Capítulo
2

Introdução a Computação Paralela com o Open MPI

Sandro Athaide Coelho

Abstract

This paper introduces an overview about Distributed Parallel Computing and related technologies using Message Passing Interface approach (MPI). MPI is a standard specification to message-passing systems sponsored by MPI-Forum.

Resumo

Este minicurso introduz uma visão geral sobre programação paralela distribuída através da abordagem de message passing. O MPI (Message Passing Interface) é uma especificação mantida pelo MPI-Forum e se tornou um padrão para comunicação de dados em Computação Paralela e de Alto Desempenho.

2.1. Introdução

Um dos grandes desafios da Ciência da Computação atualmente é viabilizar soluções computacionais que reduzam o tempo de processamento e forneçam respostas ainda mais precisas. Frequentemente surgem propostas com as mais diversas abordagens que exploram novas formas de resolver tais problemas ou tentam ainda melhorar as soluções existentes.

Uma das grandes áreas que se dedica a propor tais melhorias é a computação paralela e de alto desempenho – HPC (*High Performance Computing*) do inglês – que tem como base várias subáreas da Ciência da Computação, como por exemplo, arquitetura de computadores, linguagens de programação, compiladores, teoria da computação e computabilidade, recebendo ainda , influências de varias outras áreas, tais como computação gráfica.

Este minicurso introduzirá o padrão MPI (*Message Passing Interface*). O MPI é um conjunto de especificações de protocolos de comunicação utilizados em HPC, desenvolvido pelo consórcio MPI Forum, composto por pesquisadores e pela indústria.

A linguagem de programação utilizada durante o minicurso será o C e o conjunto de bibliotecas do Open MPI. Esta implementação fornece ainda compiladores para as linguagens de programação Fortran e C++ que não serão abordadas no minicurso, porém, após o entendimento dos exemplos podem ser facilmente adaptados.

O minicurso está organizado da seguinte maneira. A seção 2.2 apresenta motivações para aprender computação paralela e suas aplicações. A seção 2.3 introduz aspectos de arquitetura de computadores. Já a seção 2.4 mostra a classificação de Flynn. Na seção 2.5 é introduzido os dispositivos de interconexão voltados para computação paralela. Na seção 2.6 há uma explicação sobre *Clusters* e *Grids*. Na seção 2.7 é mostrado algumas métricas para computação paralela. Na seção 2.8 é apresentado o padrão MPI e sua API, enquanto a seção 2.9 apresenta uma conclusão sobre o tema.

2.2. Motivações para Computação Paralela

Técnicas que otimizem o tempo de processamento, algoritmos mais eficientes e computadores mais rápidos abrem novos horizontes possibilitando realizar tarefas que antes eram inviáveis ou mesmo levariam muito tempo para serem concluídas. Sutter (2005), em seu artigo "*The free lunch time is over",* afirma que não podemos mais esperar que nossos algoritmos fiquem mais rápidos apenas com atualizações de CPU. Muitos aplicativos se aproveitaram do fenômeno do "almoço grátis", com ganhos regulares de desempenho por décadas, simplesmente esperando novas versões de CPU, memórias e discos mais rápidos.

A Lei de Gordon Moore [Moore 1965] previa um crescimento exponencial de transistores dos chips – 100% a cada 18 meses – pelo mesmo custo. Este crescimento traduzia em ganhos exponenciais de velocidade do *Clock* e no número de instruções executadas. Porém este crescimento não pode mais, até o momento, superar os limites físicos e problemas como geração e dissipação de calor além do alto consumo de energia. Existem ainda problemas relativo ao custo no desenvolvimento de pesquisas para reduzir componentes, sem quebrar a compatibilidade do modelo atual.

A saída da indústria para superar tal limitação tem sido disponibilizar mais de um núcleo nos processadores, ao invés se concentrarem somente no aumento do *Clock*. Infelizmente algoritmos estritamente seriais não são beneficiados.

Com os avanços da ciência, o método cientifico (observação, teoria e experimentação) ganhou um importante aliado: a simulação numérica. Este por sua vez passou a ser definido como método cientifico moderno [Quinn 2003]. Ainda segundo Quinn (2003) "a simulação numérica é uma importante ferramenta para os cientistas que as vezes não podem testar suas teorias, seja pelos custos e tempo que isto envolve, por questões éticas ou ainda quando é impossível realizá-las*"*.

Simular eventos naturais como a previsão do tempo, dinâmica de fluídos, sequenciamento genético ou mesmo buscar novas drogas para a cura de doenças requerem modelos mais complexos e com grande poder computacional. Devido a magnitude de tais problemas, estes por sua vez são categorizados como os Grandes Desafios da Ciência[Quinn 2003].

Durante a construção de tais modelos, além da preocupação direta com a solução do problema, é necessário atender requisitos como precisão e/ou tempo total de

processamento -"*Wall Clock Time*". Para estes aspectos, devem ser avaliadas a utilização de tecnologias que suportem tais requisitos e ainda promovam escalabilidade. Este último ponto é muito importante pois afeta diretamente a continua evolução e aprimoramento da abordagem numérica proposta.

Fora do contexto científico, vivenciamos também uma explosão no volume de dados. As corporações, por exemplo, buscam cada vez mais ferramentas que transformem estes dados em informações valiosas, com o objetivo de fornecer respostas as necessidades de seus negócios e os ajudem a otimizar seus processos.

Em geral, tais tecnologias tem como base a tríade *Recognition – Minning – Synthesis* [Chen 2008], que utilizam gigantescos *datasets* e lançam mão do uso de soluções massivamente paralelas para processar e obter suas respostas.

