UNIVERSIDADE ESTADUAL PAULISTA
JÚLIO DE MESQUITA FILHO
Campus de Presidente Prudente

CAIO TAKESHI ONISHI

Avaliação de Desempenho de Aplicações Paralelas Utilizando MPI e PETSc

Presidente Prudente
2011

CAIO TAKESHI ONISHI

Avaliação de Desempenho de Aplicações Paralelas Utilizando MPI e PETSc

Monografia apresentada como parte do Trabalho de Conclusão de Curso ao Departamento de Matemática, Estatística e Computação (DMEC), da Universidade Estadual Paulista "Júlio de Mesquita Filho", sob o título **"Avaliação de Desempenho de Aplicações Paralelas Utilizando MPI e PETSc"**.

Orientador: Prof. Dr. Ronaldo Celso Messias Correia

Presidente Prudente

2011

TERMO DE APROVAÇÃO

CAIO TAKESHI ONISHI

Avaliação de Desempenho de Aplicações Paralelas Utilizando MPI e PETSc

Monografia sob o título **"Avaliação de Desempenho de Aplicações Paralelas Utilizando MPI e PETSc"**, defendida por Caio Takeshi Onishi e aprovada em 20 de dezembro de 2011, em Presidente Prudente, Estado de São Paulo, pela banca examinadora constituída pelos seguintes membros:

Prof. Dr. Ronaldo Celso Messias Correia
Orientador

Prof. Dr. Rogério Eduardo Garcia

Prof. Ms. Fernando Pacanelli Martins

Presidente Prudente, 20 de dezembro de 2011

Dedico este trabalho à minha família,
que sempre me apoiou e incentivou
na busca pelos meus objetivos.

AGRADECIMENTOS

Primeiramente agradeço a Deus, por tudo que já conquistei e recebi em minha vida, e por me guiar sempre em tudo o que faço.

Agradeço a toda minha família, principalmente aos meus pais, Carlos e Matilde, por sempre terem me dado apoio incondicional em qualquer decisão que eu tomasse, e terem sempre me incentivado nas buscas pelos meus objetivos; à minha irmã, Carla, pelos anos de companheirismo, convivência e apoio; e aos meus tios, tias e avós por todo o carinho e incentivo.

Também agradeço à minha namorada, Larissa, por todo o carinho e compreensão neste último ano, me dando forças nos momentos mais difíceis.

Agradeço a todas as pessoas que conheci em Presidente Prudente, que fizeram destes anos os melhores possíveis; aos membros da república Butantan: Bolinha, Birigui, Toiço, Moller, Allan, Diogo, Kaótico, Ademar e Codorna; e a todos os amigos que compartilharam momentos comigo durante a faculdade.

Aos meus companheiros de estágio, Cho, Fabrício e Max, pelo conhecimento compartilhado; ao vice-diretor do Serviço Técnico de Informática da Unesp de Presidente Prudente e grande amigo, Raphael Garcia, por ser um modelo de esforço, dedicação e caráter, que certamente me acresentou muito, pessoal e profissionalmente; e a todos os demais funcionários do STI.

Agradeço a todos os professores pelos ensinamentos; ao Prof. Dr. Rogério Eduardo Garcia, pelas sugestões em relação a este trabalho; ao Fernando Pacanelli pela ajuda e paciência, oferecendo suporte com o *cluster*; e ao meu orientador, Prof. Dr. Ronaldo Celso Messias Correia, pela orientação e apoio durante o desenvolvimento deste trabalho.

E, finalmente, à minha segunda família, que conviveu comigo na república Hangar: Dida, Felca, Doidera e Tio, pelos momentos inesquecíveis, as festas, o futebol, as noites de *video game*, a parceria nos estudos e a amizade sincera por todos estes anos.

RESUMO

Este trabalho apresenta um estudo sobre o uso de padrões e diretivas de programação paralela em sistemas distribuídos, utilizando o padrão MPI e o conjunto de ferramentas PETSc, realizando uma análise e comparação de seus desempenhos com certos problemas matemáticos envolvendo matrizes. Os conceitos são aplicados para a resolução de problemas que envolvem Análise de Componentes Principais (PCA), que são executadas em um *cluster* Beowulf. Os resultados obtidos são comparados com os de uma aplicação análoga de execução sequencial, e em seguida é analisado se houve ganho no desempenho da aplicação paralela.

Palavras-Chave: Aplicações Distribuídas. Cluster. MPI. PETSc. PCA.

ABSTRACT

This work presents a study about the use of standards and directions on parallel programming in distributed systems, using the MPI standard and PETSc toolkit, performing an analysis of their performances over certain mathematic operations involving matrices. The concepts are used to develop applications to solve problems involving Principal Components Analysis (PCA), which are executed in a Beowulf cluster. The results are compared to the ones of an analogous application with sequencial execution, and then it is analized if there was any performance boost on the parallel application.

Keywords: Distributed Applications. Cluster. MPI. PETSc. PCA.

Sumário

Lista de Figuras

Capítulo 1

INTRODUÇÃO

O crescente surgimento de problemas complexos, que exigem alto poder de processamento, impulsionou a busca por métodos que pudessem oferecer soluções em tempo hábil. Os avanços da tecnologia seguiram uma escala exponencial, com a invenção e evolução de componentes de capacidades que seriam inimagináveis há 20 anos atrás.

No entanto, estes avanços no *hardware* se aproximam de um ponto em que a velocidade de evolução não é mais tão elevada (NASCIMENTO, 2010 apud GALLINA, 2006), pois se torna cada vez mais difícil criar componentes que ultrapassem os limites atuais.

Devido a essa preocupação, surgiu um ramo na computação chamado Computação de Alto Desempenho (*High Performance Computing* - HPC), que visa à busca por maneiras de se obter o maior desempenho possível do *hardware* e do *software*, estudando técnicas para acelerar o processamento de informações.

A Computação de Alto Desempenho se baseia principalmente no uso de supercomputadores ou aglomerados de computadores (*clusters*) que, aliados a técnicas como o processamento paralelo de informações, têm sido amplamente utilizados para a resolução de problemas que requerem elevado poder computacional. Por este motivo é fundamental estudar algoritmos, padrões e diretivas para o desenvolvimento de aplicações que possibilitam usufruir deste alto poder de processamento.

O uso de *clusters* na HPC é uma alternativa barata e eficiente, além de ser o sistema mais popular entre os computadores mais poderosos do mundo (TOP500, 2011). Isto comprova a relevância de se trabalhar com este sistema para se obter melhor desempenho nas aplicações.

Dentre os padrões de programação paralela, um dos mais difundidos é a Interface de Passagem de Mensagens (*Message Passing Interface* - MPI), que é composta por diretivas e bibliotecas que possibilitam o desenvolvimento de aplicações paralelas distribuídas.

Este trabalho combina o poder computacional do *cluster* com as técnicas e padrões da programação paralela, para o desenvolvimento de aplicações paralelas e posterior análise e comparação com sua correspondente sequencial.

1.1 Objetivos do Trabalho

O objetivo deste trabalho é estudar e apresentar padrões, bibliotecas, métodos, rotinas e ferramentas para desenvolvimento de aplicações para sistemas de memória distribuída, mais especificamente *clusters*, com o objetivo de aplicar estes conhecimentos na implementação de aplicações numéricas que demandem alto poder computacional e realizar testes e análises a fim de avaliar o desempenho e eficiência de tais aplicações.

1.2 Organização do Texto

Este trabalho está dividido em 7 capítulos, incluindo esta introdução. No Capítulo 2, são apresentados conceitos de *Cluster*, explicando seu funcionamento, componentes e modelos. O Capítulo 3, por sua vez, introduz o padrão de programação paralela de passagem de mensagens, o *Message Passing Interface* (MPI), mostrando seus recursos e rotinas principais, além de apresentar o pacote *Portable, Extensive Toolkit for Scientific Computation* (PETSc), que oferece suporte e facilidades para o uso do MPI, e algumas de suas funcionalidades. O Capítulo 4 aborda técnicas de avaliação de desempenho, descrevendo maneiras de se medir o desempenho de uma implementação. O Capítulo 5 explica o método de Análise de Componentes Principais. No Capítulo 6 são apresentados os testes e resultados obtidos com a implementação. E, por fim, o Capítulo 7 apresenta as conclusões deste trabalho.

Capítulo 2

CLUSTERS

2.1 Introdução

Um *cluster* é um sistema composto por um conjunto de computadores independentes interconectados por uma rede, que podem trabalhar de maneira individual ou coletiva. Cada computador, denominado nó, pode ser uma estação de trabalho, uma máquina PC (*personal computer*) ou um servidor SMP (*symmetric multiprocessor*) (PASSOS, 2006).

Devido ao alto poder computacional obtido com a combinação de diversas máquinas, existem várias finalidades para o emprego de um *cluster*, como por exemplo: alta capacidade e desempenho para resolução de um único problema, alto desempenho sobre processos com grande carga de trabalho, redundância (executar um processo em vários nós simultaneamente para garantir que não haverá falhas), execução de atividades em paralelo, acesso múltiplo de discos ou múltiplas entradas e saídas (BAKER, 2000).

2.2 Tipos de *Clusters*

Por conta das variadas motivações para o uso de *clusters*, eles podem ser divididos em três categorias: *cluster* de alta disponibilidade (*High Availability*), de balanceamento de carga (*Load Balancing*) e *cluster* de desempenho (PASSOS, 2006), que são descritos a seguir.

2.2.1 *Clusters* de Alta Disponibilidade

Este tipo de *cluster* é construído para prover alto índice de disponibilidade de serviços e recursos. Em outras palavras, ele deve garantir que um determinado serviço esteja disponível ininterruptamente, além de garantir que não haja falhas ou perda de informação.

Para que isso ocorra, é necessário um *software* que monitore as máquinas em rede, e identifique se os serviços estão sendo executados corretamente. Caso alguma máquina falhe, outra deve assumir a tarefa que estava sendo executada por ela (CARVALHO, 2007). Por isso, sua principal característica é a redundância, e, portanto deve haver replicação de dados (*backup*) entre máquinas para evitar perda de informação. Normalmente, este tipo de *cluster* é usado em tarefas críticas, que não toleram falhas.

2.2.2 *Clusters* de Balanceamento de Carga

Os *clusters* de balanceamento de carga são usados para distribuir o fluxo e a carga de trabalhos entre máquinas de uma rede, e é uma solução para sistemas que recebem um elevado número de requisições, como por exemplo, servidores *web* (*web farms*) (PASSOS, 2006).

Nesta implementação, é necessário que haja monitoramento da comunicação e redundância, de modo que os trabalhos sejam corretamente distribuídos para os nós disponíveis a cada momento, e não haja perda de informação.

2.2.3 *Clusters* de Desempenho

Clusters de desempenho, ou de processamento distribuído, têm o objetivo de oferecer maior poder de processamento, e, consequentemente, alto desempenho, e normalmente são empregados na execução de tarefas computacionais de alto custo (CARVALHO, 2007), que possam ser divididas em pequenas "subtarefas" e processadas paralelamente entre os diferentes nós do *cluster*.

Uma implementação bastante popular de *cluster* de desempenho é o *cluster* Beowulf, que foi criado pela NASA em 1994, e é amplamente usado por empresas e universidades, principalmente na área de computação científica.

Seu modelo é baseado em máquinas comuns (computadores pessoais, por exemplo), uma

rede privada, e um sistema operacional (BEOWULF, 2007). Desta forma, é possível se obter um grande poder computacional a partir de componentes de baixo custo financeiro, comparado a equipamentos de última tecnologia.

O *cluster* Beowulf é composto por vários nós, sendo um, denominado nó mestre, responsável por gerenciar e administrar a submissão de tarefas, enquanto os demais (nós escravos) são responsáveis por executá-las. A Figura 1 ilustra o esquema de um *cluster* Beowulf.

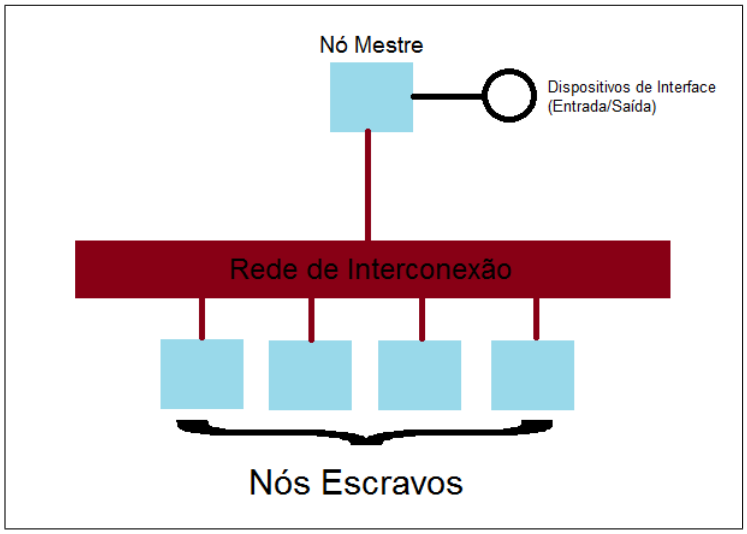


Figura 1: Esquema de um *Cluster* Beowulf.

A comunicação entre nós, que envolve envio e recebimento de requisições e dados processados, é feita pela rede, normalmente por meio de troca de mensagens ou *sockets* (BEOWULF, 2007).

Como cada computador executa suas tarefas independentemente, não existe restrição de compatibilidade entre as máquinas, desde que possibilitem a instalação do sistema Beowulf. Isso leva a outra vantagem, que é a facilidade de substituição de equipamentos defeituosos.

2.3 Componentes do *Cluster*

Segundo Baker (2000), os principais componentes para se construir um *cluster* são os nós (ou máquinas) que se encarregam de fazer a computação, e a rede de interconexão, que permite a troca de dados entre os nós. Estes nós podem ser computadores comuns, equipados com processadores (com um ou mais núcleos), memória, disco rígido e, dispositivos de interface. A rede também pode ser uma convencional, como Ethernet.

Além destes componentes de *hardware*, Baker (2000) e Caringi (2006) citam ainda componentes de *software*: um sistema operacional, que pode ser um derivado do UNIX, protocolos de comunicação, como o TCP/IP, por exemplo, um *middleware* (que faz o monitoramento e gerenciamento do *cluster*) e um ambiente de programação paralela.

2.3.1 Componentes de *Hardware*

Basicamente, o *hardware* necessário para o funcionamento de um *cluster* é composto por computadores e rede comuns. Com os recentes avanços da tecnologia, praticamente qualquer computador é elegível a ser usado como nó de um *cluster*. Desta forma, é fácil é barato conseguir componentes de *hardware* para a construção de um *cluster*.

2.3.1.1 *Hardware* dos Nós

Cada nó do *cluster* deve ser capaz de computar dados e armazená-las localmente. Por isso, a presença de alguns componentes no computador é essencial. São eles:

- **Processador:** componente básico para o processamento dos dados, pode ser de um núcleo ou mais;
- **Memória:** usualmente é usada a memória de acesso randômico (RAM), e serve como memória local de cada nó;
- **Disco rígido:** o disco serve para armazenamento local de dados de entrada ou de saída;
- **Dispositivos de interface:** necessários para comunicação, entrada e saída de informações.

