shell remota (linha de comando), utilizando encriptação. O ssh é o substituto do ***telnet***, que foi desativado por motivos de segurança. No **ssh**, toda a comunicação é codificada, impedindo que outra pessoa veja o conteúdo da transmissão (por exemplo, sua senha). Para acessá-lo, instale-o em seu computador e rode

ssh cenapad.unicamp.br -l *username*

Pode ser necessário setar a variável TERM de acordo com o seu terminal ou ambiente de janelas. Isto ocorre porque a comunicação utiliza códigos de controle, que representam coisas como as teclas de setas e comandos de posicionamento do cursor na tela. Como estes códigos podem variar conforme o sistema utilizado, você deve setar a variável TERM para indicar o tipo de terminal correto. Os mais comuns são:

xterm	vt100	aixterm
vt52	vt220	dtterm

Para selecionar um tipo de terminal, utilize

setenv TERM *tipo-do-terminal*[3]

[3] Esta sintaxe é válida para **csh** e **tcsh**. Se você estiver utilizando **ksh** (Korn Shell), utilize: `export TERM=<tipo-do-terminal>`

Anotações

O *logon* através do **telnet** ou **ssh** funciona como se você estivesse usando o próprio monitor e teclado da máquina remota[4]. Tudo que você digita é processado nela, e não no seu computador (local), que é apenas o terminal.

Por este motivo, é possível utilizar um ambiente remoto poderoso dispondo de um microcomputador básico, pois este precisa apenas rodar o ssh-cliente para se conectar ao sistema remoto.

Para executar remotamente aplicativos que possuam interface gráfica, entretanto, seu computador local (terminal) precisa suportar conexões "X-Windows" (ambiente de janelas). Este protocolo permite que a tela gráfica do aplicativo, que está rodando na máquina remota, seja mostrada na sua máquina local.

O Windows não possui suporte nativo para conexões "X", mas existem aplicativos que implementam esta função. Unix possui suporte inerente, desde que se esteja utilizando ambiente gráfico.

[4] Tecnicamente, há diferenças, porém elas não fazem parte do escopo deste treinamento.

Uso Remoto (continuação)

scp – "secure copy": É o equivalente seguro do serviço **ftp**, também desativado por motivos de segurança[5]. A sintaxe básica do scp é a seguinte:

scp [[user@]host1:]filename1... [[user@]host2:]filename2

ou seja,

scp *origem destino*

Os colchetes indicam parâmetros opcionais. Entretanto, se o seu username na sua máquina não for o mesmo do ambiente CENAPAD-SP, será preciso especificar o parâmetro "user", pois, caso contrário, o scp tentará conectar-se utilizando o username errado.

[5] Obs: o serviço de **ftp anônimo, que permite acessar os diretórios públicos,** continua disponível (ftp.cenapad.unicamp.br); não sendo permitido, entretanto, utilizar remotamente o ftp de forma identificada, pelos motivos descritos.

Anotações

Assim como para o ssh x telnet, utilize sempre o scp em lugar do ftp. Ao utilizar sistemas remotos, haverá muitas situações em que você precisará digitar sua senha para efetuar alguma operação. No telnet e ftp, a senha é transmitida sem codificação, de modo que um hacker poderia facilmente capturá-la no trajeto entre a sua máquina local e a máquina remota. O ssh e scp utilizam encriptação, evitando assim que outra pessoa tenha acesso à sua conexão.

Uso Remoto (continuação)

WWW – "World Wide Web": O serviço WWW do CENAPAD-SP é acessível, sem restrições, pelo endereço

http://www.cenapad.unicamp.br

No serviço WWW encontram-se dados institucionais sobre o CENAPAD-SP, bem como instruções detalhadas sobre o uso do ambiente. Destaque para os seguintes assuntos:

- Equipamentos: características e configurações
- Softwares instalados
- FAQ (*Frequently Asked Questions*)
- LoadLeveler – sistema de filas
- Treinamentos oferecidos

Anotações

Unix

O CENAPAD-SP utiliza em suas máquinas os sistemas operacionais AIX versão 4.3.2, nas máquinas IBM, e OSF/1.3.2, na máquina DEC.

Ambos são as implementações Unix dos respectivos fabricantes, sendo, portanto, sistemas multitarefa e multiusuário, que oferecem o que existe de mais avançado em termos de sistemas operacionais de alto desempenho.

A seguir, é dada uma lista de comandos básicos do Unix. Para obter mais referências, consulte o Guia do Usuário CENAPAD-SP, também disponível em nossa home page (http://www.cenapad.unicamp.br).

Anotações

Unix – Alguns dos comandos mais utilizados

Obtendo Ajuda

man <*comando*> → apresenta o texto da ajuda para o comando especificado

Ex: man ls

apropos <*substring*> → lista os comandos que contêm a *substring* como parte do nome

Ex: apropos mkdir
apropos ls | more

Manipulação de Arquivos e Diretórios

cd <*diretório*> → muda para o <*diretório*>
Ex: cd /usr/local
cd ..
cd ~
cd ~pvm3

pwd → informa o diretório corrente

Anotações

Em Unix, o símbolo "~" (til) representa o seu diretório principal (home) [6]. Assim, quando você utiliza "~/teste", você está se referindo ao arquivo de nome "teste" que está localizado no seu diretório principal. "~/Mail/mailbox" é o arquivo de nome "mailbox", localizado no diretório "Mail", que por sua vez localiza-se no seu diretório principal.

[6] A localização “física” do diretório *home* varia conforme a instalação do sistema (ela é definida pelo administrador da rede). No CENAPAD-SP, o diretório *home* pode sempre ser acessado pelo caminho `/u/<username>`.

Unix – Manipulação de Arquivos e Diretórios (cont.)

ls [*opções*] <*arquivos*> → lista os arquivos/diretórios do diretório corrente

Ex: ls /tmp
ls –lF
ls –lF /usr/local/bin
ls ~/porcent*

file <*arquivo*> → informa o tipo do <*arquivo*>

more <*arquivo*> → mostra o conteúdo do <*arquivo*>, tela a tela

Ex: more ~/etc/passwd
more ~/.cshrc
ps –ef | more
ls –lF /usr/local/bin | more

mkdir <*diretório*> → cria um diretório

Ex: mkdir ~/teste
mkdir /tmp/temporario
mkdir outro

rmdir <*diretório*> → remove um diretório

Anotações

Não execute o comando more (ou o comando cat, que mostra um arquivo sem parar a cada tela) com arquivos binários, pois eles não contém dados em formato legível. Em caso de dúvida, verifique antes o formato do arquivo com o comando file.

Unix – Manipulação de Arquivos e Diretórios (cont.)

cp <*origem*> <*destino*> → copia arquivos e/ou diretórios

Ex.: execute a seqüência abaixo:

```
ls –lF /usr/local > ~/arquivo.txt
cp ~/arquivo.txt /tmp
mkdir /tmp/testando
cp ~/* /tmp/testando
```

mv <*origem*> <*destino*> → move arquivos e/ou diretórios (idem ao anterior, porém apaga o original)

rm <*arquivo*> → apaga arquivo(s)/diretório(s)

head [*opções*] <*arquivo*> → mostra as linhas iniciais do <*arquivo*>

Ex: head ~/.cshrc
head –2 ~/.cshrc

tail [*opções*] <*arquivo*> → mostra as linhas finais do <*arquivo*>

Ex: tail ~/.cshrc

head –2 ~/.cshrc

Anotações

Cuidado com os comandos cp, mv e rm. Os dois primeiros podem sobrepor um arquivo já existente, e o terceiro apagará definitivamente o(s) arquivo(s) especificado(s).

Um erro clássico é digitar "rm *.o", mas bater por engano o espaço no lugar do ponto. Se não houver um arquivo cujo nome seja somente "o", irá aparecer a seguinte mensagem:

```
rm: o: No such file or directory
```

Os comandos Unix são muito poderosos quando usados em conjunto. Tendo assimilado o funcionamento do pipe "|" e a forma como os comandos interagem, pode-se executar operações complexas utilizando comandos que cabem em uma ou duas linhas.

Manipulação de Arquivos e Diretórios (continuação)

chmod <*perm*> <*arq*> → muda permissões de acesso a arquivos

Ex: cd
ls –lF /usr/local > arquivo.txt
ls –l arquivo.txt
chmod o-w arquivo.txt
ls –l arquivo.txt

diff <*arq1*> <*arq2*> → compara dois arquivos, mostrando as diferenças

Controle de Processos

ps [*opções*] → lista processos em execução

Ex: ps –ef
ps –furoot

kill [*tipo-sinal*] <*proc-id*> → "mata" um processo

Ex: ps –fuminame

```
UID   PID  PPID C    STIME   TTY  TIME CMD
miname 3374 46624 7  17:38:43 pts/1  0:00 ps -fuminame
miname 39928 51956 0 12:57:07 pts/1  0:00 –csh
```

kill –9 39928

Anotações

Cuidado! Se você deixar algum arquivo com permissão de escrita para "todos", qualquer usuário do sistema pode apagá-lo ou alterar seu conteúdo. Analogamente, não deixe arquivos confidenciais com permissão de escrita para "todos" – ou mesmo para o grupo, se for o caso.

As permissões são apresentadas em três grupos:

rwx rwx rwx - read, write, execute
(o) (g) (a) - owner, group, all

Evite encerrar programas com o comando "kill", caso haja outra alternativa, pois isto pode resultar em inconsistências diversas – por exemplo, arquivos de lock que normalmente seriam descartados ao final da execução poderão não ser apagados, caso o programa seja cancelado abruptamente.

Uso Remoto

ssh <*host*> [*comando*] → Efetua logon (ou executa o *comando*) na máquina especificada

Ex: ssh delos
ssh delos ls /tmp

logout → encerra a sessão

scp <*origem*> <*destino*> → Transfere arquivos entre duas máquinas, de forma segura

Outros Comandos

history [-*n*] → lista os últimos *n* comandos executados

passwd → troca a senha

Editores de Texto

nedit → editor simples (ambiente gráfico)

pico → editor simples (modo texto)

vi → editor de texto

Anotações

Em ambientes que utilizam NIS ou FileCollection, existe uma máquina central que gerencia as senhas de usuários; as demais atualizam estas informações a partir dela. No CENAPAD-SP, esta máquina é a “leros”. Portanto, é preciso conectar-se a ela para executar o comando *passwd*.

O **nedit** é um editor que possui interface gráfica. Portanto, para utiliza-lo é preciso que sua máquina local possua suporte a conexões “X-Windows”.

X-Windows

Para executar aplicações que possuam interface gráfica, é preciso que sua máquina local suporte o protocolo *X-Windows*, que é padrão para interfaces gráficas em Unix[7].

Este ambiente permite que você rode programas em uma determinada máquina, mas veja os resultados (janelas) em outra. Por exemplo, pode-se executar programas nas máquinas do CENAPAD-SP e fazer com que suas janelas sejam mostradas na sua máquina local.

Para tanto, você precisa fazê-la aceitar conexões deste tipo. Execute, na sua máquina local, o comando

xhost + *máquina-remota*

Este comando libera sua máquina para aceitar a conexão X solicitada pela máquina remota. Caso contrário, ela não vai aceitar as janelas que lhe forem enviadas.