Áreas emergentes como a Web semântica, por exemplo, buscam algoritmos e formas de resolver a escalabilidade com ênfase na capacidade de processamento de tarefas mais complexas e na habilidade de processar grandes *datasets* [Peiqiang et al. 2009].

Há fascinantes aplicações e excelentes oportunidades onde a computação paralela pode representar ganhos de desempenho significativos. Poderíamos relacionar inúmeros outros casos, porém acredito que os exemplos acima são suficientes e nos motivam a investir tempo para conhecimento e aprimoramento do assunto.

2.3. Arquitetura de Computadores

2.3.1. Memória distribuída (*Distributed Memory*)

Ambientes computacionais onde cada unidade de processamento possui suas próprias memórias e processadores(Figura 2.1), que estão conectados através de um barramento de rede onde trocam informações, são classificados como ambiente de memória distribuída.

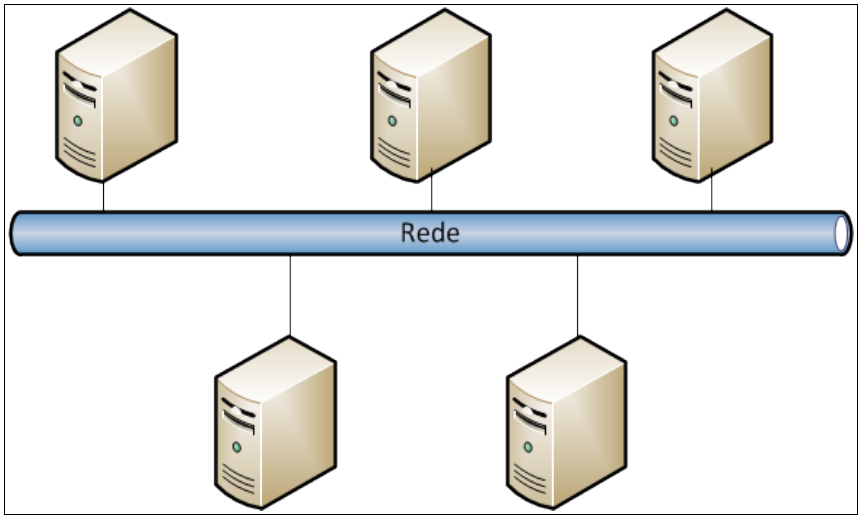

Figura 2.1. Exemplo de arquitetura paralela distribuída

A grande vantagem desta configuração é a flexibilidade em expandir ou reduzir quantas unidades de processamento forem necessárias, com o objetivo de dimensionar o seu parque computacional de acordo com o problema a ser tratado.

Os modelos que exploram ambientes computacionais de memória distribuída são os que adotam o paradigma de *message passing* (troca de mensagens). Tais modelos são formados por rotinas de comunicação e sincronização de tarefas - sendo o MPI um exemplo - que podem ser ativadas a partir de três modos:

- disponibilidade: o processo somente poderá enviar uma mensagem se o receptor estiver disponível;
- *standard*: uma mensagem pode ser enviada mesmo se não houver um receptor para ele;
- síncrono: similar ao modo padrão, porém o processo de envio só é considerado como concluído quando o receptor inicia a operação de recebimento.

Estes modelos são altamente portáveis e habilitam a execução de programas nos mais diferentes tipos de máquinas paralelas.

2.3.2. Memória compartilhada (*Shared Memory*)

Nesta arquitetura vários processadores compartilham o mesmo *pool* de memórias (Figura 2.2). A comunicação entre eles é realizada através de leitura e escrita neste espaço compartilhado que podem ser explorados por modelos de programação relativamente simples.

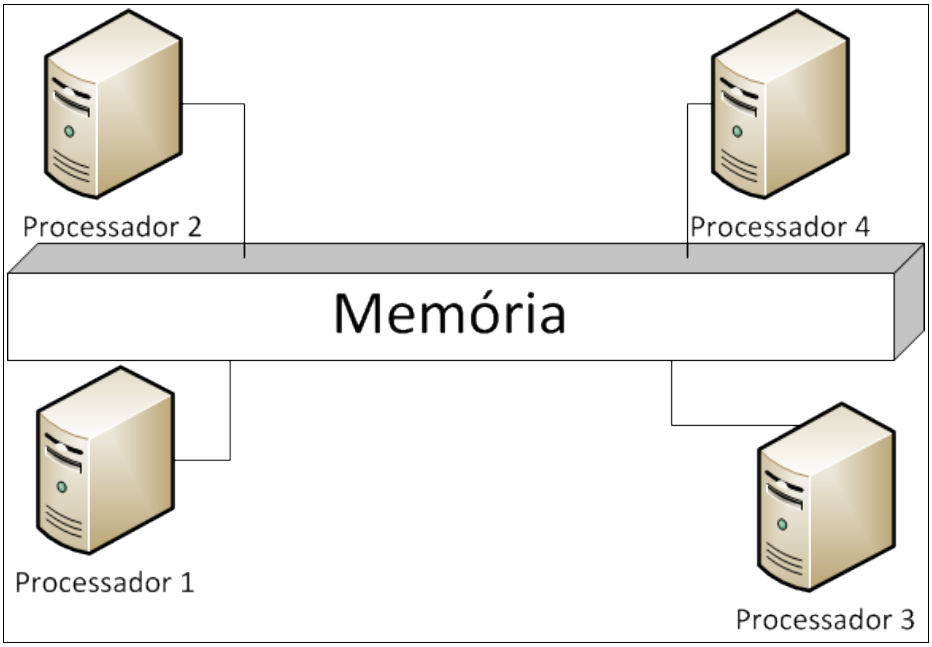

Figura 2.2. Exemplo de arquitetura paralela de memória compartilhada

A vantagem desta arquitetura é a velocidade e simplicidade dos programas, superando por exemplo o modelo distribuído, que exige uma programação mais elaborada e depende de uma camada de interconexão para a troca de informações. Já a principal desvantagem está na escalabilidade: os programas ficam limitados a capacidade física do computador onde estão sendo executados.