2.3.1.2 Rede de Interconexão

A implementação de *clusters* só é possível se houver uma comunicação adequada entre seus diferentes nós. As redes de interconexão servem para enviar e receber dados que são processados em cada nó do *cluster*.

Aplicações paralelas distribuídas dependem ainda mais desta comunicação, pois cada nó processa uma parte do programa, necessária para que o programa seja executado como um todo.

Assim como os nós do *cluster* são compostos por componentes comuns, a rede também pode ser implementada de maneira simples, usando padrões básicos como o Ethernet, por exemplo.

Porém, à medida que a motivação do desenvolvimento de uma aplicação se baseia cada vez mais na obtenção de maior desempenho, muitas vezes são buscadas alternativas que diminuam a latência na comunicação entre os nós, já que esta interfere diretamente no tempo total de execução de um determinado programa.

Padrões como o Fast Ethernet, com taxa de transferência de 100Mb/s (megabits por segundo), ou o Gigabit Ethernet, com taxa de 1000Mb/s, têm sido usados para se obter uma comunicação mais rápida entre os nós. Outro padrão bastante usado é chamado Myrinet, com taxa de transferência de até 1Gb/s (gigabit por segundo). No entanto, sua implementação é mais custosa financeiramente (BAKER, 2000). Recentemente, tornou-se comum o uso de um padrão chamado InfiniBand, que atinge taxa de transferência de até 120Gb/s (IBTA, 2011).

2.3.2 Componentes de *Software*

Dentre os componentes de *software* essenciais para a configuração de um *cluster*, podem-se destacar: sistema operacional, linguagens, padrões, bibliotecas e ambientes de programação, bem como ferramentas de instalação, configuração, administração e monitoramento do sistema.

2.3.2.1 Sistema Operacional

A grande maioria dos *clusters* usa sistemas baseados em UNIX, como o Linux, por conta de seu suporte a diversas arquiteturas e liberdade de distribuição, além de oferecer características como multitarefa preemptiva, suporte multiusuário, memória virtual, suporte a vários processadores, entre outras (CARINGI, 2006).

No entanto, nada impede o uso do sistema operacional Windows, representado pelas distribuições *Microsoft Cluster Server*, *Component Load Balancing* e *Network Load Balancing Services*.

2.3.2.2 Sistemas de Gerenciamento e Monitoramento de Recursos

O controle e o monitoramento dos recursos do *cluster* são feitos por um software específico, que implementa serviços como submissão de tarefas, alocação, administração e liberação de recursos, balanceamento de carga, entre outros. Um exemplo de gerenciador de recursos é o TORQUE, que permite visualizar e monitorar o funcionamento e estado do sistema, e implementa os serviços supracitados.

2.3.2.3 Ambiente de Programação Paralela

O desenvolvimento de programas paralelos em sistemas distribuídos requer a comunicação entre os diferentes nós do sistema, para que as diversas partes que são processadas em cada nó se juntem para formar o resultado final. Essa comunicação entre os nós pode ocorrer por *sockets* ou por troca de mensagens, sendo a última o modelo mais comum.

Para que isso ocorra, é necessário que haja um ambiente que dê suporte a esta troca de mensagens, como por exemplo a Máquina Virtual Paralela (*Parallel Virtual Machine* – PVM), ou a Interface de Troca de Mensagens (*Message Passing Interface* – MPI) (BAKER, 2000), que oferecem condições que possibilitam a implementação de programas paralelos.

2.3.3 Desenvolvimento de Aplicações no Ambiente do *Cluster*

Com os diferentes tipos de arquiteturas, sistemas operacionais e *hardware* existentes, sempre existe o problema da portabilidade e do desenvolvimento para diferentes ambientes. Todo o processo de portabilidade, *redesign*, otimização e análise acaba se tornando extremamente caro, e certamente frustrante para o desenvolvedor. Por isso é de suma importância o uso de padrões e diretivas para o desenvolvimento de aplicações para *cluster*, especialmente quando se trata de programação paralela.

Um dos padrões mais aceitos no desenvolvimento de tais aplicações é o MPI, que é suportado pelas linguagens C, C++ e Fortran, e, basicamente, se encarrega de fazer a comunicação entre os diferentes nós do *cluster*, além de oferecer diversas rotinas que auxiliam na construção de aplicações paralelas distribuídas.

O funcionamento do padrão MPI e as funcionalidades que ele oferece estão detalhados no Capítulo 3.

2.4 Considerações Finais

O uso de *clusters* para a realização de tarefas que demandam alto poder computacional está cada vez mais difundido, se tornando muito mais popular que sistemas de processadores massivamente paralelos (*Massively Parallel Processors* - MPP), que são sistemas onde uma única máquina tem milhares de processadores acoplados, além de uma arquitetura especialmente projetada. A Figura 2 exibe um gráfico comparativo do uso dos diferentes tipos de sistemas entre os maiores supercomputadores do mundo. Essa populariação do uso de *clusters* se deve principalmente ao fato de seus custos serem relativamente baixos, sua implementação pouco complexa, e seu desempenho bastante elevado, como é indicado na Figura 3.

No entanto, para se desenvolver aplicações para *clusters* de desempenho, é necessário estudar seu funcionamento, e, principalmente as diretivas e padrões requeridos por este ambiente.

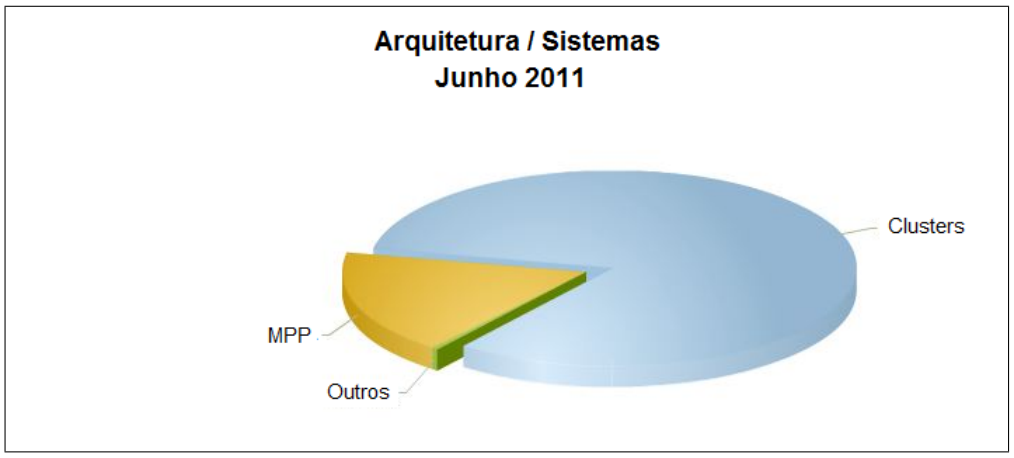


Figura 2: Uso de arquiteturas de supercomputadores (TOP500, 2011).

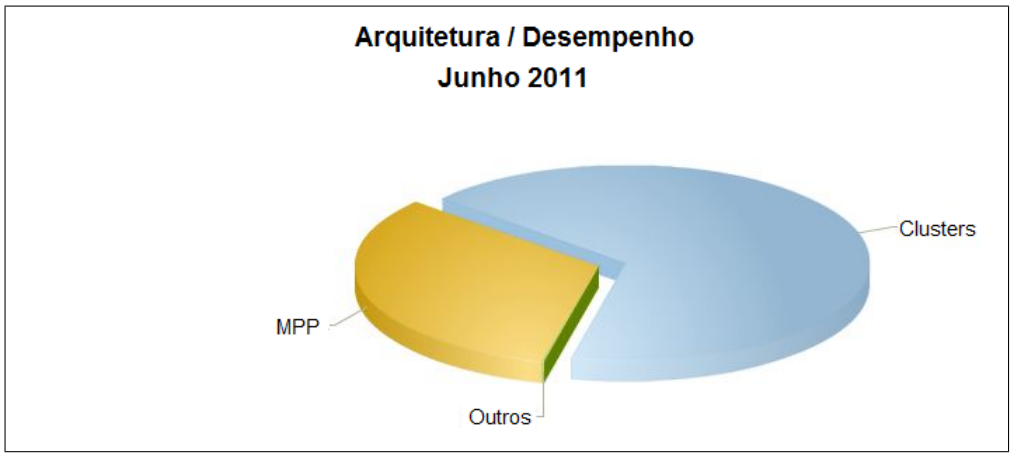


Figura 3: Desempenho de arquiteturas de supercomputadores (TOP500, 2011).

Capítulo 3

INTERFACE DE TROCA DE MENSAGENS

3.1 Introdução

O *Message Passing Interface* (MPI) é um padrão de interface de troca de mensagens para programação paralela em sistemas distribuídos. Ele inclui rotinas de comunicação ponto a ponto, operações coletivas, suporte a grupos de tarefas, contextos de comunicação e topologias de aplicação (SILVA, 2006), que possibilitam ao programador explicitar a divisão de tarefas.

As principais vantagens do padrão MPI são a portabilidade e a facilidade de implementação (MPI FORUM, 2009), uma vez que o mesmo código pode ser executado em diversas arquiteturas de *hardware* e sistemas operacionais, desde que haja uma biblioteca MPI disponível. Além de poder ser executado tanto em um sistema de nós distribuídos quanto em uma única máquina com múltiplos processadores.

3.2 Recursos do MPI

O padrão MPI oferece uma série de recursos para a implementação de uma aplicação paralela. Tais como:

- **Comunicação ponto a ponto:** É o mecanismo básico de comunicação. Trata o envio e recebimento de mensagens pelos processos.

- **Tipos de dados:** Tipos que podem ser manipulados, enviados e recebidos pelas rotinas do MPI. Muitos se equivalem aos tipos presentes na linguagem utilizada.
- **Operações coletivas:** Possibilita comunicação entre grupos de processos.
- **Grupos de processos:** O MPI possibilita a manipulação de grupos ordenados de processos separadamente.
- **Contextos de comunicação:** Dão a habilidade de ter contextos de troca de mensagens distintos, isolando-os de comunicações externas.
- **Topologias de processos:** Atributo opcional que possibilita criar uma topologia virtual dentro de um grupo de processos.
- **Gerenciamento do ambiente:** Habilidade de gerenciar o ambiente de execução, definindo versão, alocação de memória, controle de erros, sincronização e inicialização.
- **Criação e gerenciamento de processos:** Permite criar e gerenciar processos, auxiliando no controle de concorrência e recursos do sistema.
- **Comunicação unilateral:** É um tipo de comunicação que permite que um único processo faça o papel tanto de remetente quanto de destinatário, impedindo que ele fique esperando outros processos se tornarem disponíveis para sincronização.
- **Interfaces externas:** Permite criar camadas que permitem acessar determinados setores da implementação, auxiliando a criação de ferramentas (*debuggers*, analisadores de desempenho).
- **Entrada/saída paralelas de arquivos:** O MPI oferece suporte para manipulação e acesso a arquivos em implementações paralelas.
- ***Binding* para as linguagens C, C++ e Fortran:** Oferece funções, métodos ou subrotinas de acordo com a forma apropriada para cada uma dessas linguagens.

3.3 Comunicação

O MPI oferece duas maneiras de comunicação entre processos: a comunicação ponto a ponto e a comunicação coletiva, e o funcionamento de cada uma é explicado a seguir.

3.3.1 Comunicação Ponto a Ponto

Um processo se comunica com outro diretamente, por meio de rotinas de envio e recebimento de mensagens. A rotina pode ser bloqueante ou não. Rotinas bloqueantes deixam o processo bloqueado até que a mensagem esteja seguramente armazenada em um *buffer* (do receptor ou temporário, do sistema).

Esta comunicação pode ser no modo "bufferizado" (*buffered*), no qual a comunicação pode ser completada mesmo que a mensagem ainda não tenha chegado ao receptor correspondente. A mensagem fica armazenada em um *buffer* do sistema temporariamente.

A comunicação também pode ocorrer no modo síncrono. Neste modo é necessário que haja uma confirmação de recebimento da mensagem.

Outro modo de comunicação é o pronto (*ready*). Aqui, a comunicação é completada independente do *status* do receptor, a fim de que se obtenha um ganho no desempenho.

Por último, existe o modo de comunicação padrão; nele, o MPI decide se o sistema armazena a mensagem no *buffer* ou não. Nesse caso, o MPI leva em questão o espaço de armazenamento disponível e o desempenho.

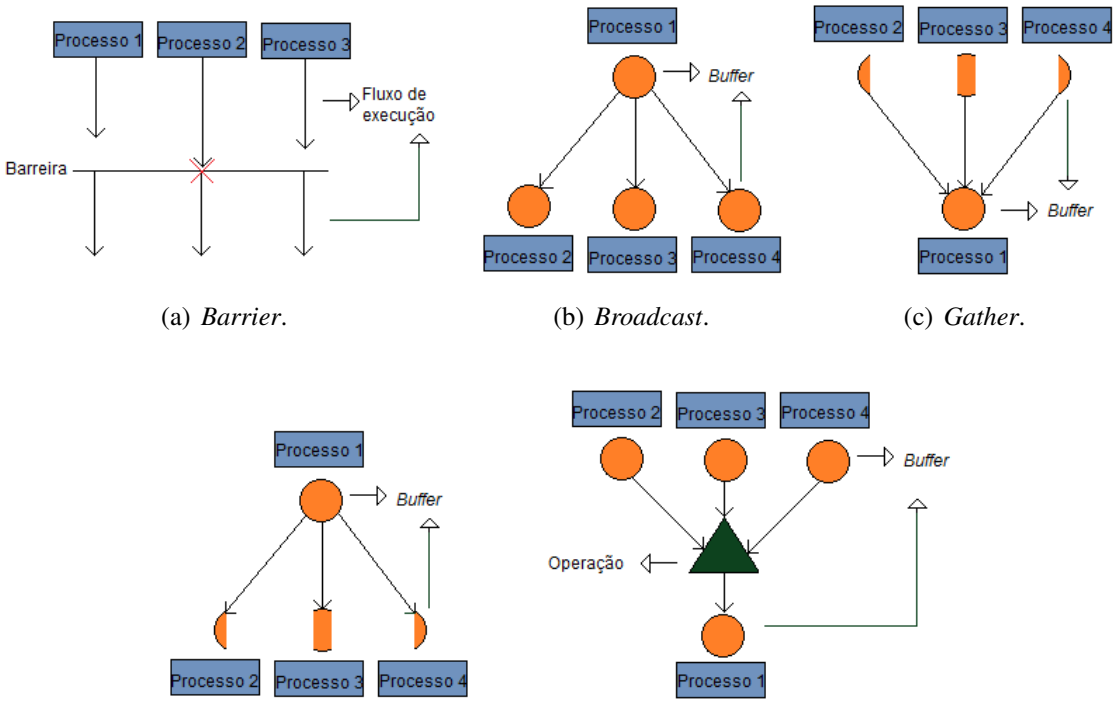
3.3.2 Comunicação Coletiva

É a comunicação que envolve um ou mais grupos de processos. Pode ocorrer entre vários processos do mesmo grupo (intracomunicação), ou entre processos de grupos distintos (intercomunicação).