Uma vez conectado à máquina remota (com o **ssh**), indique onde será apresentada a interface gráfica, com o comando

setenv DISPLAY *máquina-local*:0.0

Isto faz com que que as janelas não sejam mostradas na própria máquina remota, mas sim na que você indicou.

Lembre-se que interfaces gráficas produzem um tráfego de rede considerável, portanto conexões lentas podem tornar seu uso inviável.

[7] Não confunda com Microsoft Windows, que não possui suporte nativo ao X-Windows.

Anotações

Se você não especificar a *máquina-remota*, o ambiente de janelas passará a aceitar janelas enviadas por qualquer máquina na Internet, de qualquer lugar do mundo. Isso representa um potencial problema de segurança, pois é mais um serviço que atacantes externos dispõem para explorar.

Quando você utiliza o **ssh** para conectar-se a uma máquina remota, ele automaticamente configura a variável “DISPLAY” – desde que você esteja na *shell* primária. Caso contrário, basta setar o DISPLAY conforme indicado.

Laboratório

Este tópico apresenta informações básicas sobre o uso do sistema de filas (LoadLeveler).

1. Libere a apresentação de janelas em sua máquina:

   hostname (anote o nome da sua máquina)
   xhost +

2. Conecte-se ao CENAPAD-SP, com o comando

   ssh cenapad.unicamp.br –l username[8]

4. Abra um editor de textos, por exemplo:

   pico

5. Crie um programa, por exemplo o “Hello, world!” em C:

```
#include <stdio.h>
main() {
   printf (“Hello, world!\n”);
}
```

   Salve o arquivo, por exemplo, com o nome “hello.c”.

6. Compile-o, com o compilador apropriado:

   gcc hello.c -o hello

[8] Para mostrar o seu *username*, existe o comando “whoami”.

Anotações

Sua máquina local poderá não estar aceitando conexões “X” externas, ou seja, não está permitindo que janelas de aplicações remotas sejam apresentadas nela. Neste caso, habilite o recebimento de conexões X com o comando

xhost + *nome-da-máquina-remota*

O uso do “&” ao final do comando faz com que ele seja executado em *background*, ou seja, ele fica rodando, porém não “prende” a linha de comando. Rode o comando “ps” para ver quais são os processos que estão em execução.

Se a compilação terminou sem problemas, deve ter sido gerado o executável chamado “hello”. Verifique com o comando “**ls -l**”, que deve listar, entre outros, o arquivo “hello”, que deve estar com permissão de execução. Como este programa é “leve”, você pode executá-lo interativamente, digitando “./hello”.

Laboratório (continuação)

7. Crie um script de submissão, por exemplo:

```
#
# @ executable = /u/<seu-login>/hello
# @ output = hello.output
# @ error = hello.error
# @ class = pequena
# @ queue
#
```

Salve o arquivo, por exemplo, com o nome “hello.cmd”.

8. Submeta o job para o sistema de filas (LoadLeveler):

llsubmit hello.cmd

9. Acompanhe o andamento do job, executando **logo em seguida** o comando:

llq -u username[9]

10. Quando o job tiver terminado, verifique os arquivos gerados:

more hello.output
more hello.error

[9] Para mostrar o seu *username*, existe o comando “whoami”.

Anotações

Os arquivos de saída especificados com “output” e “err” servem para armazenar as mensagens que o programa apresentaria na tela, ou mensagens de erro. Lembre-se que, na execução em *batch*, não haverá tela para apresentação de mensagens, nem teclado para entrada de dados.

Este comando apenas submete o job para o sistema de filas. Este somente começará a executar quando houver uma máquina livre que atenda às características especificadas (no caso deste exemplo, a única exigência é que a classe seja a “pequena”).

Referências

Centro Nacional de Processamento de Alto Desempenho em São Paulo – CENAPAD-SP

http://www.cenapad.unicamp.br

IBM – International Business Machines Inc.

http://www.rs6000.ibm.com

What is 64-bit computing?

http://www.rs6000.ibm.com/resource/technology/64bit6.html#topic3

SP System Overview

http://www.rs6000.ibm.com/hardware/largescale/index.html

suporte@cenapad.unicamp.br
consult@cenapad.unicamp.br

Anotações

1) Computação Científica

Embora não exista uma definição formal para computação científica, esta terminologia pode ser utilizada para designar o uso de computadores aplicados à análise e à solução de problemas científicos. Ao contrário da ciência da computação, que se dedica ao estudo de computadores e da computação, a computação científica dedica-se a encontrar maneiras de disponibilizar recursos de hardware e software a serviço de uma outra ciência. Por requerer de seus usuários conhecimentos em lógica computacional, em adição ao conhecimento da área científica ao qual será aplicada, pode-se dizer que a computação científica conjuga várias áreas, devendo ser encarada, portanto, como uma metodologia comum a várias ciências, que faz uso de algumas ferramentas específicas.

A utilização da computação científica não se restringe às áreas diretamente relacionadas às ciências exatas e tecnológicas. Sua aplicação é destinada a qualquer ciência que utilize tecnologia de alto desempenho no avanço do conhecimento dos seus respectivos campos de estudo, sendo indicada no tratamento de problemas que não tenham formulação matemática precisa, ou cuja solução analítica não seja simples. A computação científica também fornece, em forma de simulação numérica, complementos às teorias e testes de laboratórios, preenchendo a lacuna entre a prática e a abordagem analítica e fornecendo inferências qualitativas e quantitativas sobre o fenômeno estudado. Em muitos casos, estes fenômenos são muito complexos para serem tratados por métodos analíticos, ou muito caros e perigosos para serem estudados através de experimentos.

Como a utilização da computação científica está vinculada a implementações computacionais, o seu desempenho está condicionado ao desempenho do hardware e do software empregados. Em busca deste objetivo, a evolução dos equipamentos vem sendo acompanhada do desenvolvimento de métodos que, implementados em linguagens de programação científica, tentam solucionar problemas, buscando, ao mesmo tempo, o melhor desempenho das máquinas. Desta necessidade, surgiram métodos com a FFT, amplamente utilizada no processamento digital de sinais, e o método de Hartree-Fock, utilizado em química computacional. Atualmente, as linguagens de programação mais utilizadas na implementação de algoritmos para computação científica são o Fortran77, o Fotran90, o C e o C++.

O Fortran é uma linguagem voltada para programação científica, que possui várias funções matemáticas já implementadas, bem como bibliotecas científicas para cálculos específicos, como o LAPACK, o NAG e o ESSL, e biblioteca para programação em paralelo, como PVM e MPI. O Fortran90 é uma evolução desta linguagem, com avanços significativos como, por exemplo, a inclusão de comandos que possibilitam a programação orientada a objetos e a simplificação da sintaxe de “arrays”.

Embora concebida para o desenvolvimento de sistemas operacionais nos Laboratórios Bell, a linguagem C tornou-se muito popular no meio científico, devido à sua enorme praticidade, portabilidade e potencialidade na programação científica. O C++, concebido a partir da filosofia da programação orientada a objetos, conservou em seu núcleo as palavras chaves do C e introduziu outras mais, para a programação orientada a objeto. Também estão disponíveis para o C e C++ algumas bibliotecas científicas como CLAPACK, LAPACK++, UMFPACK, NAG e ESSL, e bibliotecas para programação em paralelo, como PVM e MPI. Além destas linguagens, a computação científica também utiliza softwares para visualização científica de dados, bibliotecas que disponibilizam recursos visuais e tornam a interface com o usuário do programa final mais amigável (Open GL, WxWindows), bem como estratégias para otimização do processamento, como o paralelismo, por exemplo.

Com base nestas afirmações, pode-se deduzir que, para utilizar a computação científica é preciso estar familiarizado com arquiteturas de computadores e estruturas de dados associados a estas arquiteturas, ter um bom entendimento da análise e da implementação de algoritmos numéricos, bem como da maneira como eles tratam estas estruturas de dados, e da arquitetura dos computadores. Assim, arriscando uma definição, pode-se dizer:

"Computação científica é a junção de três áreas com o propósito de um melhor entendimento de alguns fenômenos: o problema científico, a arquitetura do computador a ser usado e o algoritmo a ser implementado".

2) Linguagens de Programação

Uma linguagem de programação é um conjunto de palavras e de regras de sintaxe, utilizado para instruir um computador a executar uma determinada tarefa. Estas instruções transmitidas ao computador através de um programa, que pode ser entendido como um veículo de comunicação entre o homem e a máquina.

Independentemente da finalidade e da forma como acontece, somente se pode estabelecer comunicação se as partes envolvidas na mesma forem capazes de entender o código utilizado nela. Com base neste princípio, foi desenvolvida a primeira linguagem de programação: a linguagem de máquina. A linguagem de máquina é composta por um conjunto de instruções simples, baseadas nas operações de armazenamento e leitura de dados no banco de memória de um processador e representadas por um código binário. A linguagem de máquina é o único código inteligível pelas CPUs, que possuem, cada uma, a sua própria linguagem de máquina.

Embora os primeiros programas de computador tenham sido escritos em linguagem de máquina, esta notação era extremamente complexa e passível de erros, por ser representada por código binário. Portanto, a comunicação estabelecida pela linguagem de máquina era eficaz, embora pouco eficiente, já que os programadores tinham muito pouca familiaridade com este código. Por este motivo, a linguagem de máquina foi substituída por uma nova linguagem de programação, chamada linguagem assembly. Similarmente à linguagem de máquina, o assembly é formado por um conjunto de instruções simples, que se traduzem diretamente em linguagem de máquina. Em vez de código binário, no entanto, o assembly utiliza um código mais amigável, formado por nomes.

Antes de 1954, todo programa de computador era feito em assembly. Entretanto, mesmo sendo mais simples que a linguagem de máquina, logo foi constatado que as linguagens assembly poderiam ser melhoradas, já que não conseguiam representar fórmulas matemáticas de forma natural, além de não serem portáveis, já que cada família de processadores tinha o seu próprio código assembly. A idéia era desenvolver uma maneira mais econômica e eficiente de se programar. Como resultado do experimento realizado por Jonh Backus surgiu o Fortran (uma abreviação para Formula Translation System), a primeira linguagem de alto nível desenvolvida. O termo linguagem de programação normalmente se refere às linguagens de alto nível (Fortran, C, Pascal, Basic, etc), que são linguagens cuja sintaxe é mais próxima à linguagem humana.

Como os processadores entendem somente códigos binários, todo programa escrito em assembly e em linguagem de alto nível precisa ser traduzido para linguagem de máquina. Por isso, a evolução das linguagens de programação, da linguagem de máquina até as linguagens de alto nível, precisou ser acompanhada do desenvolvimento de instrumentos que intermediassem a comunicação entre homem e máquina, atuando

como tradutores. Esta é a função dos interpretadores e dos compiladores, comentados a seguir.

2.1) Compiladores e Interpretadores

Existem duas maneiras de executar programas escritos em linguagem de alto nível. A mais comum é compilar o programa; outra maneira é submeter o programa a um interpretador. Um interpretador traduz, executando em seguida, cada linha de um programa escrito em linguagem de alto nível. Em vez de disso, um compilador analisa e traduz inteiramente o programa escrito em linguagem de alto nível (código fonte), criando, ao final deste processo, um arquivo executável.