No passado, máquinas com esta arquitetura apresentavam custos elevados, porém com a decisão da indústria em trabalhar ganhos de desempenho, adicionando

vários núcleos em um mesmo chip, conhecidos atualmente como *cores*, reduziu significativamente os preços desta tecnologia.

Os modelos de programação projetados para este tipo de arquitetura geralmente usam processos com baixo custo computacional. Estes por sua vez são conhecidos como *Threads.* As abordagens mais conhecidas são as disponíveis nos sistemas operacionais, como o *Pthreads* em sistemas Unix *Like* e *WinThreads* no Windows .

Há ainda modelos de alto nível como o OpenMP, que fornece uma API de alto nível, portável, composta de diretivas de compilação, bibliotecas de execução e variáveis de ambientes. O OpenMP é baseado no modelo *fork-join* (Figura 2.3).

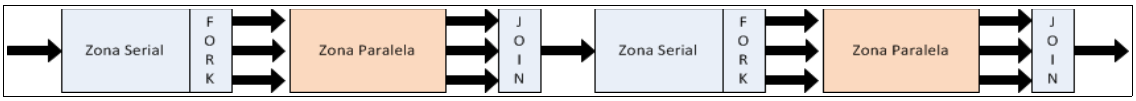

Figura 2.3. *Fork-join* no OpenMP

A seguir é apresentado um exemplo de uma função que soma um vetor paralelamente, usando o OpenMP (Figura 2.4).

```
//soma e imprime a soma do vetor
void somaVetor(double *vetor, int tamanho)
{
   int i =0;
   double soma=0;
   #pragma omp parallel for reduction(+:soma)
   for(i=0; i<tamanho; i++)
   {
      soma+=vetor[i];
   }
   printf("Resultado da soma %3.f\n", soma);
}
```

Figura 2.4. OpenMP: a diretiva #pragma omp parallel for informa ao compilador que a região deve ser paralelizada.

2.3.3. UMA (*Uniform Memory Access*)

É possível ainda subdividir as duas grandes classificações anteriores: Memória Distribuída e Memória Compartilhada, de acordo com vários critérios, dentre os quais citamos o tempo de acesso à memória.

O modelo UMA (*Uniform Memory Access*) é uma arquitetura de memória compartilhada, sendo o tempo de acesso ao *pool* de memórias uniforme, independente da localização do processador. Cada processador possui um *cache* privado que é sincronizado com a memória principal através de algoritmos de *cache-coherence* embarcados no hardware [Martin et al. 2012].

Devido as suas características, este modelo tornou-se muito adequado a aplicações de uso geral.

2.3.4. NUMA (*Non-Uniform Memory Access*)

Quando o tempo de acesso de memória está intrinsecamente relacionado a localização do processador, este grupo é denominado como NUMA. Esta arquitetura é desenhada para prover o compartilhamento total de recursos, executando um mesmo sistema operacional. Como a memória de todas as máquinas participantes são mapeadas em um mesmo espaço de endereçamento, a comunicação se dá por uma rede de interconexão.

2.4. Classificação de Flynn

Flynn (1972) elaborou uma classificação muito difundida e aceita atualmente para arquiteturas paralelas (Figura 2.5). Foram usados como parâmetros o fluxo de instruções e de dados para gerar a tabela de enquadramento das arquiteturas. Como resultado, temos as 4 (quatro) grandes classes, relacionadas a seguir:

- *Single Instruction, Single Data* (SISD): nesta categoria temos os computadores que não exploram nenhum tipo de paralelismo. Uma única unidade de controle é responsável por processar um único fluxo de instruções num único fluxo de dados. Os exemplos nesta categoria são os antigos computadores pessoais, antigos mainframes e demais máquinas que implementam a arquitetura tradicional de von Neumann [Navaux 2003];
- *Single Instruction, Multiple Data* (SIMD): projetos de arquiteturas onde uma única instrução é executada simultaneamente em múltiplos fluxos de dados. Navaux (2003) ressalta que para que este tipo de processamento ocorrer, a unidade de memória não pode ser implementada como um único módulo. Enquadram-se neste grupo máquinas *Array* como o Cray J-90, o CM-2e o MasPar, além das GPUs [Hennesy, 2012]. Seriam categorizados como MISD, as arquiteturas que apresentam múltiplos fluxos de instrução e executam suas instruções sob um mesmo fluxo de dados;
- *Multiple Instruction, Single Data* (MISD): esta categoria é fonte de divergência entre os vários cientistas e pesquisadores da área de arquitetura de computadores. De acordo com Navaux (2003), Mattson (2005) nenhum sistema conhecido se encaixa nesta categoria. Entretanto Quinn (2003) aponta como exemplo para esta categoria o *systolic array;*
- *Multiple Instruction, Multiple Data* (MIMD): finalmente encaixam-se aqui os multiprocessadores (memória compartilhada) e os multicomputadores (memória privada). As duas arquiteturas apresentam múltiplas unidades de processamento que manipulam diferentes fluxos de instrução com diferentes fluxos de dados .