Existem, portanto, algumas rotinas que executam essa comunicação coletiva. São elas:

- ***Barrier Synchronization:*** o processo permanece bloqueado até que todos os processos do grupo cheguem ao mesmo ponto de execução, seu comportamento está ilustrado na Figura 4(a);
- ***Broadcast:*** envia a mensagem para todos os processos do grupo, inclusive a ele mesmo, como ilustra a Figura 4(b);
- ***Gather:*** recebe os conteúdos dos *buffers* de envio de todos os processos do grupo, e seu comportamento pode ser observado na Figura 4(c);
- ***Scatter:*** os dados do processo são divididos e distribuídos entre os demais processos do grupo, assim como indica a Figura 4(d);

- ***Gather-to-all:*** funciona como o *Gather*, mas não só o processo *root* (emissor da rotina) recebe os dados, mas todos os demais do grupo;
- ***All-to-all Scatter/Gather:*** funciona como o *Gather-to-all*, porém, cada processo envia dados distintos para cada um dos destinatários;
- ***Reduce:*** executa uma operação com os dados dos *buffers* de todos os processos do grupo, e envia o resultado para o *root*, como está ilustrado na Figura 4(e);
- ***All-Reduce:*** igual ao *Reduce*, enviando o resultado para todos os processos do grupo;
- ***Reduce-Scatter:*** variante do *Reduce*, divide o resultado em blocos e distribui entre os processos.



(a) *Barrier*. (b) *Broadcast*. (c) *Gather*.

(d) *Scatter*. (e) *Reduce*.

Figura 4: Comportamento das rotinas de comunicação coletiva do MPI.

3.4 Rotinas do MPI

O padrão MPI oferece mais de uma centena de rotinas para programação, e algumas funções são fundamentais para iniciar a programação no MPI (NASCIMENTO, 2010):

- `MPI_Init`: Inicia uma computação MPI.
- `MPI_Finalize`: Finaliza uma computação MPI.
- `MPI_Comm_Size`: Retorna o número de processos dentro de um grupo.
- `MPI_Comm_Rank`: Identifica um processo dentro de um determinado grupo.
- `MPI_Send`: Envia mensagem.
- `MPI_Recv`: Recebe mensagem.
- `MPI_Bcast`: Envia mensagens para todos os processos de um grupo.
- `MPI_Barrier`: Bloqueia o processo até que os demais cheguem ao mesmo ponto de execução.

3.5 O Pacote PETSc

O *Portable, Extensive Toolkit for Scientific Computation* (PETSc) é um conjunto de estruturas de dados e rotinas que oferecem suporte para implementação de aplicações de alto custo computacional em sistemas paralelos, especialmente aplicações da área científica, desenvolvido por um grupo de pesquisadores e disponível no *site*: http://www.mcs.anl.gov/petsc/index.html.

O PETSc foi desenvolvido para auxiliar no desenvolvimento de aplicações que usem métodos numéricos de álgebra linear, tratando o paralelismo internamente, utilizando o padrão MPI para as trocas de mensagens. Em outras palavras, ele não exige que o paralelismo seja implementado explicitamente, executando operações matemáticas de modo paralelo por meio de suas rotinas, que fazem esse tratamento visando à otimização.

Este pacote oferece ferramentas para resolução de equações lineares e não-lineares e operações com vetores e matrizes, entre outras operações matemáticas, que podem ser usadas em códigos nas linguagens Fortran, C e C++ (BALAY et al., 2010).

3.5.1 Recursos do PETSc

O PETSc é composto por uma variedade de bibliotecas, sendo cada uma destinada à manipulação de um determinado tipo de objeto. Algumas das bibliotecas contidas no PETSc lidam com:

- Vetores;
- Matrizes esparsas ou densas;
- Solução de problemas não lineares (*Nonlinear Equations Solver* - SNES);
- Solução de sistemas lineares por métodos iterativos (*Krylov Subspace Methods* - KSP);
- Integração temporal de equações diferenciais ordinárias (*Timestepping*);
- Pré-condicionamento de matrizes.

Essas bibliotecas oferecem rotinas de manipulação e operações envolvendo estes tipos, facilitando consideravelmente o trabalho de desenvolvimento de aplicações. Na maioria das vezes, uma função realiza uma operação bastante específica, retornando o resultado, poupando o desenvolvedor das complicações da elaboração do algoritmo.

A Figura 5 mostra as principais bibliotecas do PETSc.

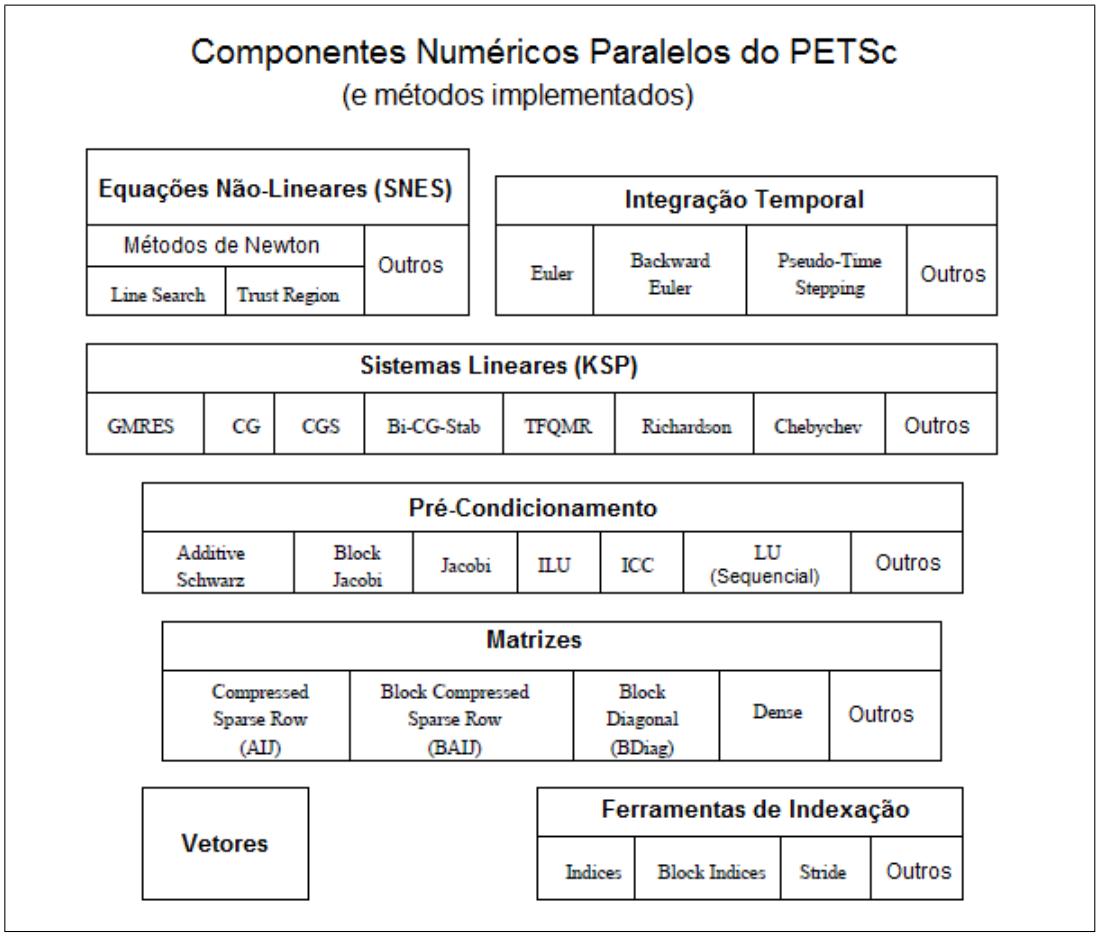


Figura 5: Bibliotecas específicas oferecidas pelo PETSc.

3.5.1.1 Vetores

O vetor é um dos objetos mais simples do PETSc. Seguem abaixo algumas das principais rotinas para manipulação de vetores:

- `VecCreateMPI(MPI_Comm comm, int m, int M, Vec *x);`

  Cria um vetor distribuído por todos os processos do comunicador, onde `m` indica o número de componentes por processo, e `M` é o número de componentes do vetor. É importante enfatizar que todos os processos do comunicador devem chamar as rotinas de criação.

- `VecSet(Vec x, PetscScalar value);`

  Atribui um valor para todos os componentes do vetor.

- `VecSetValues(Vec x, int n, int *indices, PetscScalar *values, INSERT_VALUES);`

  Atribui valores diferentes para componentes diferentes do vetor. Nesta rotina, `n` é o número de componentes sendo inseridos, `indices` corresponde aos índices dos componentes, e `values` é um *array* de valores a serem inseridos. Depois dessa rotina, é necessário chamar as rotinas `VecAssemblyBegin(Vec x)` e `VecAssemblyEnd(Vec x)`.

- `VecGetValues();`

  Retorna os valores do vetor do processo local.

- `VecDuplicate(Vec old, Vec *new);`

  Cria um novo vetor (`new`) igual a um vetor existente (`old`).

- `VecDestroy(Vec x);`

  Destrói um vetor.

- `VecGetLocalSize(Vec v, int *size);`

  Obtém o número de elementos armazenados localmente.

Existem também rotinas que executam operações envolvendo vetores. Algumas das principais delas são apresentadas no Quadro 1.

Quadro 1: Operações de vetor do PETSc.

Função	**Operação**
VecAXPY(Vec y, PetscScalar a, Vec x)	$y = y + a * x$
VecAYPX(Vec y, PetscScalar a, Vec x)	$y = x + a * y$
VecWAXPY(Vec w, PetscScalar a, Vec x, Vec y)	$w = a * x + y$
VecAXPBY(Vec y, PetscScalar a, PetscScalar b, Vec x)	$y = a * x + b * y$
VecScale(Vec x, PetscScalar a)	$x = a * x$
VecSum(Vec x, PetscScalar *r)	$r = \sum x_i$
VecCopy(Vec x, Vec y)	$y = x$
VecSwap(Vec x, Vec y)	$x = y$ e $y = x$

3.5.1.2 Matrizes

O PETSc oferece uma variedade de implementações para manipulação e operações com matrizes, que facilitam muito o processo de programação, pois não há necessidade de se criarem *arrays* bidimensionais, que seria o método convencional.

Algumas das rotinas básicas de manipulação de matrizes são:

- `MatCreate(MPI_Comm comm, Mat *A);`

  Cria uma matriz sequencial se estiver rodando em apenas um processo, ou paralela se estiver rodando em vários.

- `MatSetSizes(Mat A, int m, int n, int M, int N);`

  Os valores `m` e `n` definem as dimensões locais, enquanto `M` e `N` definem as globais.

- `MatSetValues(Mat A, int m, const int idxm[], int n, const int idxn[], const PetscScalar values[], INSERT_VALUES);`

  Insere valores na matriz, onde `idxm` e `idxn` indicam, respectivamente, o índice global da linha e da coluna. Os valores `m` e `n` indicam a dimensão do sub-bloco. O *array* `values` é bidimensional, e contém os valores a serem inseridos.

As operações incluem multiplicação de matrizes, cálculo da matriz transposta, soma com vetor, entre outras, como mostra o Quadro 2.

Quadro 2: Operações de matriz do PETSc.

Função	**Operação**
MatAXPY(Mat Y, PetscScalar a, Mat X, MatStructure)	$Y = Y + a * X$
MatMult(Mat A, Vec x, Vec y)	$y = A * z$
MatMultAdd(Mat A, Vec x, Vec y, Vec z)	$z = y + A * x$
MatMultTranspose(Mat A, Vec x, Vec y)	$y = A^T * x$
MatMultTransposeAdd(Mat A, Vec x, Vec y, Vec z)	$z = y + A^T * x$
MatScale(Mat A, PetscScalar a)	$A = a * A$
MatConvert(Mat A, MatType type, Mat *B)	$B = A$
MatGetDiagonal(Mat A, Vec c)	$x = diag(A)$
MatTranspose(Mat A, MatReuse, Mat *B)	$B = A^T$
MatZeroEntries(Mat A)	$A = 0$

3.5.1.3 Solução de Sistemas Lineares (KSP)

A biblioteca KSP (cujo nome vem de *Krylov Subspace*) é uma das principais do PETSc, porque oferece acesso a todos os pacotes de solução de sistemas lineares, incluindo as formas paralela e sequencial, direta e interativa (BALAY et al., 2010). Seu objetivo é resolver sistemas da forma $Ax = b$.

Principais rotinas:

- `KSPCreate(MPI_Comm comm, KSP *ksp);`

  Cria um contexto para a solução do sistema linear.

- `KSPSetOperators(KSP ksp, Mat Amat, Mat Pmat, MatStructure flag);`

  Atribui os operadores do sistema; `Amat` é a matriz que o define; `Pmat` é o precondicionador (i.e. a matriz a partir da qual `Amat` é construída), e `flag` é usado quando o mesmo método de precondicionamento é usado repetidamente em sistemas do mesmo tamanho.

- `KSPSolve(KSP ksp, Vec b, Vec c);`

  Resolve o sistema, onde `b` é o vetor que representa os valores à direita do sistema, e `c` é o vetor solução.

- `KSPSetType(KSP ksp, KSPType method);`

  Define o método a ser usado na resolução.

- `KSPDestroy(KSP ksp);`

  Destrói o contexto.

O Quadro 3 apresenta os métodos suportados para a resolução de sistemas lineares.

Quadro 3: Métodos para resolução de sistemas lineares.

Método	**KSPType**
Richardson	KSPRICHARDSON
Chebychev	KSPCHEBYCHEV
Gradiente Conjugado	KSPCG
Gradiente Biconjugado	KSPBICG
Gradiente Conjugado Quadrado	KSPCGS
Gradiente Biconjugado Estabilizado	KSPBCGS
Resíduo Mínimo Generalizado	KSPGMRES
Resíduo Quase-Mínimo Livre de Transpostas	KSPTFQMR
Resíduo Conjugado	KSPCR
Método dos Mínimos Quadrados	KSPLSQR

3.5.1.4 Solução de Equações Não-Lineares (SNES)

A biblioteca SNES contém implementações de métodos para a resolução de sistemas não-lineares da forma $F(x) = 0$.

Algumas rotinas básicas podem ser visualizadas seguir e também no Quadro 4:

- `SNESCreate(MPI_Comm comm, SNES *snes);`

  Cria o resolvedor SNES.

- `SNESSetType(SNES snes, SNESType method);`

  Define o método a ser usado na resolução, dentre os que estão na Tabela **??**.

- `SNESSolve(SNES snes, Vec b, Vec x);`

  Resolve o sistema, onde `x` é o vetor solução.

- `SNESDestroy(SNES snes);`

  Destrói o contexto.

Quadro 4: Métodos da biblioteca SNES para resolução de equações não-lineares.

Método	**SNESType**
Line Search	SNESLS
Trust Region	SNESTR
Test Jacobian	SNESTEST

3.5.1.5 Integração Temporal (TS)

A biblioteca *Timestepping* (TS) contém ferramentas para resolução de equações diferenciais ordinárias e equações diferenciais algébricas.

Seguem algumas de suas rotinas:

- `TSCreate(MPI_Comm comm, TSProblemType pt, TS *ts);`

  Cria o objeto TS, onde o parâmetro `ts` pode ser `TS_LINEAR` ou `TS_NONLINEAR`.

- `TSSetType(TS ts, TSType type);`

  Define o método de solução.

- `TSSolve(TS ts);`

  Resolve o problema.

- `TSDestroy(TS ts);`

  Destrói o objeto.