Como o processo de compilação envolve uma cuidadosa análise e reorganização das instruções, visando a otimização do código, os compiladores geram programas muito mais rápidos. Por outro lado, o processo de compilação pode demandar um tempo considerável, principalmente no caso de se estar compilando códigos fontes muito longos. Assim, durante o desenvolvimento de programas, quando o programador deseja efetuar pequenas mudanças e aferir o resultado das mesmas, o mais rápido possível, a utilização de interpretadores é mais indicada.

Há vários compiladores diferentes para uma mesma linguagem. Alguns destes compiladores, normalmente de domínio público, funcionam em plataformas diferentes, sendo os outros específicos para uma determinada plataforma. Se um compilador se enquadrar na forma padrão da linguagem, pode-se esperar que um programa escrito nesta forma padrão possa ser compilado em várias plataformas diferentes, produzindo satisfatoriamente o programa executável. Também é razoável esperar que o compilador para uma plataforma específica tenha o seu funcionamento otimizado nesta plataforma. É importante lembrar que há versões de compiladores de uma mesma linguagem, desenvolvidas para diferentes plataformas, porque cada plataforma aceita um código objeto específico.

2.2) Características das Linguagens de Programação

Por sua estrutura refletir o conjunto de instruções e a arquitetura da CPU, o código assembly é considerado a linguagem de mais baixo nível. Embora não produza códigos portáveis e não seja simples de utilizar, por estar bem perto do hardware do sistema, esta linguagem utiliza mais eficientemente os recursos da máquina, sendo, por isso, utilizada ainda hoje em aplicações bem específicas, que requerem grande interatividade com a arquitetura utilizada.

Por fornecerem um alto grau de abstração do hardware e oferecerem maior portabilidade na construção de programas, as linguagens de alto nível tais como o Fortran 90, C++, ADA, Java e Pascal, tornaram-se muito populares. Como cada uma destas linguagens tem características muito particulares, é difícil eleger uma como a melhor. Discussões sobre este assunto costumam apontar a superioridade do Fortran no processamento numérico de dados, a flexibilidade da linguagem C e o enfoque das características orientadas ao objeto do C++. Independentemente dos atributos que a diferencie das demais, uma boa linguagem de programação precisa ter:

- Sintaxe precisa, que permita a construção de rotinas livres de ambigüidades

- Funções que expressem tarefas simples, com relativa facilidade

2.3) Linguagens de Programação mais Utilizadas em Computação Científica

Em geral, computação científica utiliza em suas implementações as linguagens de alto nível. As linguagens mais utilizadas são:

Fortran77 – Muito difundido no meio acadêmico, o Fortran77 permite declaração de *arrays* com tamanho desconhecido, dentro do escopo de uma subrotina.

Fortran90 – Uma grande evolução do Fortran77. O compilador dessa linguagem aceita os comandos do Fortran77, embora um flag (-fixed) deva ser incorporado ao comando de compilação para que ele reconheça o formato. Várias práticas obsoletas, que prejudicavam o desempenho no Fortran77, são desencorajadas. Houve uma melhoria significativa na sintaxe de *arrays*, a qual engloba um paralelismo inerente. Possui comandos para alocação dinâmica de memória, para programação orientada a objeto e permite trabalhar com ponteiros.

C – Particularmente popular entre programadores que utilizam PCs, por requerer menos memória que outras linguagens, a linguagem C foi desenvolvida para a construção de sistemas operacionais. Mesmo sendo uma linguagem de alto nível, o C tem instruções muito mais próximas do assembly que as demais linguagens de alto nível, o que permite a elaboração de códigos muito eficientes. O C é simples, portável e possui recursos para manipulação dinâmica de memória, através do uso de ponteiros, disponíveis em bibliotecas uma vez que não fazem parte do núcleo da linguagem.

C++ – Conserva as palavras chaves do núcleo da linguagem C, acrescentando outras para o uso na programação orientada a objeto. O C++ é uma das linguagens de programação mais utilizadas em aplicações gráficas e, além de possuir todos os recursos do C, inclui os recursos de alocação dinâmica no próprio núcleo da linguagem.

2.4) Elaborando um Programa

A elaboração de um programa obedece, em geral, à seguinte seqüência de passos:

- Especificar o problema

Ao escrever um programa é indispensável que o problema a ser resolvido esteja totalmente formulado e entendido, o que tornará mais simples expressar a sua solução em forma de algoritmo.

- Analisar a solução do problema e descreve-la em uma seqüência de passos (algoritmo)

Os procedimentos descritos pelo algoritmo devem conduzir-lo mais rápido e diretamente à solução do problema. Algoritmos bem elaborados gerarão programas eficientes, que realizarão determinada tarefa sem desperdiçar tempo de execução e espaço em memória com cálculos e variáveis desnecessários.

- Escrever o código

Na escolha de uma linguagem de programação deve-se considerar tanto o tipo de aplicação que se pretende desenvolver, quanto a máquina e a plataforma disponíveis para esta implementação. É interessante que o programador explore ao máximo os recursos da linguagem pela qual optar, buscando nas funções e estrutura disponíveis, soluções simples e concisas, que facilitem o entendimento e manutenção do código. Além disso, é

importante escrever o código de maneira organizada, lembrando de identar as estruturas de controle e utilizar comentários que esclareçam a sua operação.

- Compilar e executar

No último passo estão juntas as etapas de compilação e execução. Neste momento, o programador deve testar a eficiência e confiabilidade do código gerado através de um processo contínuo de teste e depuração, lembrando que é difícil um programa recém-implementado oferecer resultados corretos na primeira vez em que é compilado e executado.

Os erros em programas são geralmente classificados em dois tipos. Os erros de compilação, também chamados de erros de sintaxe, e os erros de execução, também chamados de erros de lógica. Os erros de compilação são identificados pelo compilador, que não gera um programa executável enquanto todos os erros de sintaxe não forem eliminados. Os erros de execução, mais difíceis de detectar, podem ser removidos com ajuda de recursos de depuração, que auxiliam no acompanhamento da execução, facilitando a detecção e correção dos erros de lógica.

3) Filosofias de Programação

Um programa é uma seqüência de instruções que o computador deve seguir para realizar uma determinada tarefa. Além das várias linguagens de programação, há diferentes maneiras de organizar a estrutura de um programa, isto é, a forma como as instruções serão dispostas no programa. Este critério está relacionado à filosofia utilizada na programação. Há cinco filosofias de programação, que são:

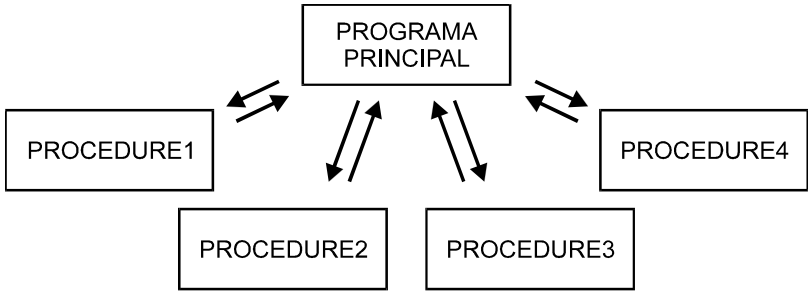
- programação não estruturada
- programação estruturada
- programação modular
- programação orientada a objeto

3.1) Programação Não Estruturada

Na programação não estruturada, uma seqüência de instruções é disposta em um único bloco , chamado programa principal, que opera diretamente todos dados. Esse tipo de técnica de programação é tremendamente desvantajoso, principalmente por gerar códigos muito longos e bem difíceis de serem lidos. Isto porque, se uma mesma seqüência de comandos é requerida em partes diferentes do programa, por exemplo, a seqüência terá que ser copiada quantas vezes for necessário.

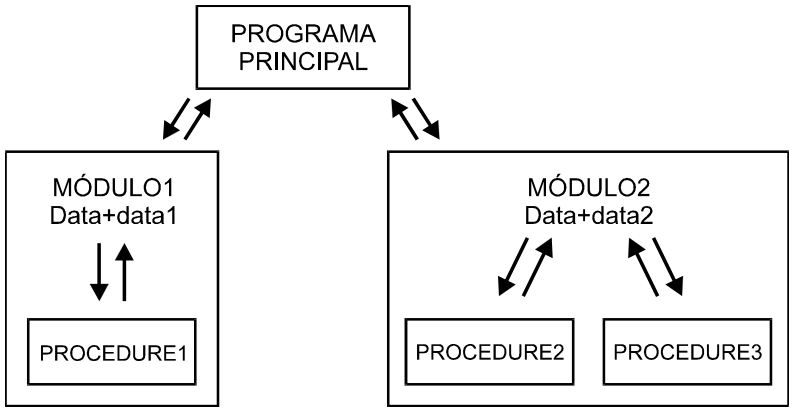
3.2) Programação Estruturada

Na programação estruturada, o programa é decomposto em subrotinas (procedure), que trabalham independentemente. Cada subrotina reúne uma seqüência de instruções que realizam determinada tarefa, como algum cálculo específico ou operação de busca, por exemplo. Sempre que necessário, o programa principal poderá acionar uma subrotina e repassar-lhe dados que precisem ser tratados; esta subrotina será executada até o final, sendo o resultado que a mesma gerar repassado ao programa principal. Após este procedimento, segue-se normalmente a execução do programa principal, até que uma nova chamada a uma subrotina seja realizada.



3.3) Programação Modular

Com a programação modular, as subrotinas com características comuns são agrupadas em módulos separados. Cada módulo pode possuir seus próprios dados, que serão manipulados através de chamadas a subrotinas que pertençam ao mesmo. A figura abaixo ilustra esse tipo de programação:



3.4) Programação Orientada para Objetos

Na programação orientada ao objeto, o programador define não apenas o tipo de dado de uma estrutura, mas também tipos de operações que podem ser atribuídas à esta estrutura. Dessa forma, cada estrutura torna-se um objeto, tal e qual no mundo real, com dados e funções característicos. Existem alguns conceitos básicos em programação orientada a objeto, sem os quais é difícil compreender o mecanismo desta filosofia de programação. Estes conceitos serão comentados nos tópicos a seguir.

4) Paralelismo

Paralelismo é uma estratégia utilizada em computação para se obter mais rapidamente a solução de problemas grandes e complexos. Segundo esta estratégia, uma tarefa grande é dividida em pequenas partes, que serão distribuídas entre vários "trabalhadores" e executadas simultaneamente pelos mesmos. Assim, na computação paralela há uma cooperação de processadores na solução de um problema.

4.1) Histórico

Os primeiros computadores realizavam dezenas de operações por segundo. Na década de noventa os computadores atingiram dezenas de bilhões de operações de ponto flutuante por segundo, sendo verificado um crescimento exponencial no desempenho de computadores desde de meados dos anos quarenta, multiplicando-se por dez a cada cinco anos.

As gerações de computadores são, em geral, divididas em cinco, cada uma correspondendo às mudanças ocorridas nos conjuntos de *hardware* que formavam os computadores de então.