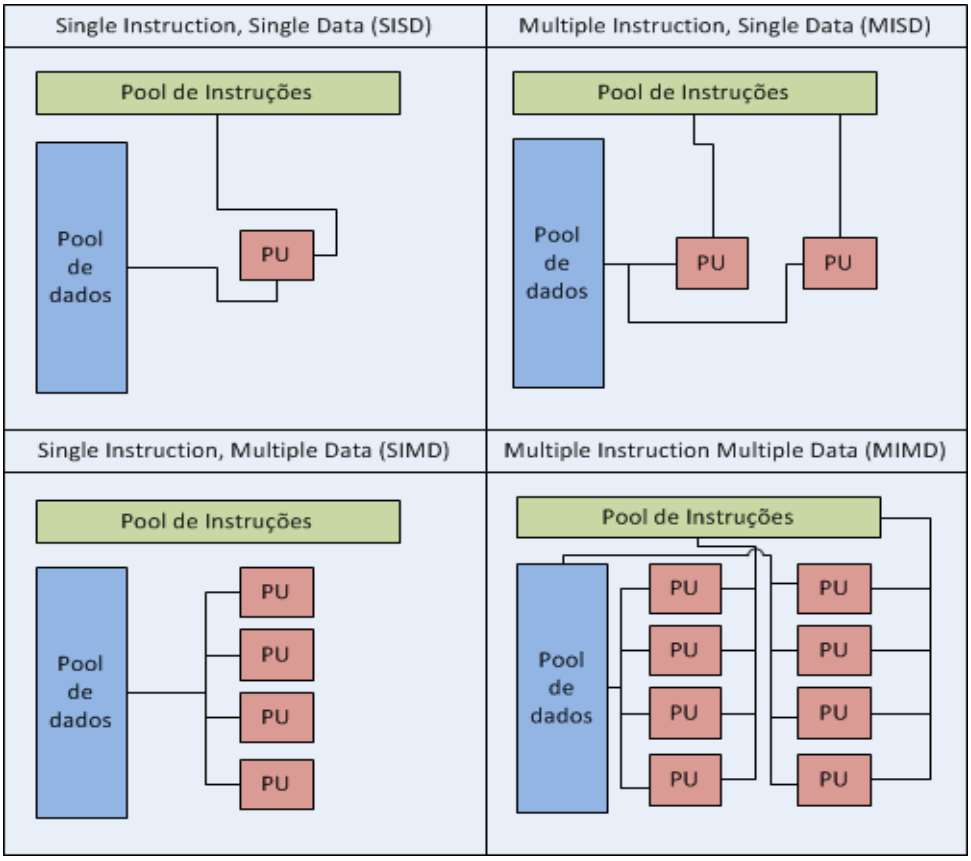

Figura 2.5. Classificação de Flynn.

2.5. Dispositivos de Interconexão

Canais de comunicação de alta velocidade, com baixa latência e máxima largura de banda são desejados para a construção de soluções distribuídas. Estes por sua vez devem atender requisitos como escalabilidade, facilidade de atualização, alta disponibilidade e bom relacionamento na relação custo/desempenho.

No artigo "*A Network Interface Controller Chip for High-Performance Computing with Distributed Pcs*", Watanabe et al. (2007) afirma que a maioria dos clusters de alto desempenho utilizam soluções SAN (*System Area Network*), como o padrão aberto chamado Myrinet. Este padrão provê alto desempenho e comunicação eficiente de pacotes de rede, com uma boa relação custo/desempenho. Porém para conectar computadores que estão posicionados a longas distâncias no andar de um prédio ou mesmo ao longo de um edifício, esta solução se torna inadequada. Por outro lado a utilização da Ethernet e do protocolo TCP/IP, em pequenas redes locais, tornou-se muito popular como camada de interconexão entre computadores, principalmente após a popularização do *Beowulf cluster*.

Os esforços na eliminação de gargalos de comunicação e a maximização da velocidade de transmissão estão presentes em tecnologias como Memory Channel, Quadrics e o Infiniband. Não iremos abordar este assunto com maior aprofundamento, porém, a seguir, encontra-se um resumo de como funcionam estas tecnologias.

- Memory Channel: segundo Dantas (2005), a abordagem conhecida como Memory Channel fornece até 100Mb/s de velocidade de transferência e a distância física pode ser de até 10 (dez) metros. Esta tecnologia usa um *switch* externo e placas específicas para realizar a sincronização dos dados.
- Quadrics ou QsNET: a topologia desta rede é do tipo *Fat-Tree*, sendo possível expandir a rede a milhares de dispositivos de comutação. A taxa de transferência pode atingir a centenas de Mb/s e fornece baixa latência. Utiliza uma interface de rede programada chamada Elan [Dantas 2005].
- Infiniband: é uma especificação para protocolos de comunicação gerida pelo grupo Infiniband Trade Association (http://www.infinibandta.org). Este grupo é composto por fornecedores líderes de mercado tais como HP, IBM, Intel, Lellanox, Oracle, Qlogic e System Fabric Works. Este protocolo fornece uma interface de comunicação com elevadas taxas de transferência e suporte a endereçamento para até 64.000 dispositivos, suportando *failover* e QoS (*Quality of Service*).

2.6. *Clusters* e *Grids*

Os *clusters* são aglomerados de computadores, interconectados através de uma mesma rede, que utilizam sistemas e componentes de hardwares especializados, fornecendo ao usuário final uma "imagem" unificada do sistema. Neste arranjo, cada componente do aglomerado é chamado de nó e geralmente existe um deles responsável por controlar e gerenciar os demais, denominado mestre. Para que sejam eficientes, a camada de comunicação deve ser rápida e confiável o suficiente para maximizar a utilização dos processadores e não gerar perdas de dados.

Durante o projeto e implementação destes aglomerados, são estudadas características importantes para garantir a confiabilidade do sistema. Aspectos como escalabilidade, tolerância a falhas, disponibilidade e a interdependência entre estes fatores são reunidas em torno da meta de construir um sistema que seja aderente ao propósito final, com menor custo de processamento/hora.