3.5.1.6 Checagem de Erros

Todas as rotinas do PETSc retornam um valor *integer*, que indica se houve erro durante a sua execução. Se a rotina for executada sem problemas, o retorno é zero. Caso contrário, outro valor é retornado.

É possível verificar o erro ocorrido por meio da macro `CHKERRQ`, que checa o valor de retorno da rotina e detecta o erro. É recomendável usar o `CHKERRQ` em cada sub-rotina, para se obter um relatório de erros completo.

3.6 PETSc e MPI

Como mencionado anteriormente, o PETSc utiliza o padrão MPI em suas trocas de mensagens. Desta forma, o paralelismo aplicado nas rotinas do PETSc é implementado, implicitamente, com MPI. Outra vantagem interessante é que é possível utilizar rotinas do MPI em qualquer parte do código.

Para se iniciar um programa PETSc, deve ser chamada a rotina *PetscInitialize*. Quando isso ocorre, automaticamente o MPI também é iniciado, caso ele já não esteja ativo, e o comunicador global (*world*) do PETSc passa a ser o comunicador global do MPI. Do mesmo modo, ao ser finalizado o PETSc (*PetscFinalize*), o MPI também é finalizado, mas também, somente caso tenha sido iniciado por ele. Ou seja, se o MPI foi chamado externamente, ele continuará em execução. É importante saber que é necessário especificar o comunicador a ser usado na criação de cada objeto do PETSc (vetor, matriz, etc.), para indicar a quais processadores o objeto será distribuído.

Apesar do fato do PETSc ser implementado com o MPI, há algumas diferenças na maneira como ele é usado e no tempo de execução entre uma aplicação escrita com MPI puro e outra com o auxílio do PETSc. A principal é que, mesmo que seja possível realizar comunicações entre os nós com o uso de funções do MPI no código feito em PETSc, as funções deste pacote não exigem isto. Ou seja, todo o paralelismo já está implementado nas funções do pacote.

Outra característica que se deve observar é que o PETSc usa estruturas de dados específicas, definidas pelas suas bibliotecas, para manipular os tipos de dados que foram descritos na Subseção 3.5.1. Por este motivo, a inicialização destas estruturas, muitas vezes, é mais custosa computacionalmente do que a inicialização de tipos primitivos da linguagem que está sendo usada. Tendo isso em vista, é recomendável analisar o comportamento dos programas feitos com estas ferramentas, para que seja possível avaliar quando é interessante usar cada uma, levando em consideração, também, que o PETSc tem uma vasta gama de funções otimizadas já implementadas, para a resolução de métodos numéricos usados em uma série de problemas. Uma análise comparativa entre a execução de programas implementados com MPI puro e PETSc pode ser encontrada no Capítulo 6.

Capítulo 4

AVALIAÇÃO DE DESEMPENHO

4.1 Introdução

O desempenho é uma característica muito importante no desenvolvimento de *software*, principalmente na área de programação paralela. Por isso, a sua avaliação é um estágio fundamental que visa à análise dos computadores de alto desempenho, dos algoritmos e das aplicações paralelas, bem como a busca por técnicas para obter melhores resultados.

O desenvolvimento de arquiteturas e *hardware* que possibilitam a execução simultânea de instruções computacionais alimenta a expectativa sobre o ganho de desempenho. Baseando-se na ideia de que quanto maior o poder de processamento, maior o desempenho de um programa, é comum acreditar que um computador com n processadores execute um programa n vezes mais rápido. No entanto, a avaliação, e principalmente a predição do desempenho em programas paralelos é algo muito mais complexo.

Portanto, dificilmente aquela afirmação é verdadeira. Isto porque certos algoritmos contêm partes estritamente sequenciais, que não podem ser paralelizados por influenciar no resultado final, ou por depender de recursos que ainda não foram definidos ou estão em outra parte da aplicação que ainda não foi executada.

Por conta disso, muitos fatores devem ser levados em conta quando se deseja avaliar o desempenho de um programa paralelo, como tempo de execução, escalabilidade, mecanismos pelos quais os dados são gerados, armazenados, transmitidos e movidos do disco e para o disco (PASSOS, 2006), e sistema, padrões e bibliotecas usados.

Para que se possa compreender, e principalmente, tentar-se obter o maior nível de paralelismo possível, é necessário, além do *hardware* apropriado, que se aborde a aplicação em questão adequadamente.

4.2 Técnicas de Avaliação de Desempenho

Segundo Laine (apud JAIN, 1991), existem basicamente duas situações em que se utilizam técnicas de avaliação de desempenho. Na primeira, com o propósito de se comparar sistemas diferentes e descobrir qual é o mais eficiente; na outra situação, desejamos estudar os parâmetros do sistema e descobrir quais os melhores valores para eles, de modo que o desempenho da aplicação seja melhor.

Uma das técnicas de avaliação e modelagem citadas por Laine (2003) é a medição, que consiste em obter medidas de parâmetros do sistema. Esta técnica pode ser aplicada somente com o sistema totalmente implementado, e pode ser usada para avaliar ou comparar o desempenho de sistemas semelhantes. Para realizar as medições, são usadas ferramentas que permitem monitorar o comportamento do *software* e/ou do *hardware*.

O objetivo é analisar as métricas que são extraídas das características do *software* e do ambiente em que ele está sendo executado. Alguns dos itens analisados incluem: o número de nós, o caminho a ser percorrido, o tamanho do problema, tempo de computação, comunicação e execução, volume de comunicação, entrada/saída, entre outros (SANTOS, 2005).

Santos (apud EAGER, 1989) destaca três medidas para a caracterização de desempenho de um sistema paralelo. São elas: tempo de execução, eficiência e *speedup*.

4.2.1 Tempo de Execução

O tempo de execução é uma das métricas mais usadas e mais confiáveis quando se quer traduzir a velocidade de processamento, tanto do *hardware* quanto do *software* (SILVA, 2006).

O tempo de execução serial é o tempo decorrido do início da execução do programa até o seu término. Já o paralelo é o tempo decorrido do momento em que o programa se inicia até que o último processador empregado termine sua execução (SANTOS, 2005).

Uma definição bastante aceita de desempenho (D) é como sendo o inverso do tempo de execução (T). Desta forma:

$$D = \frac{1}{T} \tag{4.1}$$

Esse tempo de execução é a soma do tempo de computação (T^i_{comp}), tempo de comunicação (T^i_{comm}) e tempo ocioso (T^i_{ocioso}), para cada processador i. Sendo assim, Passos (2006) define o tempo de execução como a soma daqueles tempos de todos os processadores dividida pelo número de processadores (P):

$$T = \frac{1}{P}(\sum_{i=0}^{p-1} T^i_{comp} + \sum_{i=0}^{p-1} T^i_{comm} + \sum_{i=0}^{p-1} T^i_{ocioso}) \tag{4.2}$$

O tempo de computação é o tempo em que o processador está efetivamente trabalhando na execução da tarefa, este tempo depende diretamente do tamanho do problema e da configuração do computador em que o programa está sendo executado.

Tempo de comunicação é aquele gasto com as trocas de mensagens, ou seja, a comunicação entre os diversos nós do sistema, e é composto pela soma do tempo necessário para iniciar a comunicação, mais o tempo de transferência que é determinado pelo canal de comunicação.

Por fim, o tempo ocioso é o tempo que o processador fica sem realizar instruções. Ele pode ocorrer quando o processador fica aguardando novas instruções virem de outros processadores. Por isso é fundamental que o programa esteja estruturalmente eficiente quanto ao seu paralelismo, de modo que diminua esta ociosidade.

4.2.2 Eficiência

A eficiência é uma métrica intrínseca ao tempo de execução. Ela caracteriza a efetividade com a qual o algoritmo usa os recursos do computador. Passos (apud FOSTER, 1995) a define por:

$$E = \frac{T_1}{PT_p} \tag{4.3}$$

sendo T_1 é o tempo de execução sequencial, ou seja, em um processador, e T_p é o tempo de

execução em P processadores.

Como se nota, a eficiência é uma medida relativa e não absoluta, pois leva em conta o tempo de execução de um algoritmo em múltiplos processadores em relação ao tempo de execução em apenas um.

Logo, ela é útil para se comparar as vantagens (ou desvantagens) de um programa paralelo sobre um executado sequencialmente.

4.2.3 *Speedup*

Outra importante métrica para se avaliar o desempenho de um sistema é chamada de *speedup*, e refere-se a quanto um algoritmo paralelo é mais rápido que seu correspondente sequencial.

O *speedup* é dado pela razão entre o tempo de execução sequencial e o tempo de execução paralelo (SILVA, 2006 apud FOSTER, 1995):

$$S_p = \frac{T_1}{T_p} \tag{4.4}$$

Sendo assim, o *speedup* idela é obtido quando $S_p = p$, o que significa que o tempo de execução diminui na mesma proporção que se aumenta o número de processadores.

4.3 Considerações Finais

Apesar da expectativa sobre o aumento do desempenho com o aumento do poder de processamento ou com o paralelismo da aplicação, muitas vezes a aplicação não se comporta bem quando paralelizada, principalmente se houver alta dependência entre seus módulos, ou se seu comportamento for predominantemente sequencial.

Outros fatores que exercem grande influência na análise final são o tempo de comunicação e o tempo ocioso. É possível que o tempo que os nós do sistema levam para se comunicar e trocar mensagens com requisições e resultados seja maior do que o tempo que seria gasto se um único processador estivesse realizando a tarefa, obtendo os resultados localmente.

Da mesma forma, pode ocorrer do fluxo do programa parar, à espera da execução de outro trecho, em outro nó, que seja essencial para a continuidade da execução total.

Portanto, é fundamental a realização de estudos e testes sobre a aplicação antes de se afirmar qualquer coisa em relação ao ganho de desempenho ao paralelizá-la e executá-la em um sistema distribuído.

Capítulo 5

ANÁLISE DE COMPONENTES PRINCIPAIS

5.1 Introdução

A Análise de Componentes Principais (*Principal Component Analysis* – PCA), também chamada de transformada de Hotelling ou de Karhunen-Loève, é um método estatístico muito usado para análise de dados multivariados. Sua aplicação tem por objetivo representar dados multivariados na forma de um novo conjunto, com menor quantidade de variáveis. Em outras palavras, ela permite analisar um conjunto por meio de componentes que o representam de uma forma mais simplificada.

A PCA pode ser aplicada em diversas áreas, sendo atualmente mais comum como ferramenta para análise exploratória de dados, reconhecimento de padrões, ou na criação de modelos preditivos. São destacados ainda o uso da PCA com os seguintes enfoques:

- Simplificação do conjunto de dados: encontrar uma maneira mais simplificada de representar um universo de estudo;
- Classificação: agrupar variáveis ou analisar a dispersão de indivíduos no espaço;
- Análise de interdependência: verificar se um conjunto de variáveis é dependente das demais;
- Formulação e prova de hipóteses: encontrar modelos que permitam formular hipóteses a partir de um conjunto de dados.

Segundo Bergamo (2006), a técnica de PCA torna mais conveniente a análise de um conjunto de dados, representando-os na forma de combinações lineares, reduzindo a dimensão do conjunto analisado, e, consequentemente, o número de variáveis, uma vez que as primeiras componentes principais detêm mais de 90% da informação estatística dos dados originais.

5.2 Método

O método para calcular as componentes principais segue os seguintes passos:

- Obtenção dos dados a serem analisados;
- Cálculo da média destes dados;
- Subtração da média de todas as amostras;
- Cálculo da matriz de covariância;
- Cálculo dos autovalores e autovetores da matriz de covariância;
- Montagem da matriz com as componentes principais.

Primeiramente, é montada uma matriz com as amostras dos dados a serem analisados, onde cada linha representa uma amostra, e cada coluna representa uma variável. O objetivo da PCA é representar estes dados de forma simplificada. Portanto, será possível visualizar graficamente, em duas ou três dimensões, dados que eram escritos em função de quatro ou mais dimensões.

O primeiro procedimento ao qual os dados se submetem é a subtração da média em cada dimensão, ou seja, é calculada a média de cada coluna, e, em cada linha, na coluna correspondente, é subtraído este valor. Desta forma, seja X um conjunto de amostras de uma variável, X_1 a primeira amostra e X_n a última, a média é calculada por:

$$\bar{X} = \frac{\sum_{i=1}^{n} X_i}{n} \tag{5.1}$$

Assim, um novo conjunto de dados é obtido pela fórmula:

$$Z_i^X = X_i - \bar{X} \tag{5.2}$$

Este procedimento deve ser feito para cada coluna ou variável do conjunto.

5.2.1 Matriz de Covariância

A covariância, em definição sucinta, mede o grau de interdependência numérica entre duas variáveis. Portanto, ela é medida aos pares de variáveis, e sua fórmula é dada por:

$$cov(X,Y) = \frac{\sum_{i=1}^{n} [(X_i - \bar{X}) \cdot (Y_i - \bar{Y})]}{n} \tag{5.3}$$

No caso dos dados que serão usados na técnica de PCA, cada variável representa uma coluna da matriz de amostras, e cada linha terá seus valores de covariância correspondentes. Logo, se os dados tiverem três dimensões (X, Y, Z), por exemplo, a matriz de covariância terá o seguinte formato:

$$M_{cov} = \begin{pmatrix} cov(X,X) & cov(X,Y) & cov(X,Z) \\ cov(Y,X) & cov(Y,Y) & cov(Y,Z) \\ cov(Z,X) & cov(Z,Y) & cov(Z,Z) \end{pmatrix}$$

Pode-se concluir que a matriz é simétrica sobre a diagonal principal, pois:

$$(X_i - \bar{X})(Y_i - \bar{Y}) = (Y_i - \bar{Y})(X_i - \bar{X}) \Rightarrow cov(X,Y) = cov(Y,X) \tag{5.4}$$

5.2.2 Matriz de Autovetores

Um vetor v é dito autovetor de uma matriz quadrada M se $M \cdot v$ resulta em um múltiplo de v, ou seja, em λv (multiplicação de um escalar pelo vetor). Assim, λ é chamado autovalor de M associado ao autovetor v. No contexto da PCA, "os autovalores representam o comprimento dos eixos e são medidos em unidade de variância." (BERGAMO, 2006), e os autovetores representam suas direções. Os autovalores podem ser encontrados resolvendo a equação:

$$det(M - \lambda \cdot I) = 0 \tag{5.5}$$

sendo I a matriz identidade, M a matriz dada e λ o autovalor (VASCONCELOS, 2010). Por exemplo, se a matriz M tiver dimensão 2x2, a equação seria:

$$det \begin{pmatrix} m_{11} - \lambda & m_{12} \\ m_{21} & m_{22} - \lambda \end{pmatrix} = 0 \tag{5.6}$$

Após encontrados os valores de λ (ou seja, os autovalores), para o cálculo dos autovetores é necessária a resolução do sistema:

$$\begin{pmatrix} m_{11} - \lambda & m_{12} \\ m_{21} & m_{22} - \lambda \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{5.7}$$

sendo o vetor pelo qual a matriz é multiplicada denominado autovetor.

Em matrizes de dimensões maiores que 3x3, o processo para o cálculo dos autovalores e autovetores pode se tornar bem mais complicado e trabalhoso. Por isso, normalmente são usados métodos iterativos, como o de Jacobi, que realiza iterações buscando a aproximação do valor que se quer obter.