Segundo Elson M. Toledo e Renato Silva, essa gerações têm as seguintes características:

- Dos relés (*relays*) e válvulas (1939-1950)
- Diodos e transistores (anos 50 e anos 60)
- Circuitos integrados de pequena e média escala (anos 60 meados 70)
- Dispositivos integrados de grande e muito grande escala (LSI e VLSI) (meados de 70 ate 1990)
- Dispositivos de ultra grande escala - ULSI – e microprocessadores poderosos

Essas sucessivas gerações aumentaram a velocidade dessas máquinas mais de um trilhão de vezes nas últimas seis décadas, reduzindo de forma significativa o seu custo. Elson M. Toledo e Renato S. Silva (LNCC) ainda afirmam que os blocos construtivos de *hardware* nas novas gerações de computadores deverão incluir dispositivos integrados em uma "giga-escala", novos materiais e estruturas na construção dos *chips* e componentes ópticos que aumentarão seu desempenho. Prevê-se também o surgimento de formas inovadoras de arquiteturas de máquinas, que resultarão em um substancial aumento na velocidade computacional.

Dentro deste contexto, a computação paralela era vista como uma sub-área da computação, interessante mas de pouca relevância. Contudo, à medida que os

computadores aumentam sua velocidade, a experiência tem mostrado que surgem sempre novas aplicações que demandam velocidades ainda maiores. Por isso, a computação paralela está se tornando cada vez mais importante nas atividades computacionais.

4.2. Conceitos Importantes

Tarefa (*Task*) – uma rotina lógica que efetua um cálculo ou desempenha determinada função.

Tarefas Paralelas (*Parallel Tasks*) – são tarefas que independem umas das outras, que podem ser executadas simultaneamente, produzindo resultados corretos.

Memória Compartilhada (*Shared Memory*) – ambiente em que vários processadores operam de maneira independente, compartilhando os recursos de uma única memória central.

Memória Distribuída (*Distributed Memory)* – ambiente em que vários processadores operam independentemente, sendo que cada um possui sua própria memória.

Message Passing – método de comunicação utilizado em processamento paralelo. Baseia-se na transmissão de dados (*send/receive*), via uma rede de interconexão, seguindo as regras de um protocolo de redes.

4.3) Máquinas Paralelas

Um computador paralelo ou máquina paralela é um conjunto de processadores capazes de trabalhar cooperativamente, na solução de um problema computacional. Esta definição é ampla o suficiente para incluir supercomputadores com centenas ou milhares de processadores, redes de estações de trabalho, estações de trabalho multiprocessadas e sistema acoplados ou interligados tipo hipercubo, etc.

4.3.1) Arquiteturas SMP

Sistemas SMP (*Symmetric Multi Processing*) possuem mais de um processador em um mesmo computador. Todos eles compartilham os recursos de memória e disco existentes, segundo uma política de controle de concorrência adotada pelo sistema operacional. Esta complexa arquitetura é bastante transparente para o usuário no desenvolvimento de aplicações, pois a maior parte da complexidade fica a cargo do sistema operacional.

4.3.2) Arquiteturas MPP

Em sistemas MPP (*Massive Parallel Processing*), os processadores possuem maior independência entre si, havendo pouco ou nenhum compartilhamento de recursos. Tipicamente, cada nó de um sistema MPP é um computador independente, com memória e disco próprios. O controle do paralelismo é feito pela aplicação, que deve coordenar a distribuição de tarefas e a coerência entre os diversos nós.

4.4) Grau de Paralelismo

Considere-se um sistema paralelo com *p* processadores. Defini-se por *grau de paralelismo* de um algoritmo numérico o número de operações deste algoritmo que podem ser realizadas em paralelo. Considere-se como exemplo a soma de dois vetores *"a"* e *"b"* de mesma dimensão *"n"*, realizada segundo a seqüência de passos a seguir:

```
Repita para i=1 até n
        C(i) = a(i) +b(i)
Fim do repita
```

Como facilmente pode-se constatar, algoritmo acima é composto de *n* operações (no caso adições) a serem efetuadas, que são independentes entre si e, dessa forma, podem ser realizadas simultaneamente. Concluí-se, portanto, que o grau de paralelismo deste algoritmo é o próprio *"n".*

Nesta definição constatamos que o grau de paralelismo de um algoritmo é uma medida intrínseca de seu paralelismo, sendo independente do número de processadores disponíveis para a resolução do problema. É claro, entretanto, que o número de processadores afeta o tempo de execução. Por exemplo, se estão disponíveis cem processadores *(p=100)* para resolver um problema de grau de paralelismo cem *(n=100)*, a computação pode ser realizada em 1 passo de tempo. Se *(p=10)* e *(n = 100)*, a computação seria realizada em 10 passos de tempo.

Considere-se agora, o problema de somar *"n"* números *a1,a2,a3, ..., an* utilizando o seguinte algoritmo seqüencial:

```
S=0
Repita para i=1 até n
S=S+ai
Fim do repita
```

Neste caso, como a computação se dá em vários passos, utiliza-se como medida do grau de paralelismo a soma do grau de paralelismo dos vários passos, dividido pelo número de passos; em outras palavras, a soma do número de operações que podem ser realizadas em paralelo, em cada passo, dividida pelo número de passos. Assim, neste exemplo, temos "*n*" passos, cada um com uma única operação, resultando num grau de paralelismo unitário.

Neste último exemplo, ocorre o contrário do que ocorreu na soma de dois vetores. Este algoritmo independe do número de processadores utilizados. Diz-se que o algoritmo de soma de dois vetores mostrado é 100% paralelizável, enquanto o algoritmo

acima descrito para a soma de *n* números é 100% seqüencial. Entre esses dois extremos encontramos algoritmos que possuem taxas diferentes de paralelização, sendo desta forma alguns algoritmos mais paralelizáveis que outros.

4.5) Desempenho

Na computação seqüencial, um algoritmo é bem caracterizado em termos do trabalho que realiza, podendo ser avaliado pela contagem das operações envolvidas e da quantidade de memória necessária. Neste contexto, o desempenho de um computador é melhor do que de outro quando a mesma aplicação é executada mais rapidamente no primeiro que no segundo. Uma medida muito utilizada para medir o desempenho de um computador é o MFlops (MFlop/s), que define o desempenho através da taxa de milhões de operações de ponto flutuante por segundo que este é capaz de realizar. Também é comum ser fornecido pelo fabricante o "*peak performance*" que é o máximo valor de MFlops que a máquina pode teoricamente obter.

Com relação às máquinas paralelas, a dificuldade está na definição de medidas adequadas para avaliação do desempenho deste tipo de processamento. Medidas adequadas são essenciais para o estabelecimento de diretrizes no projeto de algoritmos, de novas arquiteturas, na identificação de "*gargalos*" que prejudicam a paralelização de algoritmo e até mesmo na escolha de uma arquitetura mais apropriada ao problema a ser resolvido.

Ainda que a velocidade de processamento e as exigências de memória continuem sendo fatores essenciais, deve-se também levar em conta outros dois fatores que afetam significativamente o desempenho neste tipo de computação: gastos de tempo para comunicação e perdas de tempo devido à sincronização. A influência destes fatores pode reduzir sensivelmente a eficiência de um algoritmo paralelo, resultando em tempos de processamento distantes do razoável quando aumentamos o número de processadores e as dimensões do problema a ser resolvido.

Dentre as principais medidas que buscam avaliar o desempenho da computação paralela, estão: o ganho (ou *speedup*) (*S*) e a eficiência (*E*). O ganho é um fator que compara o tempo total consumido por um algoritmo quando executado em uma máquina seqüencial, em relação ao tempo requerido pelo mesmo algoritmo quando executado em uma máquina paralela com *P* processadores. A expressão para o ganho é definida como:

$$Sp = Tseq/Tp$$

Onde:

Tseq – tempo consumido por uma máquina seqüencial

Tp – tempo consumido por uma máquina paralela

O fator ganho nos dá uma medida de como uma aplicação paralela é executada em comparação a um programa seqüencial equivalente. Teoricamente ele deve ser sempre menor ou igual ao número de processadores *P*. Entretanto, dependendo de como o tempo seqüencial seja medido, o valor máximo de *Sp* pode variar, pois o processador usado para medir *Tseq* pode diferir muito em velocidade dos processadores usados na máquina paralela, podendo mascarar a eficiência do algoritmo paralelo.

Também a especificação do algoritmo usado é também muito importante, já que algoritmos distintos para um mesmo problema apresentam tempo de processamento diferentes. Ou ainda, a versão paralela de um algoritmo pode não ser a melhor opção quando executada em um único processador. Uma alternativa é definir um algoritmo como sendo seqüencial ótimo para um problema particular e usá-lo para medir o tempo *Tseq* na

expressão do ganho. A definição deste algoritmo ótimo, no entanto, é muito vaga.

Da definição de ganho, vemos que seu valor ideal é igual ao número de processadores ($Sp = P$). Isto leva ao conceito de eficiência de um sistema paralelo como:

$$E = Sp/P$$

A eficiência (E) dá uma indicação da porcentagem do tempo total realmente despendido na aplicação por cada um dos processadores.

4.6) Implementando o Paralelismo

4.6.1) Elaborando um Algoritmo Paralelo

Da própria definição de paralelismo, pode-se supor que a elaboração de um algoritmo paralelo deve partir da decomposição de um algoritmo em pequenas tarefas independentes, que possam ser executadas simultaneamente. Para realizar-se esta decomposição, é preciso que se tenha em mente a idéia de decomposição funcional e decomposição de domínio.

Na decomposição funcional, o problema é decomposto em diferentes tarefas, distribuídas entre processadores que as executarão simultaneamente. Neste tipo de decomposição, perfeito para um programa dinâmico e modular, cada tarefa será um programa diferente. Na decomposição de domínio, os dados são decompostos em grupos e distribuídos entre processadores, que os utilizarão na execução de um mesmo programa. Este tipo de decomposição é mais indicado para processamentos que utilizem dados estáticos, como resoluções de matrizes de alta ordem.

Como cada método de decomposição é mais indicado a um determinado tipo de aplicação, é preciso que se tenha em mente como é que a decomposição de um algoritmo serial deve contribuir para a redução da complexidade e/ou do volume de cálculos realizados por cada processador, em relação ao que ocorreria na realização do mesmo processamento serialmente. Neste contexto, é importante também evitar a construção de tarefas que levem mais tempo para serem coordenadas que executadas.

4.6.2) Exemplos de Bibliotecas para Implementação de Paralelismo

Alguns exemplos de bibliotecas que implementam o paralelismo são o PVM, P4, Linda, e MPI, as quais podem ser chamadas a partir de qualquer linguagem. Para utiliza-las, o programador deve paralelizar explicitamente o seu código, convivendo, portanto, com problemas de sincronização. As rotinas de uma destas bibliotecas devem ser chamadas por um programa escrito em uma linguagem de programação (C, Fortran, etc...), para a criação e coordenação de tarefas em paralelo.

Alguns softwares e bibliotecas implementam paralelismo de forma transparente ao usuário. Sua utilização não requer plenos conhecimentos acerca de programação paralela, uma vez que o paralelismo nas mesmas é implementado através de rotinas que as acompanham. Enquadram-se nesta classe as bibliotecas PESSL, OSLP, e os softwares Gaussian, Gamess e Dalton.