Há disponível no mercado *clusters* com as mais diversas finalidades, porém os mais comuns são os relacionados a seguir:

- alto desempenho: utilizados para executar aplicações que requerem grande quantidade de processamento como em aplicações que envolvem cálculos numéricos, simuladores e programas científicos em geral;
- alta disponibilidade: são *clusters* construídos com aplicações auto gerenciáveis, que conseguem identificar falhas e proteger o sistema garantindo o *uptime* dos aplicativos em execução;
- balanceamento de carga: composto de mecanismos que gerenciam a carga de trabalho entre os nós do agrupamento, escalonando de forma controlada, para não sobrecarregar somente alguns recursos. Conta também com recursos que identificam a indisponibilidade e falhas dos nós gerenciados.

Computadores de múltiplos domínios, trabalhando em torno de uma mesma tarefa com o objetivo de finalizar-la o mais rápido possível são classificados como *Grid*.

Os *Grids* utilizam *middlewares* que dividem uma grande tarefa em pequenas partes distribuindo entre os nós participantes e consolidando os resultados.

Um dos mais bem-sucedidos *middlewares* para computação em *Grid* é o projeto *open source* encabeçado pela Universidade da Califórnia, nos EUA. O *Berkeley Open Infrastructure for Network Computing,* mais conhecido pelo seu acrônimo BOINC. O BOINC é um software para computação voluntária utilizado em importantes projetos como o SETI@home, que analisa ondas de rádio provenientes do espaço sideral em busca de vidas extraterrestres. O CERN (*European Organization for Nuclear Research*) também utiliza este software para analisar os dados gerados pelo Grande Colisor de Hádrons , o LHC. Há ainda projetos específicos como o World Community Grid e o Folding@Home, este último da universidade de Standford, que se dedicam a buscar cura para doenças.

2.7. Métricas em Computação Paralela

Em um primeiro contato com a computação paralela, podemos nos deixar levar pela errônea ideia de que ao dividir o trabalho em *n* unidades de processamento, os ganhos desempenho seriam proporcionalmente em *n* vezes. É importante ressaltar que isto nem sempre é possível. Navaux (2003) relaciona os grandes desafios na área de processamento de alto desempenho, que explicam o motivo pelo qual nem sempre é possível obter tal resultado. Os mesmos estão pautados em 4 (quatro) aspectos: arquiteturas paralelas, gerenciamento das máquinas, linguagens mais expressivas que consigam extrair o máximo de paralelismo e finalmente uma maior concorrência nos algoritmos.

Para nos auxiliar a mensurar os ganhos ao paralelizar algoritmos seriais, podemos utilizar a métrica denominada *Speedup*. Em computação paralela, o *Speedup* é utilizado para conhecer o quanto um algoritmo paralelo é mais rápido que seu correspondente sequencial. Pode ser obtido pela seguinte equação:

$$Speedup = \frac{Tempo\ serial}{Tempo\ Paralelo}$$

O resultado deve variar entre 0 (zero) – quanto mais próximo a zero, maiores foram os ganhos - e 1 (um). Caso seja superior a 1 (um), o algoritmo paralelo é menos eficiente que o serial e precisa ser avaliado para a remoção de gargalos, ou ainda, para apurar se foram escolhidas as melhores abordagens durante o projeto do seu código.

Objetivando, portanto, estimar teoricamente os ganhos ao se paralelizar todo ou parte de um código, foi anunciada a lei de Amdahl, a qual assume que o desempenho do algoritmo fica limitado pela fração de tempo das partes paralelizáveis durante sua execução, sendo obtido a partir da equação:

$$Speedup = \frac{1}{(Fração\ Sequencial\ do\ Programa + (1 - \frac{Fração\ Sequencial\ do\ Programa}{Número\ de\ Processadores}))}$$

Alguns cientistas consideram o *Speedup* obtido pela lei de Amdahl uma abordagem pessimista, pois não avalia a complexidade e o tamanho do problema a ser tratado. Diante disto, foi anunciada uma segunda lei, que leva em conta tais fatores, segundo a equação:

$$Speedup = Núm.processadores - Fração\ sequêncial\ do\ programa\left(Núm.processadores - 1\right)$$

Esta lei é conhecida como lei de Gustafon-Barsis.

2.8. O MPI

Durante a década de 90, foram produzidas duas especificações que unificaram os trabalhos dos participantes do MPI Forum (http://www.mpi-forum.org) na área de processamento paralelo e distribuído. Este trabalho foi encabeçado por um grupo formado por cerca de 60 pessoas provenientes de mais de 40 organizações dos Estados Unidos e Europa, sendo fortemente influenciado pelos resultados até então alcançados nos grupos do IBM T.J. Watson Research Center, Intel's NX/2 , Express, nCUBE's Vertex, p4 e o PARMACS. Outras importantes contribuições vieram também do Zipcode, Chimp, PVM, Chameleon e PICL.

A primeira versão – MPI-1 - foi publicada no ano de 1994 contando com os requisitos base necessários para o padrão de troca de mensagens. Os pontos mais complexos foram deixados para a versão MPI-2, lançada posteriormente em 1997 [Mattson et al. 2010] .

Os trabalhos mais recentes do MPI Forum remontam aos anos de 2008 e 2009, tendo como diretrizes a correção de *bugs*, a compilação das duas especificações – MPI-1 e MPI-2 - em um único documento e a publicação de revisões *minor*: MPI-2.1 e MPI-2.2. A última atualização aconteceu em setembro de 2012, quando o grupo publicou o maior grupo de atualizações no padrão, incluindo novas extensões, a remoção de *bindings* obsoletos para o C++ e o suporte ao FORTRAN 2008. Este último *release* foi nomeado como MPI-3.0.