A técnica consiste em encontrar uma matriz M', que é dita matriz diagonalizada de M, a partir de transformações que aproximam todos os seus valores de zero, exceto a diagonal principal, que resulta em todos os autovalores de M (PRESS et al., 1992). A transformação aplicada é:

$$M' = P_{pq}^T \cdot M \cdot P_{pq} \tag{5.8}$$

sendo P_{pq} uma matriz com todos os elementos da diagonal principal iguais a um, exceto nas linhas p e q, onde os valores são o cosseno do ângulo de rotação (α) ao qual a matriz será submetida. Os valores que não estão na diagonal principal são todos iguais a zero, com exceção dos elementos adjacentes ao centro na diagonal secundária, que são $-\operatorname{sen}\alpha$ e $\operatorname{sen}\alpha$, resultando na matriz:

$$P_{pq} = \begin{pmatrix} 1 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \vdots & \ddots & \cdots & \cdots & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cos\alpha & \cdots & \operatorname{sen}\alpha & \cdots & \vdots \\ \vdots & \cdots & \vdots & 1 & \vdots & \cdots & \vdots \\ \vdots & \cdots & -\operatorname{sen}\alpha & \cdots & \cos\alpha & \cdots & \vdots \\ \vdots & \cdots & \cdots & \cdots & \cdots & \ddots & \vdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 1 \end{pmatrix} \tag{5.9}$$

Em seguida, são realizadas rotações seguindo os critérios de convergência, a qual, segundo Press et al. (1992), pode ser verificada por:

$$S = \sum_{r \neq s} | \, m_{rs} \, |^2 \tag{5.10}$$

Por fim, a matriz P_{pq} resultará em uma matriz com os autovetores de M.

5.2.3 Matriz de Componentes Principais

O último passo da PCA é a montagem da matriz com as componentes principais e é neste ponto que se reduz a dimensão do problema. É escolhida a quantidade de autovetores, que é a quantidade de variáveis com as quais se deseja representar o conjunto. Estes autovetores são os componentes principais, que retêm a maior parte da informação contida nos dados (SMITH, 2002). Escolhidos os autovetores, é montada uma matriz com os autovetores mais relevantes, classificados a partir dos autovalores, com a seguinte estrutura:

$$M_{Autovetores} = \begin{pmatrix} autovetor_1 & autovetor_2 & \cdots & autovetor_p \end{pmatrix}_{pxp} \tag{5.11}$$

Em seguida, deve-se montar a matriz de dados ajustados a partir dos dados obtidos na equação 5.2, assim estruturada:

$$M_{DadosAjustados} = \begin{pmatrix} Z_1^X & Z_1^Y & \cdots \\ Z_2^X & Z_2^Y & \cdots \\ \vdots & \vdots & \cdots \\ Z_n^X & Z_n^Y & \cdots \end{pmatrix}_{nxp} \tag{5.12}$$

E finalmente, calculam-se os dados finais, multiplicando-se a matriz de autovetores pela matriz de dados ajustados, chegando-se à matriz de PCA:

$$M_{PCA} = M_{DadosAjustados} \cdot M_{Autovetores} \tag{5.13}$$

5.3 Considerações Finais

A Análise de Componentes Principais é um método estatístico muito importante para se analisar conjuntos multivariados. Pelo fato de lidar com matrizes, muitas vezes, de grandes dimensões, esse método realiza um alto número de operações, demandando alto poder computacional. Por isso, é interessante aplicar técnicas de paralelismo, visando à redução do tempo de computação para se realizarem os cálculos das componentes principais, e avaliar o comportamento e o impacto dessas técnicas sobre o seu algoritmo.

Capítulo 6

TESTES E IMPLEMENTAÇÃO

6.1 Introdução

A implementação do algoritmo de PCA foi escolhida para a realização de análises e comparação de desempenho entre códigos paralelo e sequencial. Este programa é um módulo de uma implementação feita por Bergamo (2006), que analisa entradas que são captadas por sensores de uma língua eletrônica, e identifica a substância lida por estes sensores.

Em sua implementação, Bergamo (2006) utiliza a PCA para reconhecimento de padrões e posterior análise destes valores para identificação da substância presente na amostra. É feita a leitura de um arquivo que contém todas as amostras lidas, correspondendo ao número de sensores utilizados. São feitas diversas leituras, portanto, o arquivo contém uma lista, com *m* leituras e *n* amostras, desta forma, podendo ser interpretada como uma matriz *mxn*. Ao final, é gerado um novo arquivo, com os valores encontrados após a realização do procedimento da PCA.

Como foi visto no Capítulo 5, um dos passos deste procedimento envolve a multiplicação de matrizes, algo que pode se tornar muito custoso computacionalmente se as matrizes tiverem dimensões grandes. Por conta disso, é interessante a paralelização do código de PCA, visando à diminuição do seu tempo de execução e aumento de desempenho.

6.2 Abordagem Usada nas Implementações

O processo de paralelização de um algoritmo nem sempre é algo trivial. É necessário fazer uma análise do comportamento do programa sequencial, e verificar trechos ou módulos independentes, que são candidatos a serem paralelizados. A execução desses trechos separadamente não pode influenciar no resultado final que se espera do programa.

Considerando isto, uma estratégia bastante popular no meio da computação paralela é a chamada "Mestre-Escravo", onde um dos processos em execução é denominado "Mestre", e é responsável pelo gerenciamento e submissão de tarefas aos demais processos, que são ditos "Escravos".

Esta estratégia é típica do modelo de programação apresentado neste trabalho, a troca de mensagens. Nele, um processo fica responsável pela entrada/saída de dados e encaminhamento de tarefas aos demais, por meio das mensagens *send* e *recv* do MPI, como mostra a Figura 6.

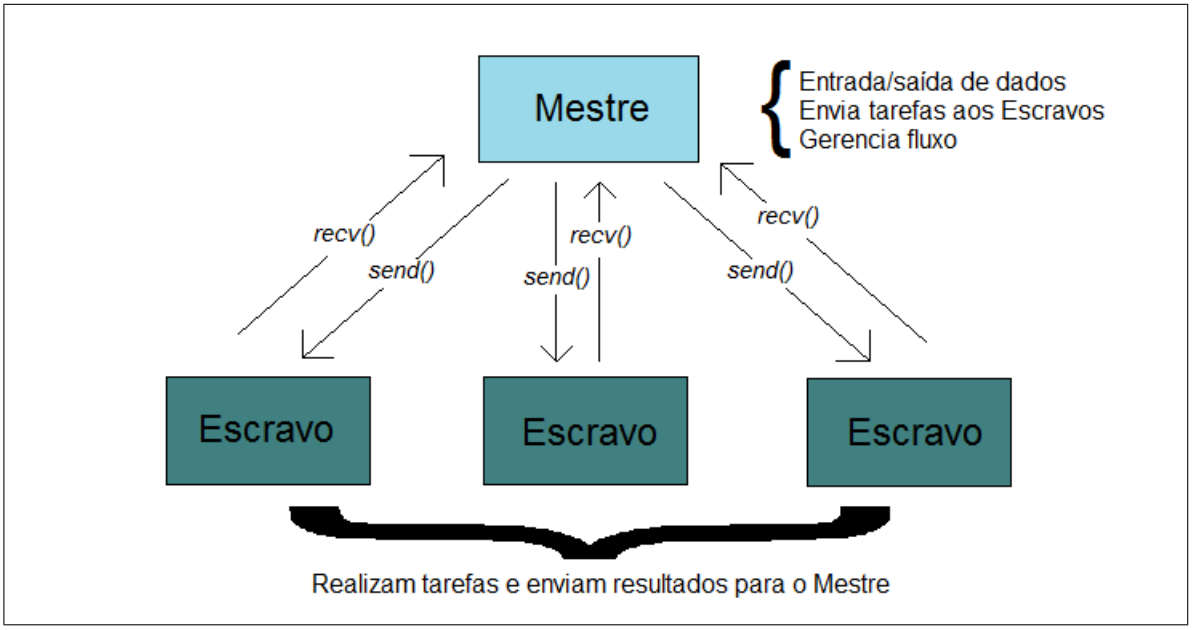


Figura 6: Estratégia Mestre-Escravo.

Além disso, esta estratégia é adequada à arquitetura do *cluster* Beowulf, que é o ambiente utilizado para a execução do programa implementado neste trabalho, e é eficiente e adequada ao problema apresentado, que envolve grande quantidade de iterações que podem ser divididas entre vários processos sem resultar em resultados diferentes dos esperados. Desta forma, à medida que se aumenta o número de processadores empregados na resolução do problema, a eficiência também aumenta (NASCIMENTO, 2010), pois cada processo precisa lidar com menos cálculos.

6.3 Ambiente de Execução

O ambiente utilizado para a execução do programa implementado neste trabalho foi o *cluster* "ericclaptonCS", pertencente ao Departamento de Matemática, Estatística e Computação da Faculdade de Ciências e Tecnologia – Unesp, localizado no endereço eletrônico 171d.fct.unesp.br.

O *cluster* possui um nó mestre, que gerencia seu funcionamento, e dez nós escravos, que realizam as tarefas, e é configurado da seguinte forma:

- Sistema operacional Debian, kernel 2.6.32-5-amd64;
- As contas e perfis de usuários são gerenciadas exclusivamente pelo nó mestre, por meio do sistema NIS/NFS;
- A comunicação entre os nós é feita por ssh;
- O monitoramento é feito pelo GANGLIA;
- Os processos são submetidos pelo TORQUE, via script PBS;
- A implementação da interface de troca de mensagens é a MPICH2;

O *hardware* das máquinas tem as seguintes especificações:

- Processador Intel Core 2 Quad 2.66GHz;
- Memória DDR2 de 2GB em cinco nós, e 4GB nos demais;
- Disco rígido com capacidade de 600GB;
- Placa de vídeo Nvidia 9200 com CUDA.

6.4 Testes Iniciais

O problema de PCA envolve, entre outros procedimentos, a manipulação e multiplicação de matrizes. Por este motivo, antes de o programa final ser implementado, foram realizados testes e análise de desempenho com multiplicações de matrizes de diferentes tamanhos, com algoritmos sequenciais e paralelos, os últimos tanto usando MPI quanto o pacote PETSc.

No código escrito com MPI puro, é preciso especificar os trechos a serem executados em paralelo, tornando o processo um pouco mais trabalhoso, enquanto que com o uso do PETSc, este procedimento é simplificado, uma vez que o pacote oferece funções que realizam os cálculos sem que o programador precise especificar o paralelismo.

A Figura 7 ilustra o comportamento do algoritmo em MPI. A matriz é dividida entre os processos, e cada processo fica responsável por calcular um trecho da matriz. Neste caso, cada processo está encarregado de multiplicar uma linha, mas este número varia de acordo com o número de processos empregados, distribuindo-se as linhas igualmente a todos eles.

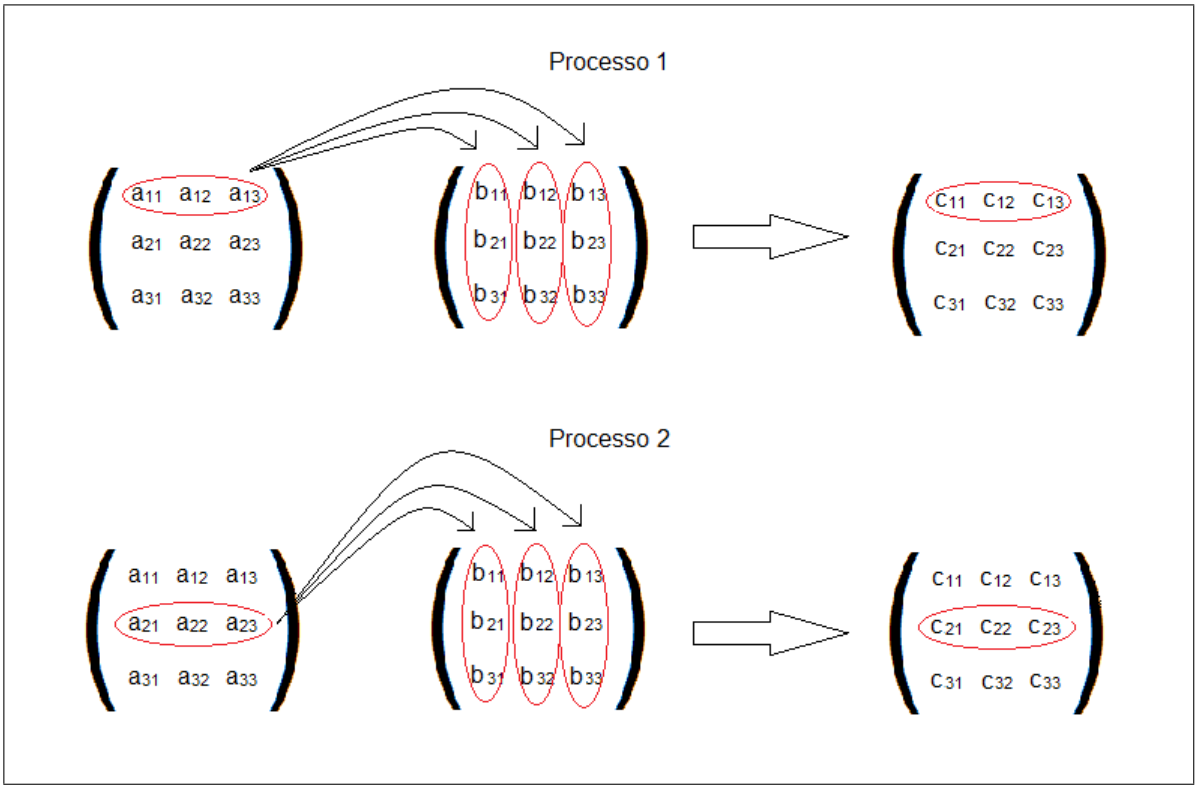


Figura 7: Comportamento do código de multiplicação de matrizes no MPI.

Já com o PETSc, este trabalho de distribuição é feito automaticamente, pelas ferramentas do pacote. Ele funciona de maneira que não é necessário saber como a linguagem trata este paralelismo. A seguir é apresentado o código para a multiplicação de matrizes com MPI.

```
1 if(taskid == MASTER) {
2  for (dest=1; dest<=numworkers; dest++)
3 {
4   rows = (dest <= extra) ? averow+1 : averow;
5   MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
6   MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
```

```
7     MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,
          MPI_COMM_WORLD);
8     MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
9     offset = offset + rows;
10   }
11   for (i=1; i<=numworkers; i++)
12  {
13    source = i;
14    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
15    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
16    MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
          MPI_COMM_WORLD, &status);
17   }
18  }
19  if(taskid > MASTER) {
20   MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
21   MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
22   MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &
         status);
23   MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status
         );
24   for (k=0; k<NCB; k++)
25   for (i=0; i<rows; i++)
26  {
27    c[i][k] = 0.0;
28    for (j=0; j<NCA; j++)
29      c[i][k] = c[i][k] + a[i][j] * b[j][k];
30    }
31   MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
32   MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
33   MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
34  }
```

Nota-se que tanto a distribuição das tarefas entre os processos como o envio dos resultados são indicados explicitamente pelo programador. Por outro lado, o mesmo cálculo, com o pacote PETSc, utiliza apenas a instrução:

```
MatMatMultNumeric(A,B,C);
```

Portanto, pode-se perceber que a simplicidade e praticidade do uso do PETSc é muito maior do que o MPI puro.