5) Message Passing

"Message-Passing" é um dos vários modelos computacionais para conceituação de operações de programa. O modelo "Message-Passing" é definido como:

* Conjunto de processos que possuem acesso à memória local;
* Comunicação dos processos baseados no envio e recebimento de mensagens;
* A transferência de dados entre processos requer operações de cooperação entre cada processo (uma operação de envio deve "casar" com uma operação de recebimento).

5.1) BIBLIOTECAS "MESSAGE-PASSING"

O conjunto operações de comunicação, formam a base que permite a implementação de uma biblioteca de "Message-Passing":

Domínio público - PICL, PVM, PARMACS, P4, MPICH, etc;

Privativas - MPL, NX, CMMD, MPI, etc;

Existem componentes comuns a todas as bibliotecas de "Message-Passing", que incluem:

* Rotinas de gerência de processos (inicializar, finalizar, determinar o número de processos, identificar processos);
* Rotinas de comunicação "Point-to-Point" (Enviar e receber mensagens entre dois processos);
* Rotinas de comunicação de grupos ("broadcast", sincronizar processos).

5.1.1) Terminologia de Comunicação

Buffering Cópia temporária de mensagens entre endereços de memória efetuada pelo sistema como parte de seu protocolo de transmissão. A cópia ocorre entre o "buffer" do usuário (definido pelo processo) e o "buffer" do sistema (definido pela biblioteca);

Blocking Uma rotina de comunicação é "blocking", quando a finalização da execução da rotina, é dependente de certos "eventos" (espera por determinada ação, antes de liberar a continuação do processamento);

Non-blocking Uma rotina de comunicação é "non-blocking", quando a finalização da execução da rotina, não depende de certos "eventos" (não há espera, o processo continua sendo executado normalmente);

Síncrono Comunicação na qual o processo que envia a mensagem, não retorna a execução normal, enquanto não haja um sinal do recebimento da mensagem pelo destinatário;

Assíncrono Comunicação na qual o processo que envia a mensagem, não espera que haja um sinal de recebimento da mensagem pelo destinatário.

5.1.2) Comunicação "Point-to-Point"

Os componentes básicos de qualquer biblioteca de "Message-Passing" são as rotinas de comunicação "Point-to-Point" (transferência de dados entre dois processos).

Bloking Send Finaliza, quando o "buffer" de envio está pronto para ser reutilizado;

Receive Finaliza, quando o "buffer" de recebimento está pronto para ser reutilizado;

Nonblocking Retorna imediatamente, após envio ou recebimento de uma mensagem.

5.1.3) Comunicação Coletiva

As rotinas de comunicação coletivas são voltadas para coordenar grupos de processos.

Existem, basicamente, três tipos de rotinas de comunicação coletiva:

* Sincronização
* Envio de dados: Broadcast, Scatter/Gather, All to All
* Computação Coletiva: Min, Max, Add, Multiply, etc

5.1.4) "Overhead"

Existem duas fontes de "overhead" em bibliotecas de "message-passing":

"System Overhead" É o trabalho efetuado pelo sistema para transferir um dado para seu processo de destino;

Ex.: Cópia de dados do "buffer" para a rede.

"Syncronization Overhead" É o tempo gasto na espera de que um evento ocorra em um outro processo;

Ex.: Espera, pelo processo origem, do sinal de OK pelo processo destino.

6) Programação MPI

6.1) O que é MPI ?

* Message Passing Interface
* Uma biblioteca de "Massage-Passing", desenvolvida para ser padrão em ambientes de memória distribuída, em "Message-Passing" e em computação paralela.
* "Message-Passing" portável para qualquer arquitetura, tem aproximadamente 125 funções para programação e ferramentas para se analisar a performance.
* Utilizado por programas em C e FORTRAN.
* A plataforma alvo para o MPI são ambientes de memória distribuída, máquinas paralelas massivas, "clusters" de estações de trabalho.
* Todo paralelismo é explícito: o programador é responsável em identificar o paralelismo e implementar um algoritmo utilizando construções com o MPI.

6.2) HISTÓRICO

Fins da década de 80 Memória distribuída, o desenvolvimento da computação paralela, ferramentas para desenvolver programas em ambientes paralelos, problemas com portabilidade, performance, funcionalidade e preço, determinaram a necessidade de se desenvolver um padrão.

Abril de 1992 "Workshop" de padrões de "Message-Passing" em ambientes de memória distribuída (Centro de Pesquisa em Computação Paralela, Williamsburg, Virginia);

Discussão das necessidades básicas e essenciais para se estabelecer um padrão "Message-Passing";

Criado um grupo de trabalho para dar continuidade ao processo de padronização.

Novembro de 1992 Reunião em Minneapolis do grupo de trabalho e apresentação de um primeiro esboço de interface "Message-Passing" (MPI1). O Grupo adota procedimentos para a criação de um MPI Forum;

MPIF consiste eventualmente de aproximadamente 175 pessoas de 40 organizações, incluindo fabricantes de computadores, empresas de softwares, universidades e cientistas de aplicação.

Novembro de 1993 Conferência de Supercomputação 93 - Apresentação do esboço do padrão MPI.

Maio de 1994 Disponibilização, como domínio público, da versão padrão do MPI (MPI1 http://www.mcs.anl.gov/Projects/mpi/standard.html

Dezembro de 1995 Conferência de Supercomputação 95 - Reunião para discussão do MPI2 e suas extensões.

6.3) Implementações de MPI

MPI-F: IBM Research

MPICH: ANL/MSU - Domínio Publico

UNIFY: Mississipi State University

CHIMP: Edinburgh Parallel Computing Center

LAM: Ohio Supercomputer Center

6.4) CONCEITOS E DEFINIÇÕES

Rank Todo processo tem uma única identificação, atribuída pelo sistema quando o processo é iniciado. Essa identificação é continua e começa no zero até n-1 processos.

Group Grupo é um conjunto ordenado de N processos. Todo e qualquer grupo é associado a um "communicator" e, inicialmente, todos os processos são menbros de um grupo com um "communicator" já pré-estabelecido (MPI_COMM_WORLD).

Communicator O "communicator" define uma coleção de processos (grupo), que poderão se comunicar entre si (contexto). O MPI utiliza essa combinação de grupo e contexto para garantir uma comunicação segura e evitar problemas no envio de mensagens entre os processos.

Aplication Buffer É um endereço normal de memória (Ex: variável) aonde se armazena um dado que o processo necessita enviar ou receber.

System Buffer É um endereço de memória reservado pelo sistema para armazenar mensagens. Dependendo do tipo de operação de send/receive, o dado no "aplication buffer" pode necessitar ser copiado de/para o "system buffer" ("Send Buffer" e "Receive Buffer"). Neste caso teremos comunicação assíncrona.

Blocking Comunication Uma rotina de comunicação é dita "bloking", se a finalização da chamada depender de certos eventos.

Ex: Numa rotina de envio, o dado tem que ter sido enviado com sucesso, ou, ter sido salvo no "system buffer", indicando que o endereço do "aplication buffer" pode ser reutilizado.

NonBlocking Comunic. Uma rotina de comunicação é dita "Non-blocking" , se a chamada retorna sem esperar qualquer evento que indique o fim ou o sucesso da rotina.

Ex: Não espera pela cópia de mensagens do "aplication buffer" para o "system buffer", ou a indicação do recebimento de uma mensagem.

OBS: É da responsabilidade do programador, a certeza de que o "aplication buffer" esteja disponível para ser reutilizado. Este tipo de comunicação é utilizado

para melhorar a relação entre computação e comunicação para efeitos de ganho de performance.

6.5) COMPILAÇÃO

A implementação do MPI definiu num único comando, as tarefas de compilação e linkedição:

FORTRAN 90

mpxlf <fonte> -o <executável>

C Standard

mpcc <fonte> -o <executável>

C++

mpCC <fonte> -o <executável>

OBS: É possível utilizar todas as opções de compilação dos compiladores C e FORTRAN.

6.6) EXECUÇÃO

% poe <arquivo executável>

A execução de um programa, com rotinas MPI, no ambiente IBM/SP/AIX, é feita através de uma interface de operação que configura o ambiente paralelo, ou seja, configurar o POE.

O Parallel Operation Environment possui variáveis de ambiente que irão determinar o modo de execução de um programa paralelo. Essas variáveis podem ser definidas no momento da execução do programa, configuradas uma a uma antes da execução ou, definidas no profile de configuração da área do usuário (.cshrc).

Opções do POE

MP_PROCS=n	Especifica o número (n) de processos que serão inicializados. (-procs);
MP_HOSTFILE=	Especifica um arquivo com o nome das máquinas que poderão participar do processamento. (-hostfile);
MP_EUIDEVICE=	Especifica o tipo de adaptador (css0, fi0, en0, tr0) que deve ser utilizado para comunicação IP entre os nós (-euidevice);

MP_EUILIB= Especifica qual será o tipo de protocolo de comunicação (ip/us) utilizado entre os processos. (-eulib);

MP_ADAPTER_USE= Especifica o modo de uso (dedicated/shared) do adapator de comunicação. (-adapter_use);

MP_CPU_USE= Especifica o modo de uso (unique/multiple) da cpu. (-cpu_use);

6.7) ROTINAS BÁSICAS

Para um grande número de aplicações, um conjunto de apenas 6 subrotinas MPI serão suficientes para desenvolver uma aplicação no MPI.

Arquivo "Include"

Necessário para todos os programas ou rotinas que efetuam chamadas para a biblioteca MPI. Normalmente é colocado no início do programa.

C #include "mpi.h"

FORTRAN include "mpif.h"

6.7.1) Inicializar um processo MPI

MPI_INIT

-Primeira rotina MPI utilizada.

-Sincroniza todos os processos no inicio de uma aplicação MPI.

C int MPI_Init (*argc,*argv)

FORTRAN call MPI_INIT (mpierr)

argc Apontador para um parâmetro da função main;

argv Apontador para um parâmetro da função main;

mpierr Variável inteira de retorno com o status da rotina.

6.7.2) Identificar processo do MPI

MPI_COMM_RANK

-Identifica o processo, dentro de um grupo de processos.

-Valor inteiro, entre 0 e n-1 processos.

C int MPI_Comm_rank (comm, *rank)

FORTRAN call MPI_COMM_RANK (comm,rank,mpierr)

comm MPI communicator.

rank Identificação do processo.

mpierr Status da rotina.

6.7.3) Contar processos no MPI

MPI_COMM_SIZE

-Retorna o número de processos dentro de um grupo de processos.

C	int MPI_Comm_size (comm, *size)
FORTRAN	call MPI_COMM_SIZE (comm,size,mpierr)

comm	MPI Communicator.
size	Número de processos inicializados.
mpierr	Status da rotina.

6.7.4) Enviar menssagens no MPI

MPI_SEND

-"Blocking send".

-A rotina só retorna após o dado ter sido enviado.

C	int MPI_Send(*sbuf,n,type,dest,tag,comm)
FORTRAN	call MPI_SEND(sbuf,n,type,dest,tag,comm, mpierr)

sbuf	Endereço inicial do dado que será enviado.
n	Número de elementos a serem enviados.
type	Tipo do dado.
dest	Identificação do processo destino.
tag	Rótulo da mensagem.
comm	MPI communicator.
mpierr	Variável inteira de retorno com o status da rotina.