Apesar de não ser uma norma homologada por institutos como IEEE, IETF, o MPI é "*De facto*" um padrão para arquiteturas de memória distribuída, escrito e ratificado pelo MPI Forum. A idéia central do MPI consiste em prover rotinas de sincronização e controle processos, além da distribuição e consolidação dos dados entre os computadores interconectados, utilizando para isto o paradigma de troca de mensagens.

O modelo da API do MPI, entrega ao programador uma camada de abstração, altamente portável, entre o sistema operacional/tecnologias de comunicação e a aplicação, permitindo escalar a execução do programa entre os computadores da rede, com o uso de suas memórias e poder de processamento, baseado no padrão de divisão e conquista. Por ser uma camada de alto nível, é possível utilizar o mesmo programa em uma grande variedade de computadores que estejam interconectados. A API é muito adequada a programas de propósito geral desenhados para sistemas SIMD e MIMD.

O padrão tem como foco auxiliar a todos que precisem ou desejem criar aplicativos portáveis e paralelos em Fortran e C, fornecendo uma interface simples que atenda desde soluções menos sofisticadas até as mais avançadas, voltadas para o alto desempenho, como as disponíveis em computadores massivamente paralelos.

Cabe enfatizar que o MPI é uma especificação. No entanto, há disponíveis no mercado implementações livres e pagas. Podemos relacionar no modelo pago o Intel MPI, HP MPI e o MATLab MPI. Já as implementações *open source* mais utilizadas são

o Open MPI e o MPICH. Fabricantes de supercomputadores costumam incluir *features* e ferramentas específicas que visam maximizar o desempenho em suas máquinas.

Apesar de todas as facilidades do padrão, a identificação das zonas paralelas (áreas do código que podem ser executadas simultaneamente a outras tarefas) e o dimensionamento de carga entre os computadores participantes, ficam a cargo do desenvolvedor e devem ser explicitadas no programa. Uma análise prévia do problema a ser tratado, irá auxiliá-lo nesta tarefa.

Como o objetivo é o aumento de desempenho, deve-se ter em mente o projeto de um programa que realiza a menor troca de mensagens possível, visando anular e/ou reduzir o impacto do tempo despendido na distribuição e consolidação dos dados, no tempo total de execução. Podemos relacionar como exemplos os Hipercubos e Meshes, que são formas de interconexão que visam reduzir e otimizar o processamento. Não iremos abordar este tema aqui por ser tratar de um tópico complexo para quem está começando a explorar computação paralela.

Para finalizar a apresentação do padrão, vamos conhecer alguns conceitos importantes inerentes a API, os quais nos ajudarão a compreender as funções disponíveis e como funciona o MPI. Estes são:

- Processo: é o módulo executável, ou seja, seu programa, que pode ser dividido em N partes, e executado em uma ou mais máquinas;
- Grupo: é o conjunto de processos. No MPI a constante MPI_COMM_WORLD armazena o comunicador de todos os processos definidos na aplicação MPI;
- *Rank*: o controle de execução do programa em MPI depende diretamente da correta identificação do processador no grupo de trabalho. Esta identificação é um número inteiro que varia de 0 a N-1 processos. A função MPI_Comm_rank determina qual é o identificador de cada processo dentro do comunicador;
- Comunicador: é o componente que representa o domínio de uma comunicação em uma aplicação MPI;
- *Buffer* de Aplicação: é o espaço reservado em memória para armazenar os dados das mensagens que um processo necessita enviar ou receber, gerenciado pela aplicação;
- *Buffer* de Sistema: é o espaço reservado em memória pelo sistema para o gerenciamento de mensagens.

2.8.1 Estrutura de um código MPI

Antes de avançarmos nas funções que compõe a API, precisamos compreender alguns aspectos estruturais no código. Na declaração de dependências, é obrigatório a inclusão do cabeçalho mpi.h em todos os arquivos que utilizarem algum recurso do MPI. Este cabeçalho contém as definições de funções e constantes necessários durante o desenvolvimento e compilação do seu programa. Todas as chamadas MPI devem estar entre o MPI_Init e o MPI_Finalize, conforme a seguir(Figura 2.6).

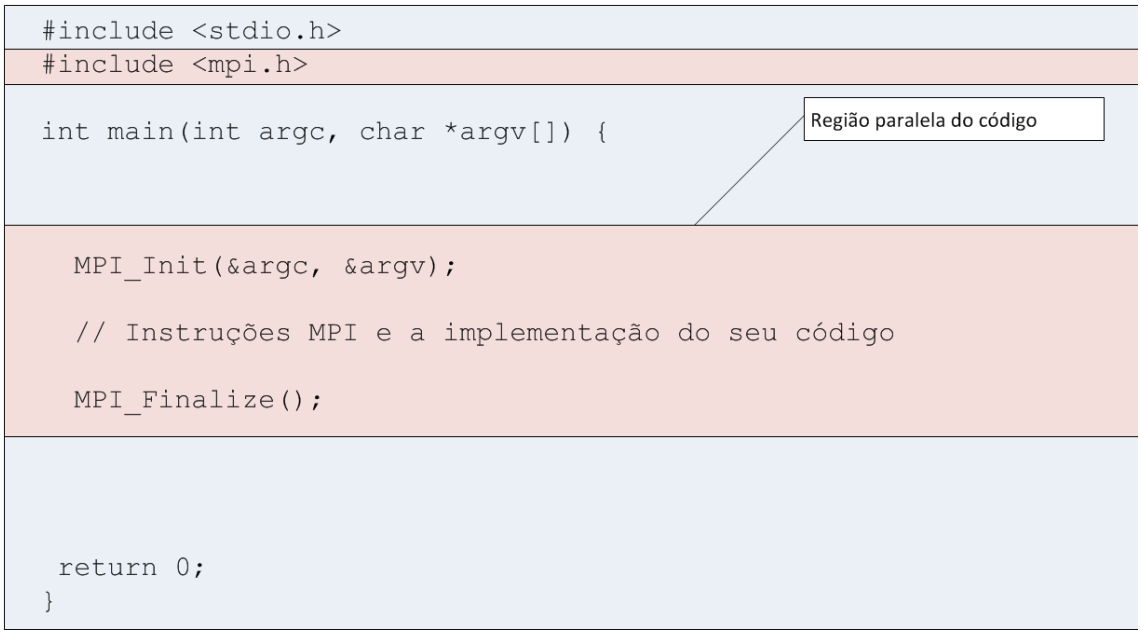

Figura 2.6. Estrutura de um código em C/MPI.