6.5 Análise do Desempenho dos Testes

Foram realizados testes com códigos sequenciais, paralelos com MPI puro e paralelos com o auxílio do PETSc, executando multiplicações de matrizes de dimensões 500x500, 1000x1000, 3000x3000 e 5000x5000, usando quatro nós do *cluster*. A Tabela 1 mostra os tempos de execução parcial (referente só à parte do código em que a multiplicação ocorre) e total de cada programa.

Tabela 1: Tempo de execução dos algoritmos.

Dimensão da Matriz	**Tempo de Execução (s)**					
	Sequencial		**MPI**		**PETSc**	
	Parcial	**Total**	**Parcial**	**Total**	**Parcial**	**Total**
500x500	1.2	1.2	0.48	0.48	0.15	0.61
1000x1000	9.82	9.84	3.47	3.53	1.03	7.42
3000x3000	286.69	288.5	95.22	95.57	26.63	983.02
5000x5000	1338.45	1339.66	453.26	453.9	122.18	6238.81

Observando o tempo de execução dos programas, nota-se que o programa feito com MPI puro obteve um ganho de desempenho relevante, pois seus tempos de execução parcial e total reduziram bastante. O código feito com o PETSc também obteve melhores resultados quanto à multiplicação, no entanto, o tempo de execução total foi bastante superior ao programa sequencial. Isto ocorre porque o processo de criação das estruturas de dados que ele usa demanda um processamento maior. As Figuras 8 e 9 apresentam os gráficos comparando a eficiência e o *speedup* entre os códigos com MPI puro e com PETSc.

É possível perceber que, com relação à operação de multiplicação em si, ambos os códigos paralelos obtiveram melhor desempenho do que o sequencial, mas a eficiência do PETSc foi superior à do MPI. O *speedup*, ou seja, o quão mais rápido é o algoritmo paralelo, também obteve bons níveis nos dois tipos de aplicação, ultrapassando, no caso do PETSc, a taxa de *speedup* ideal, que é igual ao número de processadores empregados (nesse caso, quatro). Isso é possível porque, por ser um pacote totalmente dedicado a operações matemáticas, ele utiliza métodos de resolução mais avançados do que o convecional, que foi empregado nos programas sequencial e com MPI sem PETSc.

Com base nestes dados, conclui-se que, quando se quer obter melhor desempenho e menor tempo de execução, tanto o PETSc quanto o MPI puro podem ser boas soluções. No entanto, o PETSc demanda mais processamento na inicialização de suas estruturas da dados, por isso ele é mais indicado para aplicações mais complexas e elaboradas, para que o tempo gasto com

a inicialização de suas estruturas compense o ganho sobre as operações em si. Prova disto é que à medida que o número de elementos da matriz aumenta, aumenta também o *speedup* da aplicação desenvolvida com o PETSc.

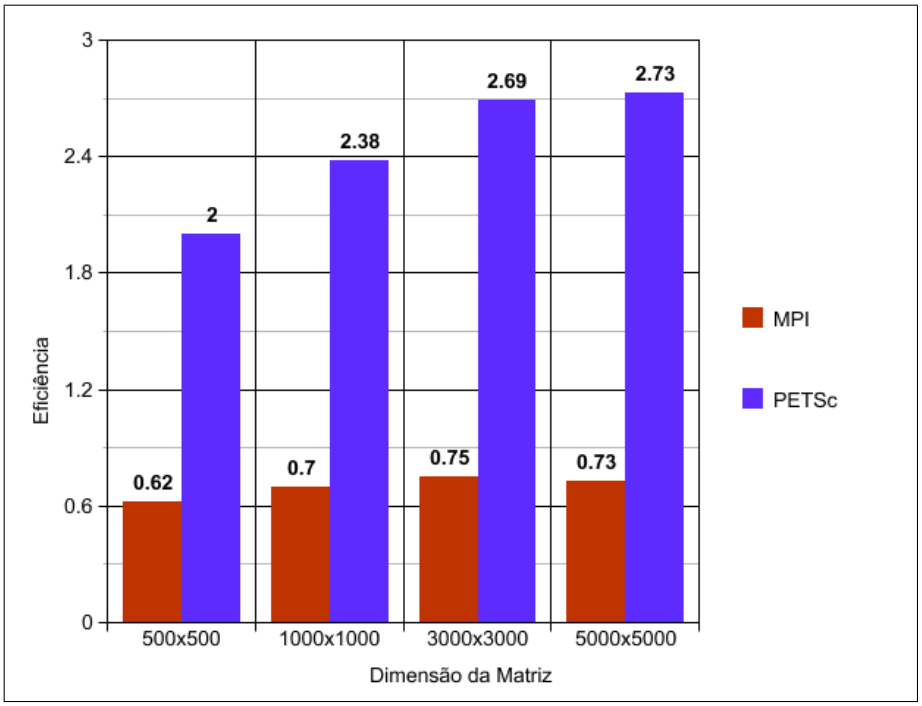


Figura 8: Gráfico comparativo de eficiência entre MPI e PETSc.

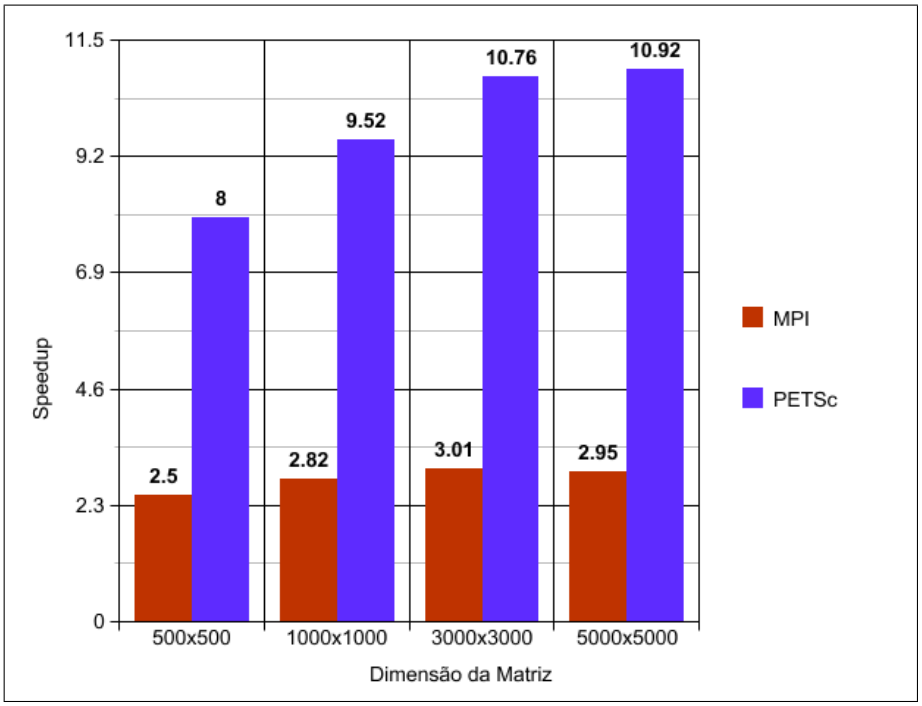


Figura 9: Gráfico comparativo de *speedup* entre MPI e PETSc.

6.6 Paralelismo na Análise de Componentes Principais

Analisando o algoritmo para o cálculo das componentes principais, nota-se que a maioria dos passos tem comportamento estritamente sequencial, ou seja, é totalmente necessário que sejam executados nesta ordem, e cada passo não pode ser dividido e processado separadamente, exceto pelo trecho em que as matrizes de autovetores e a de dados ajustados são multiplicadas.

Este processo de multiplicação de matrizes pode ser distribuído entre vários nós, na tentativa de se obter ganho no desempenho, e consequentemente, diminuição do tempo de execução. Como a quantidade de operações depende do número de elementos das matrizes, esta abordagem se torna interessante conforme este número aumenta. Desta forma, em casos em que se deseja calcular as componentes principais de milhões de elementos, este procedimento pode proporcionar bons resultados.

6.6.1 Desenvolvimento e Funcionamento da Aplicação

O programa que faz o cálculo da PCA foi escrito em linguagem C, e, considerando os resultados obtidos com os testes iniciais, foi escolhido o MPI puro para a implementação, ao invés do pacote PETSc, pois a complexidade das operações feitas na PCA não compensam o tempo gasto com a inicialização das estruturas de dados usada pelo PETSc.

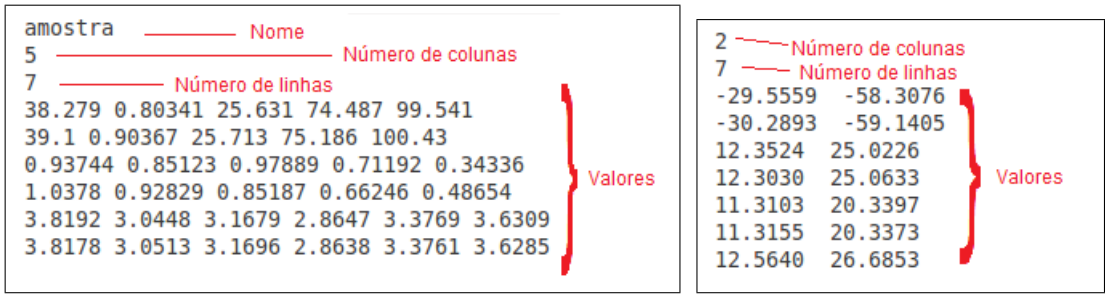
Para o funcionamento do programa, deve-se definir um arquivo texto de entrada com o seguinte leiaute: o nome da substância ou do teste na primeira linha; o número de colunas (n) na segunda, onde cada coluna representa um sensor usado para a leitura da substância pela língua eletrônica; e o número de linhas (m), representando a quantidade de vezes que a substância foi lida, na terceira linha. Em seguida, em cada linha são colocados os valores de cada leitura da língua, separando as colunas por um espaço simples.

Definido o arquivo, o programa faz a leitura dos dados, e os armazena em um *array* bidimensional do tipo *float*, de dimensão *mxn*, alocado dinamicamente no *heap*. Desta forma, é possível se trabalhar com grandes quantidades de elementos. Posteriormente, são calculadas as médias de cada coluna, somando os elementos de todas as linhas da coluna e dividindo pelo número de linhas. Este valor é subtraído de cada elemento da coluna correspondente, formando a Matriz de Dados Ajustados.

Em seguida, é calculada a matriz de covariância de dimensão *nxn*, conforme descrito na Subseção **??**, e depois a Matriz de Autovetores, também de dimensão *nxn*, usando o método de

Jacobi. Então, se multiplica a Matriz de Dados Ajustados pela Matriz de Autovetores, e obtém-se a Matriz da PCA, de dimensão *mxn*. Finalmente, são extraídas duas componentes de cada medição, ou seja, duas colunas desta matriz, e gravadas em outro arquivo texto, com o mesmo leiaute do arquivo lido, porém, sem a primeira linha com o nome da substância.

As Figuras 10(a) e 10(b) apresentam exemplos dos arquivos de entrada e saída, respectivamente. O arquivo de saída exibe duas componentes extraídas dos dados iniciais, simplificando o processo de análise dos dados. A Figura 11 exibe uma maneira de se visualizarem os dados, utilizando um gráfico de dispersão.



(a) Arquivo de entrada. (b) Arquivo de saída.

Figura 10: Exemplos de arquivos manipulados no programa.

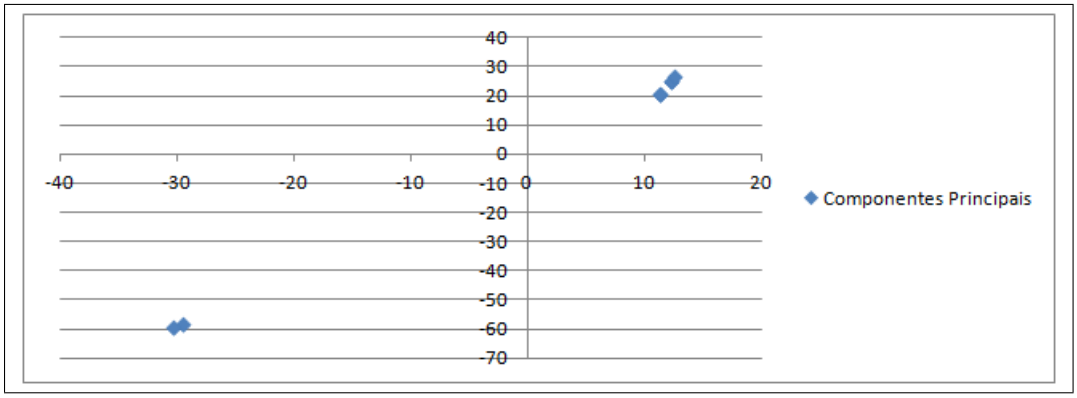


Figura 11: Gráfico de dispersão dos valores obtidos pela PCA.

6.6.2 Resultados

Foram implementadas duas versões do programa: a paralela, descrita na Subseção 6.6.1, e uma de comportamento sequencial, que serviu de base para comparação com o desempenho

da aplicação paralela. Em ambas foram testados diferentes arquivos, contendo diferentes quantidades de medições (200000, 400000, 1200000 e 2400000), e usando seis sensores de leitura (resultando em matrizes de 1200000, 2400000, 7200000 e 14400000 elementos).

Conforme a Tabela 2, o resultado obtido pela aplicação paralela não foi tão satisfatório. Isto se deve ao fato de o tempo de comunicação anular o ganho no tempo de processamento, além disso, como o número de sensores é pequeno, a Matriz de Autovetores tem dimensões pequenas, resultando em um número não muito elevado de operações. Por este motivo, o tempo de processamento, somado ao tempo de comunicação, acaba sendo praticamente igual ao tempo de processamento da aplicação sequencial. Por isso, não há ganho considerável de desempenho nestes casos.

Tabela 2: Tempo de execução dos algoritmos.

Número de Linhas	**Tempo de Execução (s)**							
	Sequencial		**MPI (4 nós)**		**MPI (9 nós)**		**MPI (1 nó, 4 proc.)**	
	Parcial	**Total**	**Parcial**	**Total**	**Parcial**	**Total**	**Parcial**	**Total**
200000	0.08	1.84	0.10	1.83	0.08	1.90	0.04	1.76
400000	0.15	3.72	0.17	3.69	0.16	3.77	0.07	3.58
1200000	0.48	11.22	0.50	11.03	0.51	11.57	0.20	10.78
2400000	0.94	22.11	0.99	21.75	1.00	22.38	0.41	21.19
4800000	1.89	43.88	1.99	42.89	1.97	43.21	0.80	41.77

Analisando o algoritmo da multiplicação de matrizes na Figura 12, chamando de PA o programa que multiplica duas matrizes de 500x500, e PB o que multiplica uma matriz de 2400000x6 por outra de 6x6, nota-se que, o programa PA realiza $500x500x500 = 125$ milhões de iterações enquanto PB realiza $6x2400000x6 = 86.4$ milhões. No entanto, as Tabelas 1 e 2 mostram que, usando quatro nós do *cluster*, o tempo de execução de PA (0.48s) é inferior ao de PB (0.99s), apesar deste realizar menos iterações. Porém, observando-se os dados do algoritmo sequencial, o programa PA, de fato, leva mais tempo para ser executado (1.2s contra 0.94s de PB).