6.7.5) Receber mensagens no MPI

MPI_RECV

-"Blocking receive".

-A rotina retorna após o dado ter sido recebido e armazenado.

C	int MPI_Recv(*rbuf,n,type,src,tag,*st,comm)
FORTRAN	call MPI_RECV(rbuf,n,type,src,tag,comm,st,mpierr)

rbuf	Variável indicando o endereço do "aplication buffer".
n	Número de elementos a serem recebidos.
type	Tipo do dado.
src	Identificação da fonte. OBS: MPI_ANY_SOURCE
tag	Rótulo da mensagem. OBS: MPI_ANY_TAG
comm	MPI communicator.
st	Vetor com informações de source e tag.
mpierr	Variável inteira de retorno com o status da rotina.

6.7.6) Finalizar processos no MPI

MPI_FINALIZE

-Finaliza o processo para o MPI.

-Última rotina MPI a ser executada por uma aplicação MPI.

-Sincroniza os processos na finalização de uma aplicação MPI.

C	int MPI_Finalize()
FORTRAN	call MPI_FINALIZE (mpierr)

mpierr Variável inteira de retorno com o status da rotina.

7) Exemplo de um Programa Básico MPI

```
program hello
include 'mpif.h'
integer me, nt, mpierr, tag, status(MPI_STATUS_SIZE)
character(12) message
call MPI_INIT(mpierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nt, mpierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, me, mpierr)
tag = 100
if(me .eq. 0) then
  message = 'Hello, world'
  do i=1, nt-1
    call MPI_SEND(mss,12,MPI_CHARACTER,i,tag,
                  MPI_COMM_WORLD, mpierr)
  enddo
else
    call MPI_RECV(message,12,MPI_CHARACTER, 0, tag,
                  MPI_COMM_WORLD,status, mpierr)
endif
print*, 'node', me, ':', message
call MPI_FINALIZE(mpierr)
end
```

Referências

1. Toledo, Elson M. e Silva, Renato S., *Introdução à Computação Paralela,*LNCC, 1997
2. Müller, Peter, *Introduction to Object-Oriented Programming Using C++*, GNA,1997,

http://www.gnacademy.org/

3. GNU Make,

http://ww.dina.kvl.dk/DinaUnix/Info/make

4. AC Marshall, *Fortran 90 Course Notes,* The University of Liverpool ,1997,

http://www.liv.ac.uk/HPC/F90page.html

5. Computanional Science Educational Project , *Fortran 90 and Computanional Science, 1995*

http://csep1.phy.ornl.gov/CSEP/PL/PL.html

6. Taylor, David A., *Object-Oriented Technology*, Addison-Wesley Publishing Company, Inc, 1992
7. Computanional Science Educational Project , *Overview of Computanional Science, 1995*

http://csep1.phy.ornl.gov/CSEP/OV/OV.html

8. Küsel, Ricardo, *Introdução ao PVM,* CENAPAD – SP

Anotações

OTIMIZAÇÃO

Conteúdo do módulo

Parte I: Análise de desempenho

- conceitos e estratégias
- Ferramentas de análise de desempenho serial
- perfiladores

Parte II: Otimização

- Otimização por compilador
- Otimização Manual
  - ☞ Bibliotecas de Subrotinas Matemáticas
    - ❑ NAG
    - ❑ ESSL

Introdução

Objetivos do Módulo

Apresentação de conceitos e ferramentas para a análise e otimização de programas seqüenciais.

Otimizar:

- melhorar desempenho significa diminuir tempo de processamento: economia de recursos.

Otimização seqüencial:

- constitui o nível básico de otimização, imprescindível inclusive para a programação paralela eficiente, já que não faz sentido um código paralelo com pobre desempenho serial, embora o desempenho paralelo deva ir muito além de sua eficiência seqüencial.

Parte I : Análise de Desempenho

1. Introdução

Constitui o marco zero na otimização, a determinação dos pontos críticos, que são regiões do código em que ocorre um afunilamento do fluxo de dados ou instruções, isto é, partes do programa que consomem a maior parte do tempo de processamento.

Técnicas de análise ou perfilamento de programas:

- cronometragem: medições de tempos de execução
- perfis: levantamentos estatísticos de execução

2. Perfiladores e Cronômetros

Há diversos instrumentos de perfilamento e cronometragem, desde ferramentas incluídas no sistema operacional até rotinas em Fortran ou C.

Comandos

- **time**
- **prof**
- **tprof**
- **timex**
- **gprof**

Rotinas

- em C: **gettimeofday**
- do XL Fortran: **rtc**, **irtc**, **dtime**_, **etime**_, **mclock**, **timef**

3. Comandos

3.1. Time

O comando **time** retorna os tempos totais de execução de um programa.

Sintaxe:

`time` <*programa_executável*>

Retorna no formato abaixo (ksh):

real	0m27.11s	(tempo real)
user	0m12.02s	(tempo total de CPU do programa)
sys	0m0.17s	(tempo de CPU do sistema operacional para a execução do programa)

3.2. time e timex

time retorna em formato diferente em csh.

timex tem mesma sintaxe que **time** e retorna em um único formato (como **time** em ksh).

Opção **-s** em **timex** retorna várias outras informações.

Mais informações com o comando **man** do Unix

3.3. prof

prof é uma ferramenta encontrada na maioria dos sistemas Unix. Possibilita o perfilamento da execução de programas ao nível de subprogramas.

Retorna, para cada subprograma (incluindo o principal):

- nome (em ordem decrescente da utilização de CPU)
- porcentagem de utilização de CPU
- tempo de execução do subprograma (em segundos)
- tempo acumulado (na ordem da lista)
- número de chamadas
- tempo médio (milisegundos) para uma chamada

Modo de utilização:

- compilar com opção **-p**
- executar o programa (arquivo **mon.out** é gerado automaticamente)
- digitar **prof** na linha de comando (utiliza **mon.out** automaticamente (padrão) para gerar as informações)

Maiores informações podem ser obtidas com o comando **man**. Programas paralelos também podem ser perfilados com **prof**.

3.4. gprof

Também presente na maioria dos sistemas Unix, fornece as mesmas informações que **prof**, além das seguintes informações, fornecidas a cada subprograma:

- Identifica que subprogramas o invocam ('pais')
- Atribui um número como índice do subprograma
- Porcentagem de CPU, incluindo o tempo gasto pelos descendentes; **útil ao chamar bibliotecas de rotinas**
- Tempo gasto pelo subprograma e seus descendentes (dados separadamente)
- Número de vezes que foi chamado
- Seus descendentes diretos, seus tempos e de seus descendentes; n° de chamadas dos 'filhos' pelos 'pais'

Utilização:

- compilar com opção -pg
- executar (gera gmon.out)
- **gprof** (na linha de comando)

Obs.: 1) **gprof** e **prof** só fornecem tempo de CPU. Entrada e saída não são considerados (tempo real).

2) Subprogramas acrescentados nas listas produzidas por **gprof** e **prof** podem ser ignorados. Maiores informações nas páginas man.

3) **gprof** e **prof** podem ser utilizados em programas paralelos.

3.5. tprof

tprof proporciona as mesmas informações que gprof ou prof (exceto informações sobre descendentes e ascendentes, obtidas com **gprof**), e ainda:

- tempo de CPU referente a cada linha do programa (desde que tenha sido compilado com a opção **-g** - a mesma necessária para o depurador xldb)
- informações sobre todos processos simultâneos ao programa em análise
- tempo de latência

Modo de utilização:

- Compilar com as opções **–g**, –**qlist** e **-qsource**, se informações ao nível das linhas do programa forem

almejadas. Não é necessário usar nenhuma opção para obter informações no nível de subprogramas.

- na linha de comando, executar o programa:

  tprof *<nome_do_executavel>*

- vários relatórios são gerados, em arquivos separados. Entre os mais importantes, tem-se

  - número das linhas de maior uso de CPU, em ordem decrescente:

    __h.*<nome_do_progr_fonte>*

  - resumo das informações:

    __*<nome_do_prog_fonte>*.all

  - perfil de um subprograma:

    __*<nome_do_subprograma><nome_do_prog_fonte>*

3.6. Como são Feitas as Estimativas

Os perfiladores **prof**, **gprof** e **tprof** fazem um levantamento dos dados, interrompendo a execução do programa uma vez a cada centésimo de segundo e anotando o ponto onde isso ocorreu. Cada interrupção é denominada um *tick*. Os dados representam estimativas realizadas dessa maneira.

No caso de **tprof**, é possível associar o número de *ticks* anotado, para cada linha de programa, durante a execução. Como será visto adiante, a utilização de opções de otimização de compilador altera o código e, por isso, não constitui uma base adequada para a associação de *ticks* às linhas do código fonte.

4. Rotinas

gettimeofday é uma rotina padrão em C na maior parte dos sistemas Unix. Retorna o tempo em segundos e microsegundos a partir de sua data-referência. Pode ser invocada também a partir de um programa em Fortran.

As rotinas XL Fortran (rtc, irtc, dtime_, etime_, mclock, timef) fornecem várias alternativas de cronometragem. Por exemplo:

- **dtime_** fornece o intervalo de tempo de CPU (*user* e *system*) desde a última chamada da rotina (ver exemplo a seguir).

```
PROGRAM   DTIME_TIME
      REAL(4)   DELTA,  dtime_
      TYPE   TB_TYPE
       SEQUENCE
       REAL(4)   USRTIME
       REAL(4)   SYSTIME
      END TYPE
      TYPE (TB_TYPE)   DTIME_STRUCT
       DELTA = dtime_(DTIME_STRUCT)
       DO   M = 1,2000000
            N = N + M
       END DO
       DELTA = dtime_(DTIME_STRUCT)
       PRINT *, 'User time: ',DTIME_STRUCT%USRTIME, 'seconds'
       PRINT *,'System time: ',DTIME_STRUCT%SYSTIME,
     'seconds'
       PRINT *, 'Elapsed time: ',DELTA, 'seconds'
END
```

Otimização

1. Introdução

Redução do tempo de processamento nos pontos críticos, mediante uma melhor utilização dos recursos de processamento.

Exceto em aplicações comerciais ou outras em que o desempenho do código seja imperativo, existem recursos capazes de realizar automaticamente a maior parte da otimização de um código.

A utilização desses recursos pode ser feita a partir de uma participação muito pequena - porém judiciosa - do programador. Em geral, otimizações implementadas à mão são mais eficientes. Por isso é necessário considerar a otimização à mão, nos casos em que a importância do desempenho sobrepõe-se a todo esforço de desenvolvimento.

Algum conhecimento das técnicas de otimização é pré-requisito para uma boa utilização das ferramentas de otimização disponíveis.

Parte II : Otimização Manual

2. Atalhos para a Otimização

Antes partir para a otimização de um código à mão, é preciso ter certeza de ser esse o melhor caminho.

Quatro princípios básicos a considerar:

- eficiência do algoritmo;
- utilização de bibliotecas de rotinas;
- uso de opções de compilação;
- utilização de pré-processadores.

2.1. Algoritmo

A primeira condição para um código eficiente é a utilização de um algoritmo adequado. Não faz sentido otimizar um código que implementa um algoritmo ruim, ineficiente.