O MPI emprega uma consistente padronização, que ajuda o desenvolvedor a identificar facilmente métodos e tipos da API. Todos os identificadores possuem o prefixo "MPI_". Tipos e constantes estão em letras maiúsculas (Ex. MPI_SHORT) e as funções estão definidas com a primeira letra em maiúscula e as demais letras em minúsculas. (Ex. MPI_Send).

Para o tratamento de exceções, a biblioteca conta com um manipulador de erros próprio. Caso ocorra algum problema durante a execução, o mesmo é responsável por sinalizar a ocorrência para seu aplicativo e abortar as chamadas MPI .

Devido ao suporte a várias linguagens, foi necessário criar tabelas de equivalência de tipos de dados entre a API e as linguagens. A seguir, a tabela de equivalência para o C (Tabela 2.1):

Tabela 2.1. Equivalência de tipos de dados

Tipo de Dados do MPI	Tipo de Dados do C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int

MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_BYTE	

2.8.2 MPI_Init

Este método inicializa o ambiente de execução do MPI, sincronizando os processos entre os computadores participantes e deve ser invocado uma única vez durante toda a execução do seu aplicativo. Caso contrário, o MPI irá sinalizar com o erro MPI_ERR_OTHER. A assinatura do método encontra-se a seguir:

```
int MPI_Init( int *argc, char ***argv );
```

argc

Ponteiro indicando o número de argumentos

argv

Ponteiro do vetor com os argumentos

Listagem 2.1. MPI_Init

Devemos assegurar que a chamada seja realizada por apenas uma *Thread* e que esta mesma *Thread* invoque, ao final do processamento, o método MPI_Finalize.

2.8.3 MPI_Finalize

Finaliza o ambiente de execução do MPI, liberando os recursos utilizados durante o processamento. Deve ser invocado sob a mesma *Thread* que executou o MPI_Init. Este método não possui argumentos.

2.8.4 Envio e recebimento de dados síncrono: MPI_Send e MPI_Recv

O MPI disponibiliza várias rotinas para realizar as operações de envio e recebimento de mensagens entre os processadores, enquanto a de envio é feita pela função MPI_Send, a de recebimento é pelo MPI_Recv. Estes dois métodos realizam a operação de envio e recebimento de mensagens de forma síncrona, ou seja, aguarda a confirmação de recebimento da mensagem para passar para a próxima instrução. As definições encontram-se a seguir:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int
dest,int tag, MPI_Comm comm);
```

buf

Endereço do *buffer* de dados a ser enviado.

count

Número de elementos a ser enviado (número inteiro e não negativo).

datatype
Tipo do dado.
dest
Rank do destinatário.
tag
Identificador para a mensagem (número inteiro).
comm
Comunicador. Em SIMD (MPI_COMM_WORLD), Em MIMD (O comunicador criado pela função MPI_Comm_create)

Listagem 2.2. MPI_Send

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,int source,
int tag, MPI_Comm comm, MPI_Status *status);
```

buf
Endereço do buffer para receber os dados.
count
Número de elementos a ser recebido (número inteiro e não negativo).
datatype
Tipo do dado.
source
Rank do remetente.
tag
Identificador para a mensagem (número inteiro).
comm
Comunicador. Em SIMD (MPI_COMM_WORLD), Em MIMD (O comunicador criado pela função MPI_Comm_create)
Status
Status da mensagem

Listagem 2.3. MPI_Recv

2.8.5 Envio e recebimento de dados assíncrono: MPI_Isend e MPI_Irecv

O MPI_Isend e o MPI_Irecv fornecem a funcionalidade de envio e recebimento assíncrono, respectivamente. Como só o comportamento é modificado, é possível combinar os métodos MPI_Send/MPI_Recv com os métodos MPI_Isend/MPI_Irecv.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request);
```

buf
Endereço do *buffer* de dado a ser enviado.
count
Número de elementos a ser enviado (número inteiro e não negativo).
datatype
Tipo do dado.
dest
Rank do destinatário.
tag
Identificador para a mensagem (número inteiro).
comm
Comunicador. Em SIMD (MPI_COMM_WORLD), Em MIMD (O comunicador criado pela função MPI_Comm_create)

request

Solicitação de envio. Utilizado pelas funções de bloqueio como MPI_Wait para assegurar a conclusão do processamento.

Listagem 2.4. MPI_Isend