Como se pode observar no código de multiplicação de matrizes com MPI, mostrado na Seção 6.4, o processo mestre envia para os escravos um número de linhas igual à quantidade total de linhas dividida pelo número de processos escravos. Desta forma, usando quatro processos (um mestre e três escravos), no programa PA são enviadas 166 linhas para cada processo, que realiza $500x166x500 = 41.5$ milhões de iterações; no programa PB, são enviadas 800 mil linhas para cada processo, resultando em $6x800000x6 = 28.8$ milhões de iterações. Ou seja, cada processo, de fato, faz a computação dos dados mais rapidamente, pois tem menos iterações (e operações) para realizar.

```
292 for (k=0; k<ColunasB; k++)
293                 for (i=0; i<LinhasA; i++)
294                 {
295                    c[i][k] = 0.0;
296                    for (j=0; j<ColunasA; j++)
297                       c[i][k] = c[i][k] + a[i][j] * b[j][k];
298                 }
299
```

Figura 12: Algoritmo da multiplicação de matrizes.

Por outro lado, ainda de acordo com o código da Seção 6.4, a quantidade de elementos enviados para os processos escravos é igual àquela quantidade de linhas (total dividido pelo número de processos) multiplicada pelo número de colunas da segunda matriz. Isto significa que o programa PA envia $166x500 = 83$ mil elementos do tipo *double* (648.43 *kilobytes*) para os demais processos, enquanto PB envia $800000x6 = 4.8$ milhões de elementos do mesmo tipo (resultando em 36.62 *megabytes*). Panda (2011) realizou estudos quanto ao tempo de comunicação das mensagens em relação ao tamanho delas, e constatou que o tamanho da mensagem exerce grande influência no tempo de comunicação do MPI. Os mesmos resultados foram observados neste trabalho, como indica a Figura 13.

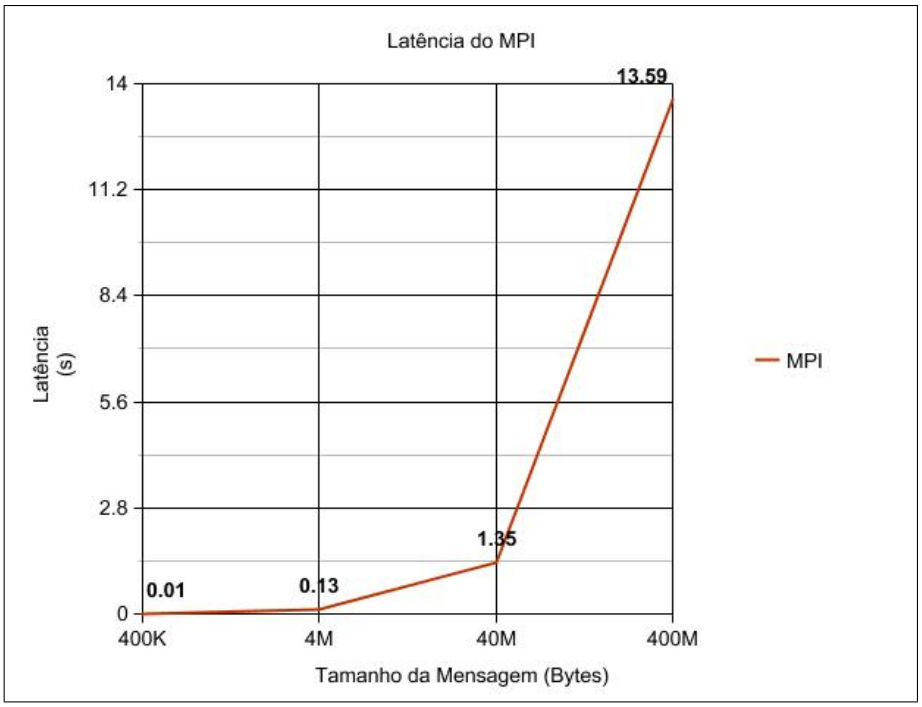


Figura 13: Latência do MPI x Tamanho da Mensagem.

Portanto, é possível afirmar que, novamente, o atraso na obtenção dos resultados é justifi-

cado pelo tempo de comunicação. Outra prova disto pode ser observada na Tabela 2 e na Figura 14, onde nota-se que o mesmo programa PB, também com quatro processos, porém executado em apenas um nó, o que diminui o tempo de comunicação, uma vez que o acesso à memória é local, de fato leva menos tempo para ser executado do que sequencialmente ou distribuído entre outros nós.

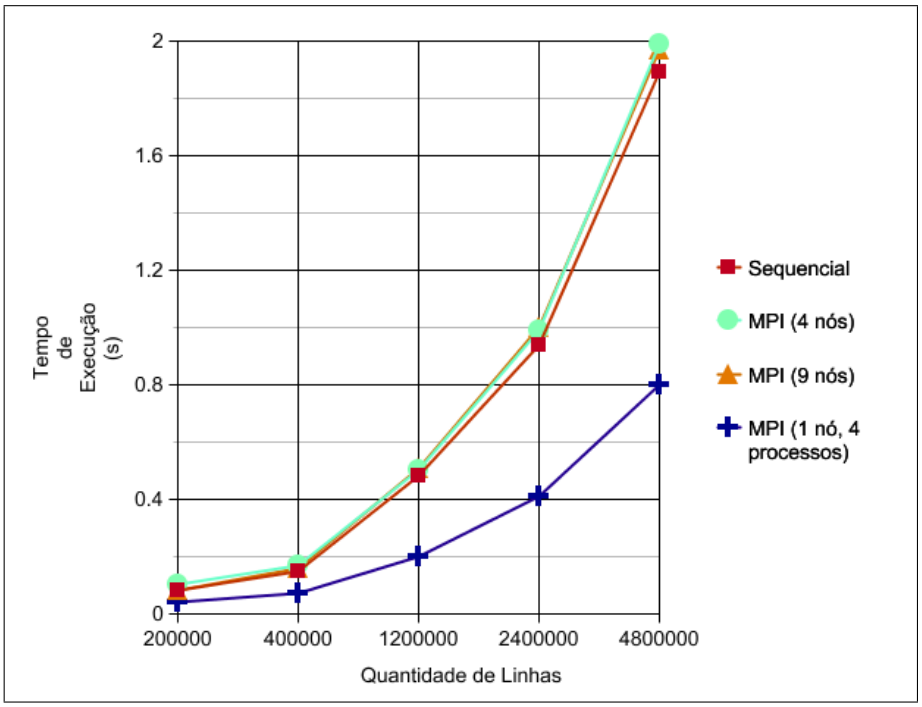


Figura 14: Tempo de execução dos algoritmos.

Capítulo 7

CONCLUSÕES

A programação paralela se tornou uma das técnicas mais utilizadas para se obter melhor desempenho na execução de algoritmos e tirar melhor proveito do poder computacional disponível, principalmente depois que começaram a ser desenvolvidos processadores e arquiteturas *multicore* e o uso de *clusters* passou a ser mais comum.

A popularização da programação paralela é reflexo de sua efetividade em conseguir bons resultados. Apesar disso, é preciso entender que esta abordagem não é válida para qualquer tipo de problema, e, mais importante, encontrar a melhor maneira de aplicar o paralelismo nos problemas em que é possível, e levar em consideração o quanto vale a pena aplicá-lo. Por isso é fundamental o estudo de formas de se avaliar o desempenho dos algoritmos, extraindo métricas, analisando resultados e estudando o comportamento dos algoritmos, das máquinas e das ferramentas usadas.

Existem diversos modelos, estratégias e padrões para se aplicar o paralelismo, e o MPI é apenas um deles, que oferece uma variedade de funções que permitem a divisão das tarefas em diferentes processos, e a comunicação entre eles por meio de envio e recebimento de mensagens. O MPI é amplamente usado pela comunidade que desenvolve aplicações paralelas, e algumas ferramentas, como o pacote PETSc, por exemplo, são feitas utilizando o MPI como base, usando-o de maneira implícita. O PETSc é uma ferramenta muito interessante para o desenvolvimento de aplicações científicas, pois ele oferece dezenas de funções que resolvem problemas matemáticos paralelamente.

Com base nos resultados obtidos neste trabalho, foi possível constatar que nem sempre o paralelismo é vantajoso, mesmo em casos em que o algoritmo é paralelizável. Foram realizados testes com o algoritmo de multiplicação de matrizes, para se avaliar o desempenho do MPI puro e do pacote PETSc. Os resultados iniciais foram satisfatórios, revelando que, em

ambos os casos, a aplicação obteve melhor desempenho na execução do algoritmo. Por outro lado, foi possível perceber que o PETSc demanda muito mais tempo no processo de criação de suas estruturas de dados, apesar de realizar as operações mais rapidamente. O fato de tais estruturas serem especialmente projetadas para facilitar a manipulação e reduzir o tempo de processamento de seus dados faz com que o custo de inicialização seja maior. Este é um típico caso em que se deve analisar o quanto vale a pena aplicar as funções do PETSc. A partir do momento que o processamento dos dados sequencialmente se torna tão custoso quanto o processo de criação do ambiente do PETSc, seu uso passa a ser vantajoso.

Na multiplicação de matrizes quadradas, o MPI obteve um ganho de desempenho relevante, diminuindo o tempo de execução do algoritmo para um terço do tempo gasto pelo programa de execução sequencial. No entanto, ele não obteve os mesmos resultados quando aplicados no programa que calcula a PCA, apesar de a operação analisada ser, também, a multiplicação de duas matrizes. Isto acontece porque a melhora do desempenho depende diretamente das dimensões das matrizes em questão. Por isso deve-se levar em conta a natureza do problema cujo cálculo da solução se quer paralelizar.

Ainda assim, vale salientar que o algoritmo paralelo só não obteve melhor desempenho por causa do tempo de comunicação entre os nós do *cluster*, pois executando o mesmo algoritmo localmente, dividido em diversos processos, ele obteve, de fato, melhores resultados que o algoritmo sequencial. Portanto, o paralelismo ainda é aplicável, desde que se planeje usar um sistema com latência baixa, de preferência com memória compartilhada.

Finalmente, pode-se concluir que, para que haja um ganho relevante neste processo, é necessário o uso de um grande volume de dados, para que o paralelismo mostre efeitos notáveis, até porque, pela natureza do algoritmo de PCA, as instruções executadas em paralelo são mínimas, e, portanto, não exercem tanta influência no tempo total de execução do programa.

Como trabalho futuro, é interessante o estudo de técnicas e estratégias que diminuam este atraso na comunicação entre os processos, a fim de se extrair o máximo que a execução de processos em paralelo pode oferecer.

Referências Bibliográficas

ABARUCHI, A. *Abaruchi*. 2010. Disponível em: <http://www.abaruchi.com/programacao/mpi/mm.html>. Acesso em: 20 dez. 2011.

BAKER, M. (Ed.). *Cluster computing white paper*. University of Portsmouth, Reino Unido, v. 2, 28 dez. 2000.

BALAY, S. et al. *PETSc users manual*, v. 3.1, 2010. Disponível em: <http://www.mcs.anl.gov/petsc/petsc-as/documentation/index.html>. Acesso em: 16 mai. 2011.

BEOWULF Project. 2007. Disponível em: <http://www.beowulf.org>. Acesso em: 4 nov. 2011.

BERGAMO, B. B. *Sistema de reconhecimento de padrões aplicado à língua eletrônica*. 2006. Monografia (Bacharel em Ciência da Computação) — Universidade Estadual Paulista, Presidente Prudente.

CARINGI, A. M. *Escalonamento estático de processos de aplicações paralelas MPI em máquinas agregadas heterogêneas com auxílio de históricos de monitoração*. 2006. Dissertação (Mestrado em Ciência da Computação) — Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre.

CARVALHO, V. F. P. *Otimização de um cluster de alto desempenho para o uso do programa PGENESIS em simulações biologicamente plausíveis em larga-escala de sistemas neurais*. 2007. Dissertação (Mestrado em Ciências) — Universidade de São Paulo, Ribeirão Preto.

IBTA. *InfiniBand trade association*. 2011. Disponível em: <http://www.infinibandta.org/index.php>. Acesso em: 20 dez. 2011.

LAINE, J. M. *Desenvolvimento de modelos de predição de desempenho de programas paralelos MPI*. 2003. Dissertação (Mestrado em Engenharia) — Escola Politécnica da Universidade de São Paulo, São Paulo.

MPI FORUM. *A message-passing interface standard*. 2009. Disponível em: <http://www.mpi-forum.org/docs/docs.html>.

NASCIMENTO, V. S. *Desenvolvimento de aplicações paralelas distribuídas com OpenMP e MPI*. 2010. Monografia (Bacharel em Ciência da Computação) — Universidade Estadual Paulista, Presidente Prudente.

PANDA, D. K. *MVAPICH: MPI over infiniband*. 2011. Disponível em: <http://mvapich.cse.ohio-state.edu/performance/mvapich2/em64t/MVAPICH2-em64t-gen2-ConnectX-QDR.shtml>. Acesso em: 10 dez. 2011.

PASSOS, L. B. C. *Avaliação de desempenho de método para a resolução da evolução temporal de sistemas auto-gravitantes em dois paradigmas de programação paralela: troca de mensagens e memória compartilhada*. 2006. Dissertação (Mestrado em Informática) — Universidade de Brasília, Brasília.

PRESS, W. H. et al. *Numerical recipes in C*: the art of scientific computing. 2. ed. New York, NY, USA: Cambridge University Press, 1992.

SANTOS, L. A. F. *Desenvolvimento de algoritmos paralelos para simulador de multiprocessadores superescalares*. 2005. Dissertação (Mestrado em Ciência da Computação) — Universidade Estadual de Maringá, Maringá.

SILVA, L. N. *Modelo híbrido de programação paralela para uma aplicação de eslasticidade linear baseada no método dos elementos finitos*. 2006. Dissertação (Mestrado em Informática) — Universidade de Brasília, Brasília.

SMITH, L. I. *A tutorial on principal components analysis*, 2002. Disponível em: <http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf>. Acesso em: 26 out. 2011.

TOP500 Supercomputer Sites. 2011. Disponível em: <http://top500.org>. Acesso em: 4 nov. 2011.

VASCONCELOS, S. *Análise de componentes principais*, 2010. Disponível em: <www.ic.uff.br/∼aconci/PCA-ACP.pdf>. Acesso em: 15 nov. 2011.