Em geral, o desenvolvimento de um algoritmo está além do interesse do programador. Mas ele deve ao menos escolher entre os melhores algoritmos disponíveis para o objetivo almejado.

2.2. Bibliotecas de Rotinas

Constituem normalmente o caminho mais fácil, rápido e seguro para um código eficiente, pois:

- estão prontas para serem usadas;
- são construídas a partir dos algoritmos mais eficientes que se conhecem;
- seus códigos são extremamente otimizados;
- estão suficientemente bem testadas.

Obs.: Para aplicações científicas costumam estar disponíveis em Fortran (f77, f90) e C (C, C++).

2.3. Opções de Compilação

As opções oferecidas pelos compiladores constituem uma das formas de se automatizar a otimização.

Na otimização realizada pelos compiladores é a eficiência não é garantida de maneira uniforme para qualquer rotina. Variações e combinações devem ser testadas.

As opções são particulares aos compiladores.

2.4. Pré-processadores

Pré-processadores procuram alterar os códigos fonte e transformá-los em algoritmos capazes de explorar ao máximo as capacidades de otimização dos compiladores.

São capazes de gerar códigos quase tão eficientes quanto os que podem ser desenvolvidos manualmente, bem como requerem menor tempo em seu desenvolvimento e estão menos sujeitos a erros que a otimização manual.

Obs.: nenhum instalado no ambiente CENAPAD-SP, atualmente.

3. Arquitetura e Otimização

Como otimização significa utilização eficiente dos recursos disponíveis, trata-se de uma adaptação, de um ajuste do algoritmo às características da máquina que o executa.

Por isso, antes de se considerar as técnicas de otimização, é conveniente recapitular um pouco da hierarquia de memória, com especial atenção ao consumo de tempo característico da transferência de memória de cada camada dessa hierarquia.

4. Hierarquia de Memória

CPU:

- Onde as instruções são executadas.

Quanto maior a velocidade de acesso de uma área de memória à CPU, menor suas dimensões (dado que o custo da memória aumenta com sua velocidade) .

Registros:

- acesso imediato à CPU (0 ciclos).

Cache:

- acesso muito rápido, dimensões pequenas:

  1 ciclo; da ordem de 100 KB.

Memória principal:

- acesso mais lento, dimensões maiores:

  cache miss: 8-12 ciclos,

  TLB *miss:* 36-56ciclos;

  tamanho: da ordem de 100 MB.

Discos:

- podem ter área de armazenamento muito maior, mas o acesso é muito lento:

  page fault: da ordem de 10^5 ciclos

Fitas:

- armazenamentos ainda maiores, acesso muito mais lento.

4.1. 'Cache'

Um *cache* é tipicamente dividido em linhas (64-256 bytes), que constituem as unidades de transferência entre ele e a memória principal.

O *cache* é implementado por meio de classes de congruência. Isto significa que o acesso de cada endereço real de memória é restrito a um número de linhas do *cache*, que é igual ao de classes de congruência implementadas.

4.1.1. Falhas de 'cache' ('cache' e TLB 'misses')

Quando um endereço referenciado não está representado no *cache,* ocorre uma falha de *cache* (8-12 ciclos). O endereço precisa ser transferido de uma página da memória principal. O endereço dessa página deve estar na lista de um *buffer*, que é uma área de transferência denominada TLB. Caso a página com o endereço referenciado não esteja nesse *buffer*, ocorre um novo

atraso (falha de TLB: 36-56 ciclos), para que o número da página referida da memória principal seja incluído na lista da TLB.

4.2. Memória Principal

A memória principal é dividida em páginas (de 4096 bytes na arquitetura RS/6000)

Se um endereço referenciado não pertence a uma página contida na TLB (isto é, se ocorre a falha de TLB), a página tem de ser procurada numa tabela maior, a fim de determinar o número da página da memória principal que contém o endereço referenciado (custo de 36-56 ciclos)

Se, entretanto, o endereço não pertence à memória principal, ocorre uma falha de página (*page fault)*, com custo da ordem de 10^5 ciclos. A operação é conduzida pelo sistema operacional, envolvendo entrada/saída, ocasionando um processo de transferência muito mais lento.

4.2.1. *Uma Boa Utilização de Memória*

Significa minimizar as transferências, utilizando a forma mais intensa possível, que é a parte mais alta da hierarquia, processando ao máximo os dados antes que ocorram as falhas de localização.

A transferência de uma linha de *cache* de cada vez favorece que as operações temporalmente próximas explorem os endereços adjacentes.

Normalmente o fluxo de dados é crítico. Raramente o volume de instruções é o fator crítico do uso de memória.

4.3. Um Exemplo dos Custos de Transferência

Considere-se o processamento repetido, através de um laço de um vetor de números reais de dupla precisão, de 150000 elementos. Considere-se que apenas este vetor está sendo processado.

Admita-se um *cache* com linhas de 128 bytes.

- Tamanho do vetor: 150000*8 = 1,2 MB

Como o vetor não cabe no *cache*, seu acesso durante as iterações tem de ser feito a cada 16 elementos:

- 128/8 = 16 (16 elementos cada linha de *cache*)

Haverá 32 falhas de *cache* por página de memória:

- bytes_por_pág / bytes_por_linha = 4096/128 = 32

Haverá 1 falha de TLB a cada página processada, ou 293 falhas para o vetor todo:

- 1,2 MB / 4096 bytes = 293 páginas para o vetor

Admitindo-se um custo por falha de *cache* e de TLB de 8 ciclos e de 36 ciclos, respectivamente, o custo total por iteração será de 85556 ciclos:

- [32*8 + 1*36] * 293 pags = 85556 ciclos

Compare-se este custo com o de paginação (cerca de 10^5). Deve-se procurar processar todo conteúdo da memória real antes de acessar novos dados em disco.

4.4. Unidades de Processamento

Um processador RS/6000 possui 3 unidades

- BPU (*branch processing unit*): processamento de instruções, condições e ramificações
- FXU (*fixed-point unit*): aritmética de inteiros, operações lógicas e com caracteres, endereçamento e operações de comparação
- FPU (floating-point unit): aritmética de ponto flutuante e operações de comparação

Em termos de desempenho, os pontos críticos situam-se nas unidades normalmente nas operações aritméticas.

4.5.Custo das operações

Embora os valores exatos variem conforme a implementação RS/6000, é importante destacar alguns aspectos gerais acerca da aritmética de **ponto flutuante**:

Adição, subtração e multiplicação possuem o mesmo custo em ciclos encadeados numa iteração, por exemplo, uma multiplicação seguida de uma adição (ou subtração) possui o

mesmo custo de uma adição isolada (FMA: *floating-point-multiply-add*)

A divisão possui um custo muito maior que a adição (13-19 vezes)

4.6.Um Efeito Secundário

A arquitetura **power** permite o cálculo de uma soma e um produto num único ciclo (FMA). No entanto, certas condições pode impedir que isso ocorra. Por exemplo quando o compilador conservadoramente presume que a(j) pode ser modificado pela atribuição a a(i), o que não ocorre neste caso. O uso de uma variável temporária resolve este problema.

```
j = n+1
do  i = 1, n
  a(i) = a(j)*a(i) + a(i)
end do
```

```
j = n+1
aux = a(j)
do  i = 1, n
  a(i) = aux*a(i) + a(i)
end do
```

5.Técnicas de Otimização

- Técnicas: otimizações
  - aritméticas (destaque: bibliotecas de subrotinas)
  - de conjuntos ordenados (*arrays*)
  - de laços
  - de construções de controle
  - de entrada/saída
  - do tamanho do código executável
- Utilização de opções de compilação
- Bibliotecas de sub-rotinas
- Uso de pré-processadores

Resumo das Principais Técnicas

5.1.Otimizações Aritméticas

A melhor maneira de processar operações aritméticas de modo eficiente é através das bibliotecas de rotinas matemáticas. Outras observações de maior interesse talvez sejam:

- **Conversão de tipos**: tem um custo relativamente elevado. É importante evitar conversões implícitas, como presença de um inteiro numa expressão de ponto flutuante.

- **Divisão**: é bem mais lenta que a multiplicação pelo recíproco.

- **Potenciação**: tende a ser mais lenta que a multiplicação encadeada, por envolver a chamada a uma rotina.

Em expressões complicadas, o compilador pode reconhecer subexpressões que se repetem se colocadas entre parênteses, calculando-as uma única vez.

5.2.Conjuntos Ordenados

Programas numericamente intensivos normalmente gastam a maior parte do tempo de CPU no processamento de conjuntos, como vetores e matrizes. A organização desse processamento de forma a minimizar transferências de memória é fundamental quando se considera o desempenho.

As principais técnicas, nesse caso, consistem em procurar minimizar os passos do acesso a elementos dos conjuntos. Também há casos em que aspectos da arquitetura devem ser levados em conta.

Os principais aspectos dessas técnicas são resumidos a seguir.

5.2.1. Minimização de Passo

No fragmento de código 01 acarreta apenas uma falha de *cache* a cada 16 elementos e uma de TLB a cada iteração, enquanto que no código 02 ocorre uma falha de TLB a cada elemento.

Código 01 - fragmento não otimizado

```
real(8) A(513,512)
! potência de 2 pode reduzir cache ao número de classes de !congruência
do  i = 1, 512, 1
   do  j = 1, 512, 1
      A(i,j) = A(i,j)*Y
      ! passo de uma página
   end do
end do
```

Código 02 - fragmento otimizado

```
real(8) A(513,512)
do  j = 1, 512, 1
  do i = 1, 512, 1
    A(i,j) = A(i,j)*Y
    ! passo de um elemento
  end do
end do
```

5.2.2. Laços com Dependência

No fragmento abaixo há dependência de dados - mais de um elemento de A é acessado por iteração. A reversão do laço j mantém o mesmo resultado.

Fragmento não otimizado

```
do  i = 1, 99
  do  j = 2, 100
    A(i,j) = A(i+1, j-1)*Y
    ! passo 100
  end do
end do
```

Fragmento otimizado

```
do  j = 100, 2, -1
  do  i = 2, 99
    A(i,j) = A(i+1, j-1)*Y
    ! passo 1
  end do
end do
```

5.2.3. *'Loop skewing'*

A dependência de dados não é tão simples no caso abaixo, no qual a reversão de laços não é suficiente para impedir a alteração de resultados com a troca de laços para a minimização de passos. A técnica denominada *loop skewing* fornece um algoritmo para resolver esse problema:

Fragmento não otimizado

```
do  i = 2, n
  do  j = 1, m-1
    A(i,j)= A(i-1, j+1) +A(i,j)
  end do
end do
```

Fragmento otimizado

```
do  j = 3, m+n-1
  do i = max0(2,j-n+1), min0(n,j-1)
    A(i,j-i) = A(i-1, j-i+1) + A(i,j-i)
  end do
end do
```

A solução por *loop skewing* se torna mais elaborada na medida em que a dependência de dados se torna mais complexa ou os laços possuem camadas mais profundas.

5.2.4. 'Blocking'

Blocking permite reduzir falhas de *cache* ou TLB, conforme ilustra o exemplo abaixo - para linha de cache de 128 bytes.