Apêndice A

APÊNDICE A - Exemplo de Código MPI

Segue um exemplo de código em C usando MPI, executando uma multiplicação de matrizes, adaptado do código de Abaruchi (2010).

```
1  #include "mpi.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define NRA 1000                 /* numero de linhas da matriz A */
6  #define NCA 1000                 /* numero de colunas da matriz A */
7  #define NCB 1000                  /* numero de colunas da matriz B */
8  #define MASTER 0               /* id do primeiro processo */
9  #define FROM_MASTER 1          /* setando tipo de mensagem */
10 #define FROM_WORKER 2          /* setando tipo de mensagem */
11
12 int main (int argc, char *argv[])
13 {
14   int     numtasks,              /* numero de processos */
15           taskid,                /* identificador do processo */
16           numworkers,            /* n mero de processos trabalhando */
17           source,                /* id do processo remetente */
18           dest,                  /* id do processo destinat rio */
19           mtype,                 /* tipo de mensagem */
20           rows,                  /* linhas da matriz A para cada processo
                 */
21           averow, extra, offset, /* usado para determinar as linhas
                 enviadas
22                                     para cada processo trabalhador */
23           i, j, k, rc;
24   double          a[NRA][NCA],       /* matriz A a ser multiplicada */
25                   b[NCA][NCB],           /* matriz B a ser multiplicada */
```

```
26                  c[NRA][NCB];            /* matriz C resultante */
27    MPI_Status status;
28
29    MPI_Init(&argc,&argv);
30    MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
31    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
32
33    numworkers = numtasks-1;
34
35  /******** processo mestre *********/
36     if (taskid == MASTER) {
37        printf("Inicializando arrays...\n");
38        for (i=0; i<NRA; i++)
39           for (j=0; j<NCA; j++)
40              a[i][j]= i+j;
41        for (i=0; i<NCA; i++)
42           for (j=0; j<NCB; j++)
43              b[i][j]= i*j;
44
45        /* Enviando os valores da matriz para os processos trabalhadores */
46        averow = NRA/numworkers;
47        extra = NRA%numworkers;
48        offset = 0;
49        mtype = FROM_MASTER;
50        for (dest=1; dest<=numworkers; dest++)
51        {
52           rows = (dest <= extra) ? averow+1 : averow;
53           printf("Enviando %d linhas ao processo %d offset=%d\n",rows,dest
                ,
54  offset);
55           MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
56           MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
57           MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,
58  MPI_COMM_WORLD);
59           MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
60           offset = offset + rows;
61        }
62
63        /* Recebendo resultados dos processos trabalhadores */
64        mtype = FROM_WORKER;
65        for (i=1; i<=numworkers; i++) {
66           source = i;
67           MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
```

```
68 &status);
69          MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
70 &status);
71          MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
72                   MPI_COMM_WORLD, &status);
73          printf("Resultados recebidos do processo %d\n",source);
74       }
75
76       /* Imprimindo resultados */
77       printf("Resultado Matriz:\n");
78       for (i=0; i<NRA; i++) {
79          printf("\n");
80          for (j=0; j<NCB; j++)
81             printf("%6.2f   ", c[i][j]);
82       }
83    }
84
85 /********** processo trabalhador **********/
86    if (taskid > MASTER) {
87       mtype = FROM_MASTER;
88       MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD,
89 &status);
90       MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status)
             ;
91       MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD,
92 &status);
93       MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD,
94 &status);
95
96       for (k=0; k<NCB; k++)
97          for (i=0; i<rows; i++) {
98             c[i][k] = 0.0;
99             for (j=0; j<NCA; j++)
100               c[i][k] = c[i][k] + a[i][j] * b[j][k];
101         }
102      mtype = FROM_WORKER;
103      MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
104      MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
105      MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
106   }
107   MPI_Finalize();
108 }
```

Apêndice B

APÊNDICE B - Exemplo de Código PETSc

Segue abaixo um exemplo de código em C usando PETSc, análogo ao código do Apêndice A executando uma multiplicação de matrizes.

```
1  #include "petscsys.h"
2  #include "petscmat.h"
3  #define LINHASA 1000
4  #define COLUNASA 1000
5  #define LINHASB 1000
6  #define COLUNASB 1000
7
8  static char help[] = "Multiplicacao entre matriz A e matriz B, resultando
       em matriz C.\n\n";
9
10 int main (int argc, char **argv) {
11         PetscErrorCode ierr;
12         PetscMPIInt rank, size, p;
13         Mat A,B,C;
14         PetscInt i,j;
15         PetscScalar valores;
16         PetscLogDouble t1, t2;
17
18         ierr = PetscInitialize(&argc, &argv, PETSC_NULL, help); CHKERRQ(
               ierr);
19         ierr = PetscOptionsGetInt(PETSC_NULL,"-p",&p,PETSC_NULL); CHKERRQ
               (ierr);
20         ierr = MPI_Comm_size(PETSC_COMM_WORLD, &size); CHKERRQ(ierr);
21         ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank); CHKERRQ(ierr);
22         ierr = PetscPrintf(PETSC_COMM_WORLD, "Numero de processos = %d\n"
               , size); CHKERRQ(ierr); // COMM_WORLD -> apenas uma impressao
               na execucao toda
```

```
23          ierr = MPI_Barrier(PETSC_COMM_WORLD);CHKERRQ(ierr); // todos os
                processos esperam os outros chegarem ate este estado para
                continuarem a execucao
24          ierr = PetscPrintf(PETSC_COMM_WORLD, "\n\nMatriz 1000x1000\n\n");
25          ierr = MPI_Barrier(PETSC_COMM_WORLD);CHKERRQ(ierr);
26
27  /************* Criando matriz A, definindo seu tamanho, e atribuindo
        parametro passado na chamada da funcao ****/
28          ierr = MatCreate(PETSC_COMM_WORLD,&A); CHKERRQ(ierr);
29          ierr = MatSetSizes(A,PETSC_DECIDE,PETSC_DECIDE,LINHASA,COLUNASA);
                CHKERRQ(ierr);
30          ierr = MatSetFromOptions(A);CHKERRQ(ierr);
31
32  /************* Criando matriz B, definindo seu tamanho, e atribuindo
        parametro passado na chamada da funcao ****/
33          ierr = MatCreate(PETSC_COMM_WORLD,&B); CHKERRQ(ierr);
34          ierr = MatSetSizes(B,PETSC_DECIDE,PETSC_DECIDE,LINHASB,COLUNASB);
                CHKERRQ(ierr);
35          ierr = MatSetFromOptions(B); CHKERRQ(ierr);
36
37  /************* Criando matriz C, definindo seu tamanho, e atribuindo
        parametro passado na chamada da funcao ****/
38          ierr = MatCreate(PETSC_COMM_WORLD,&C); CHKERRQ(ierr);
39          ierr = MatSetSizes(C,PETSC_DECIDE,PETSC_DECIDE,LINHASA,COLUNASB);
40          ierr = MatSetFromOptions(C); CHKERRQ(ierr);
41
42  /************** Montando a matriz A **************/
43
44          ierr = PetscPrintf(PETSC_COMM_WORLD, "Inicializando matriz A...\n
                \n"); CHKERRQ(ierr);
45          for(i=0; i < LINHASA; i++) {
46                  for(j=0; j < COLUNASA; j++) {
47                          valores = i*j/5;        // dando valores para os
                                elementos de A
48
49                          ierr = MatSetValues(A,1,&i,1,&j,&valores,
                                INSERT_VALUES); CHKERRQ(ierr);
50                    // inserindo valores em 1 linha e "COLUNASA" colunas,
                          na linha i e na coluna j
51                  }
52          }
53          ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
54          ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
```

```
55
56 /************** Montando a matriz B *************/
57
58         ierr = PetscPrintf(PETSC_COMM_WORLD, "Inicializando matriz B...\n
               \n"); CHKERRQ(ierr);
59         for(i=0; i < LINHASB; i++) {
60                 for(j=0; j < COLUNASB; j++) {
61                         valores = i*j/9;         // dando valores para os
                            elementos de B
62
63                         ierr = MatSetValues(B,1,&i,1,&j,&valores,
                            INSERT_VALUES); CHKERRQ(ierr);
64                   // inserindo valores em 1 linha e "COLUNASB" colunas,
                        na linha i e na coluna j
65                 }
66         }
67         ierr = MatAssemblyBegin(B,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
68         ierr = MatAssemblyEnd(B,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
69
70 /************* Montando a matriz resultante C ****************/
71
72         ierr = PetscPrintf(PETSC_COMM_WORLD, "Multiplicando...\n\n");
               CHKERRQ(ierr);
73         ierr = MatMatMult(A,B,MAT_INITIAL_MATRIX,PETSC_DEFAULT,&C);
74         ierr = MatAssemblyBegin(C,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
75         ierr = MatAssemblyEnd(C,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
76
77         ierr = MatDestroy(A); CHKERRQ(ierr);
78         ierr = MatDestroy(B); CHKERRQ(ierr);
79         ierr = MatDestroy(C); CHKERRQ(ierr);
80
81         ierr = PetscFinalize(); CHKERRQ(ierr);
82
83         return 0;
84 }
```

Apêndice C

APÊNDICE C - Tutorial para uso do PETSc

C.1 Introdução

O PETSc é um pacote de estruturas de dados e rotinas para implementação de aplicações paralelas, e oferece ferramentas para resolução de equações lineares e não-lineares, podendo ser usado nas linguagens C, C++, Fortran, e recentemente, Python.

Ele foi desenvolvido para auxiliar no desenvolvimento de aplicações paralelas, tratando este paralelismo internamente, utilizando o padrão MPI para troca de mensagens. Por este motivo, é necessário que o MPI esteja instalado na máquina ou *cluster* que executará as aplicações que contêm elementos do PETSc.

A vantagem de se usar o PETSc ao invés do MPI puro é que muitas operações matemáticas já estão implementadas de forma paralela nas rotinas do PETSc, facilitando a escrita do código, uma vez que não é preciso explicitar o paralelismo.

Neste documento será explicado como instalar, configurar, construir e executar uma aplicação com PETSc, em uma máquina com o sistema operacional Linux.

C.2 Configurando o Ambiente

C.2.1 *Download*

O PETSc encontra-se disponível para *download* em sua página oficial, nas versões completa ou *lite* (sem documentação, que pode ser acessada *online*), no endereço http://www.mcs.anl.gov/petsc/petsc-as/download/index.html. É recomendável baixar a versão completa e mais recente, pois além da documentação, ela inclui as últimas atualizações e correções.

Atualmente, o PETSc encontra-se na versão 3.1, e o nome do arquivo usado para a elaboração deste guia é petsc-3.1-p0.tar.gz.

C.2.2 Instalação

Caso o MPI ainda não esteja instalado na máquina, é necessário instalá-lo. Uma das distribuições é o OpenMPI, e para instalá-lo, execute o seguinte comando, como usuário *root*:

```
apt-get install openmpi-bin
```

Com o MPI instalado, copie o arquivo que foi baixado do site (petsc-3.1-p0.tar.gz) para o diretório que deseja instalar o PETSc. Abra o *shell* do Linux, acesse o diretório em que se encontra o arquivo, e descompacte-o, com o comando:

```
gunzip -c petsc-3.1-p0.tar.gz | tar -xof
```

Agora, é necessário definir as variáveis de ambiente. Acesse o diretório que acabara de ser extraído (cd petsc-3.1-p0), e atribua este diretório como o diretório do PETSc, e defina a arquitetura a ser usada (exemplo: Linux), da seguinte forma:

```
PETSC_DIR=$PWD; export PETSC_DIR
PETSC_ARCH=linux-gnu-c-debug; export PETSC_ARCH
```

Em seguida, basta executar o arquivo de configuração do PETSc, com as opções para definir o compilador C e o compilador Fortran, baixar o pacote BLAS-LAPACK, que inclui bibliotecas algébricas, e baixar o MPI.

```
./config/configure.py --with-cc=mpicc --with-fc=0 --download-c-blas-
lapack=1 --download-mpich=1
```

Logicamente, é necessário que estejam previamente instalados o gcc e o gfortran. Por último, devem-se executar os arquivos de teste:

```
make all
make test
```

C.3 Escrevendo Códigos com PETSc

Um código em C utilizando PETSc não difere de um código tradicional. Ele apenas precisa incluir suas bibliotecas, e suas funções já podem ser usadas conforme especificadas no manual, que está disponível em: http://www.mcs.anl.gov/petsc/petsc-as/documentation/index.html#Manual.

Não é necessário tratar explicitamente a divisão de execução entre processos no código, pois o PETSc trata isso internamente. Da mesma forma, quando se inicia o PETSc, é iniciado também o MPI, e quando ele é finalizado, o MPI também o é.

Para se escrever um código com PETSc, é preciso chamar a função *PetscInitialize*. A partir daí, é possível usar outras funções PETSc. Ao fim do código, é preciso chamar *PetscFinalize* para finalizar sua execução.

Segue abaixo um exemplo básico de código usando PETSc:

```
1
2  // Nome do arquivo: helloworld.c
3
4  static char help[] = "Synchronized printing.\n\n";
5
6  #include "petscsys.h" //incluindo uma biblioteca do PETSc
7
8
9  int main(int argc,char **argv)
10 {
11   PetscErrorCode ierr;
12   PetscMPIInt    rank,size;
13
```

```
14    ierr = PetscInitialize(&argc,&argv,PETSC_NULL,help);
15    CHKERRQ(ierr); //inicializando o PETSc
16
17    ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);
18    CHKERRQ(ierr);
19  //funcoes do MPI tambem podem ser usadas
20    ierr = MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
21    CHKERRQ(ierr);
22
23  /* As funcoes abaixos sao executadas por cada processo */
24    ierr = PetscPrintf(PETSC_COMM_WORLD,"Quantidade de
25                  processos = %d, rank = %d\n",size,rank);
26
27    CHKERRQ(ierr);
28
29    ierr = PetscSynchronizedPrintf(PETSC_COMM_WORLD,
30                      "[%d] Hello World.\n",rank);
31    CHKERRQ(ierr);
32    ierr = PetscSynchronizedFlush(PETSC_COMM_WORLD);
33    CHKERRQ(ierr);
34    ierr = MPI_Barrier(PETSC_COMM_WORLD);
35    CHKERRQ(ierr);
36
37    ierr = PetscFinalize(); //finalizando o PETSc
38    return 0;
```

C.4 Construindo e Executando uma Aplicação PETSc

Uma aplicação PETSc pode ser construída usando um arquivo *makefile*, que contém comandos para compilar o código de acordo com a arquitetura da máquina.

Neste arquivo *makefile*, é preciso colocar os caminhos dos diretórios que contém as bibliotecas a serem usadas, os arquivos que contêm as regras para a construção da aplicação, e os comandos para compilação e execução.

O quadro a seguir apresenta um exemplo básico de um arquivo *makefile*:

```
1  ALL: helloworld
2
3  CFLAGS            = -I${PETSC_DIR}/include
```

```
4
5  include ${PETSC_DIR}/conf/variables
6  include ${PETSC_DIR}/conf/petscvariables
7  include ${PETSC_DIR}/conf/rules
8
9  helloworld: helloworld.o chkopts
10         ${CLINKER} -o helloworld helloworld.o ${PETSC_SYS_LIB}
11         ${RM} helloworld.o
12
13 runhelloworld:
14         -@${PETSC_DIR}/bin/petscmpiexec -n 2 ./helloworld
```

Para construir a aplicação, basta salvar este arquivo com o nome *makefile* no mesmo diretório do código C, e usar um comando no *shell*.

Supondo que o código tenha o nome *helloworld.c*, deve-se digitar:

```
make helloworld
```

Também é possível executar via MPI:

```
/$PETSC_DIR/bin/petscmpiexec -n 4 helloworld
```

Onde $-n$ define o número de processos paralelos (quatro, neste exemplo).

Outra possibilidade é executar com o comando definido no *makefile*, usando:

```
make runhelloworld
```

O nome "*runhelloworld*" é um *label* que foi definido no arquivo *makefile*, que já chama a execução pelo *petscmpiexec*.