Fragmento não otimizado

```
real(8) A(800,160),B(160,800), C(800,160)
do  j = 1, 160
  do  i = 1, 800
    C(i,j)=C(i,j)+A(i,j)*B(j,i)
    ! passo 160 para B
  end do
end do
```

Fragmento otimizado

```
real(8) A(800,160),B(160,800), C(800,160)
do  jj = 1, 160, 16
  do  i = 1, 800
    do  j = jj, jj+15
      C(i,jj) =C(i,jj)+A(i,jj)*B(j,i)
    end do
  end do
end do
```

Apesar do aumento de um um laço, o passo para B é reduzido a 1, enquanto blocos de 16*16 são processados em 16 iterações de i. (Recomendável aumentar o bloco, a fim de minimizar o fato de que o primeiro elemento pode não ser o início de uma linha de *cache*.)

5.2.5. Matriz Triangular com 'blocking'

A troca de laços não altera a situação para as matrizes triangulares abaixo, que, no entanto, se beneficiam de um processamentos em blocos.

Fragmento não otimizado

```
n=250
do  i = 1, n
  do  j = i, n
    A(i,j)=A(i,j)+B(j,i)
  end do
end do
```

Fragmento otimizado

```
n=250
do jj=1,n,50
  do ii=1,n,50
    do i=ii,min0(ii+49,n)
      do j=max0(i,jj),min0(jj+49,n)
        A(i,j)=A(i,j)+B(j,i)
      end do
    end do
  end do
end do
```

5.3. Outras Otimizações de Laços

Além das técnicas de minimização de passos e utilização de blocos, que diminuem o custo de acesso à memória, existem outras transformações de laços que podem ser utilizadas na melhoria do desempenho. Com isso, economiza-se ciclos na inicialização e atualização de variáveis de laços e ramificação, na eliminação de carregamentos e armazenamentos da memória e na eliminação de dependências em FMAs.

Essas técnicas são ilustradas a seguir.

5.3.1. Fusão de Laços

não otimizado

```
do i=1,1000
  X=X*A(i)+B(i)
end do
  do i=1,1000
    Y=Y*A(i)+C(i)
  end do
```

otimizado

```
do i=1,1000
  X=X*A(i)+B(i)
  Y=Y*A(i)+C(i)
end do
```

No exemplo acima, há uma grande economia devido ao custo do processamento do segundo laço (descontado o processamento aritmético).

A fusão em certas condições poderia deixar de ser tão interessante se envolvesse uma extensão maior que a do *cache*.

5.3.2. Troca de Ordem

Loop interchange diminui número de execução de laços e de passos de acesso embora altere a estrutura das variáveis.

Exemplo:

Fragmento não otimizado

```
real(8) B(2,40,200)
do k=1,200
  do j=1,40
    do i=1,2
      B(i,j,k)=B(i,j,k)*0.1+1.
    end do
  end do
end do
```

Fragmento otimizado

```
real(8) B(200,40,2)
do i=1,2
  do j=1,40
    do k=1,200
      B(i,j,k)=B(i,j,k)*0.1+1.
    end do
  end do
end do
```

5.3.3. Desfatoração

A desfatoração no fragmento do código2 transforma o laço j do código1 numa única FMA:

Código1- fragmento não otimizado

```
do i=1,100
  A(i)=0.0d0
  do j=1,100
    A(i)=A(i)+B(j)*D(j)*C(i)
  end do
end do
```

Código2- fragmento otimizado

```
do i=1,100
  A(i)=0.0d0
  do j=1,100
    A(i)=A(i)+B(j)*D(j)
  end do
  A(i)=A(i)*C(i)
end do
```

5.3.4. Desdobramento de Laço

Loop unrolling pode reduzir ou eliminar dependência, eliminar carregamentos e armazenamentos na memória principal, reduzir em ciclos o custo com laços, bem como propiciar melhores oportunidades de otimização aos compiladores, devido aos blocos maiores resultantes dos desdobramentos.

Comparando o exemplo de troca de ordem anterior, o desdobramento possibilita uma economia não muito inferior, sem alterar a estrutura das variáveis:

Fragmento não otimizado

```
do k=1,200
  do j=1,40
    do i=1,2
      B(i,j,k)=B(i,j,k)*0.1+1.
    end do
  end do
end do
```

Fragmento otimizado

```
do k=1,200
  do j=1,40
    B(1,j,k)=B(1,j,k)*0.1+1.
    B(2,j,k)=B(2,j,k)*0.1+1.
  end do
end do
```

No exemplo a seguir, embora teoricamente não haja dependência, esta pode ocorrer no processamento com o armazenamento de A no mesmo registro que receberá o resultado. Este efeito é eliminado com o desdobramento, gerando melhores resultados se for feito a um nível 3 ou superior, para maximizar o uso da unidade BPU.

5.3.5. Desdobramento de Laço Externo

Permite um fluxo de instruções mais eficiente e redução de carregamentos e armazenamentos da memória. Exemplo:

Fragmento não desdobrado

```
do j=1,20
  do i=1,20
    A(i,j)=B(i,j)+C(i)
  end do
end do
```

Fragmento desdobrado

```
do j=1,20,4
  do i=1,20
    A(i,j)=B(i,j)+C(i)
    A(i,j+1)=B(i,j+1)+C(i)
    A(i,j+2)=B(i,j+2)+C(i)
    A(i,j+3)=B(i,j+3)+C(i)
  end do
end do
```

6.Entrada/Saída

Operações de entrada/saída de dados podem limitar drasticamente o desempenho, por acessarem muito lentamente a memória, além do consumo de CPU das próprias rotinas de E/S. Recomendações:

- restringir essas operações ao mínimo indispensável;
- usar dados não formatados, pois ocupam menos memória e não precisam ser convertidos evitando assim, arredondamentos;
- ler todos os dados de uma vez;
- não utilizar E/S dentro de laços;
- otimizar tamanho de registros; ex.: para um *cache* com linha de 128 bytes, o registro ideal é de 120; em arquivos não formatados são utilizados 4 bytes no início e 4 no final de cada registro; tamanhos iguais a potências de 2 são interessantes.

7.Outras técnicas

Outras técnicas poderiam ser lembradas. Em especial, as que permitem uma economia de ciclos com o controle do programa. Por exemplo, reduzindo ao mínimo as chamadas a subrotinas, chamando-as uma única vez, através do programa principal ou rotinas que as invocam. Esta técnica é conhecida como *inlining*.

Parte III : Otimização pelo Compilador

1. Introdução

Os compiladores automatizam a otimização. Técnicas manuais pressupõem a utilização conjunta das otimizações de compilador.

Compiladores não realizam todas otimizações possíveis. Em certos casos deixam de aplicar técnicas de que dispõem, por não poderem decidir algum conflito de segurança. Otimizações de compiladores às vezes podem alterar resultados, podendo deteriorar eficiência em vez de melhorar.

É preciso testar várias opções para um mesmo código, tanto para a eficiência quanto para a correção dos resultados, sendo recomendável dividir o programa em rotinas funcionalmente distintas, encontrando a combinação ideal de opções de compilação para cada uma, a fim de maximizar o desempenho.

2. Opções de Compilador (xlf, xlC)

Compiladores xlf, xlC (IBM) traduzem códigos f77 e f90 C e C++ para uma mesma linguagem intermediária, sendo então aplicadas as opções de otimização.

Um resumo das opções principais de otimização:

- -O: solicita otimização (nível 2 é o mínimo)
- -O2: otimização básica (o mesmo que **-O**)
- -O3: otimização mais agressiva. Deve-se tomar cuidado com esta opção já que a mesma pode alterar a semântica do programa, aumentar o tempo de compilação e o tamanho do código objeto. Utilizando-se juntamente com o -O3 o flag –qstrict, alterações desta natureza são impedidas.

  O compilador Fortran também admite outras opções de flag, dentre as quais:

-qhot	realiza algumas otimizações adicionais; requer ao menos **-O2**;
-qarch=p2sc	especifica a plataforma **power2** super chip (ou outra especificada)
-qtune=pwr2	otimiza para a plataforma power2 (relacionado com **-qarch**)
-qcache=auto	especifica a configuração de *cache* (relacionado com -qarch e -qtune e só tem efeito se **-qhot** estiver especificado); exemplo: **auto** especifica que seja detectada e especificada automaticamente.

3. Bibliotecas de Rotinas

As vantagens que as bibliotecas de rotinas apresentam aos usuários já foram comentadas: algoritmos muito eficientes, rotinas testadas e extremamente otimizadas. Freqüentemente as rotinas são implementadas para se ajustarem ao ambiente no qual serão utilizadas, extraindo o melhor rendimento para aquelas máquinas. Algumas vezes as rotinas chegam a ser implementadas e assembler, garantindo um grau máximo de otimização.

Todas as técnicas citadas aqui e muitas outras mais particulares são implementadas por essas bibliotecas.

3.1. Bibliotecas no CENAPAD-SP

No Cenapad-SP encontram-se instaladas várias bibliotecas de subrotinas numéricas (BLAS, LAPACK, NAG, ESSL, etc)

É interessante saber como utilizá-las.

De um modo geral, basta fazer uma chamada normal de subrotina no programa, invocando a rotina da biblioteca desejada. Depois é preciso informar na compilação a biblioteca que está sendo utilizada, indicando onde ela se encontra, se for o caso.

3.1.1. ESSL

Por exemplo, para utilizar uma rotina ESSL, basta chamá-la no programa e especificar sua utilização no *linking*. ESSL (e PESSL, a versão paralela) é um produto IBM, e por isso está finamente ajustada à arquitetura RS/6000.

Para usá-la em programas Fortran:

xlf90 -O *programa*.f -l essl -o *programa*

Utilizando-se -**lesslp2** é invocada a versão para a arquitetura power2 (-**lessl** é refere-se à arquitetura power, em geral).

Para programas em C ou C++, respectivamente:

cc -O *programa*.f -l essl -l xlf90 -o *programa*

xlC -O *programa*.f -l essl -l xlf90 -o *programa*

seguindo-se as convenções Fortran para chamada das rotinas.

Maiores informações podem ser obtidas no manual ESSL *on-line* no ambiente:

info -l essl

3.1.2. NAG

Outra biblioteca muito interessante é a NAG, disponível para diversas arquiteturas.

Para usá-la, é precisa indicar também sua localização, através da opção -**L**. No Cenapad-SP:

xlf -O *programa*.f -L/usr/local/NAG18/flib618da -lnag –o*programa*

Informações *on-line* com o comando:

NAG

4. REFERÊNCIAS

[1] Optimization and Tuning Guide for Fortran, C and C++. IBM Corporation.

[2] Performance Tuning. Maui High Performance Computing Center, 1997.

(http://www.mhpcc.edu/training/workshop/html/workshop.html)

[3] Performance Optimisation - Course Notes , Booth. S; MacDonald, N. - Edinburgh Parallel Computing Centre.

(http://www.epcc.ed.ac.uk/epcc-tec/documents/opt-course)

[4] Discussion: Performance Basics. Cornell Theory Center, 1998.

(http://www.tc.cornell.edu/Edu/Talks/course1.html)

[5] Single Processor Performance. Cornell Theory Center, 1998.

(http://www.tc.cornell.edu/Edu/Talks/course1.html)

[6] Single Processor Performance Considerations for the SP2, Cornell Theory Center, 1998.

(http://www.tc.cornell.edu/Edu/Talks/course1.html