```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,int source,
int tag, MPI_Comm comm, MPI_Request *request);
```

buf

Endereço do buffer para receber os dados.

count

Número de elementos a ser recebido (número inteiro e não negativo).

datatype

Tipo do dado.

source

Rank do remetente.

tag

Identificador para a mensagem (número inteiro).

comm

Comunicador. Em SIMD (MPI_COMM_WORLD), Em MIMD (O comunicador criado pela função MPI_Comm_create)

request

Solicitação de recebimento. Utilizado pelas funções de bloqueio como MPI_Wait para assegurar a conclusão do processamento.

Listagem 2.5. MPI_Irecv

2.8.6 MPI_Wait

Ao realizar operações assíncronas, podemos nos deparar com situações de dependência (dados, instruções, entre outros) onde é necessário assegurar a conclusão dos mesmos antes de executar a próxima instrução. Para solucionar tais questões o MPI fornece funções como o MPI_Wait.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

request

Identificador da requisição a ser bloqueada.

status

Retorna a informação sobre a situação da operação. Para testar o status utilize a função MPI_Test

Listagem 2.6. MPI_Wait

2.8.7 Comunicações Coletivas

Até o momento, foram apresentadas rotinas que fazem a comunicação entre dois processos específicos, denominados ponto-a-ponto. Adicionalmente a estas funções, o MPI provê suporte a operações que devem ser realizadas simultaneamente em mais de

dois processos de um grupo de trabalho, especificados através do contexto - para arquiteturas SIMD (MPI_COMM_WORLD), já em MIMD (o comunicador criado pela função MPI_Comm_create).

As operações mais comuns nas comunicações coletivas são as que realizam sincronização, distribuição de dados (*Broadcast*) e finalmente, as reduções (*Reduction*) para operações de consolidação de dados, como operações de soma por exemplo.

2.8.8 MPI_Barrier

O MPI_Barrier define um ponto de sincronização entre todos os processos participantes da comunicação coletiva, isto é, o código não pode prosseguir enquanto todos não estiverem no sincronizados no mesmo ponto.

```
int MPI_Barrier(MPI_Comm comm);
```

comm
Comunicador. Em SIMD (MPI_COMM_WORLD), Em MIMD (O comunicador criado pela função MPI_Comm_create)

Listagem 2.7. MPI_ Barrier

2.8.9 MPI_Bcast

A distribuição de uma mensagem para todos os processos do grupo é realizado pela função MPI_Bcast.

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,int
root, MPI_Comm comm);
```

buf
Endereço do *buffer* de dados a ser enviado.
count
Número de elementos a ser enviado (número inteiro e não negativo).
datatype
Tipo do dado.
root
Rank do destinatário.
comm
Comunicador. Em SIMD (MPI_COMM_WORLD), Em MIMD (O comunicador criado pela função MPI_Comm_create)

Listagem 2.8. MPI_ Bcast

2.8.10 MPI_Reduce

Para a consolidação dos resultados do processamento em comunicações coletivas, a função MPI_Reduce fornece vários tipos de operações (Tabela 2.2).

Tabela 2.2. Operações do MPI_Reduce no MPI

ID	Operação
MPI_MAX	Maior número

MPI_MIN	Menor número
MPI_SUM	Soma
MPI_PROD	Produto
MPI_LAND	Conjunção lógica E
MPI_BAND	Conjunção lógica bit a bit E
MPI_LOR	Conjunção lógica OU
MPI_BOR	Conjunção lógica bit a bit OU
MPI_LXOR	Disjunção lógica
MPI_BXOR	Disjunção lógica bit a bit
MPI_MAXLOC	Máximo e sua localização
MPI_MINLOC	Mínimo e sua localização

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm comm)
```

sendbuf
Endereço do *buffer* de envio.

recvbuf
Endereço do *buffer* de recebimento.

count
Número de elementos a ser enviado (número inteiro e não negativo).

datatype
Tipo do dado.

op
Identificador da operação de redução.

root
Rank do processo que realizará a operação de redução.

comm
Comunicador. Em SIMD (MPI_COMM_WORLD), Em MIMD (O comunicador criado pela função MPI_Comm_create)

Listagem 2.9. MPI_Reduce

2.8.11 MPI_Scatter e MPI_Gather

Para a distribuição e recolhimento de mensagens contendo pedaços de dados de mesmo tamanho, o MPI fornece dois métodos: o MPI_Scatter e o MPI_Gather. Estas duas funções são muito úteis em operações envolvendo álgebra linear, como multiplicação de matrizes, por exemplo. O MPI_Scatter é responsável pela distribuição dos dados, já o MPI_Gather pelo recolhimento.

```
int MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype
```

```
sendtype,void *recvbuf, int recvcount, MPI_Datatype recvtype, int
root,MPI_Comm comm)
```

sendbuf
Endereço do *buffer* para envio dos dados.
sendcount
Número de elementos a ser enviado (número inteiro e não negativo).
sendtype
Tipo do dado.
recvcount
Número de elementos no *buffer* de recebimento (número inteiro e não negativo).
recvtype
Tipo do dado.
root
Rank do processo de envio.
comm
Comunicador. Em SIMD (MPI_COMM_WORLD), Em MIMD (O comunicador criado pela função MPI_Comm_create)

Listagem 2.10. MPI_Scatter

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype
sendtype,void *recvbuf, int recvcount, MPI_Datatype recvtype, int
root,MPI_Comm comm)
```

sendbuf
Endereço do *buffer* para envio dos dados.
sendcount
Número de elementos a ser enviado (número inteiro e não negativo).
sendtype
Tipo do dado.
recvcount
Número de elementos no *buffer* de recebimento (número inteiro e não negativo).
recvtype
Tipo do dado.
root
Rank do processo de envio.
comm
Comunicador. Em SIMD (MPI_COMM_WORLD), Em MIMD (O comunicador criado pela função MPI_Comm_create)

Listagem 2.10. MPI_Gather

2.8.12 MPI_Allgather

Quando todos os processos precisam receber os dados coletados dos demais processos do grupo, o MPI_Allgather deve ser utilizado. Novamente aqui os dados precisam ser do mesmo tamanho.

```
int MPI_Allgather(void *sendbuf, int  sendcount,MPI_Datatype
sendtype, void *recvbuf, int recvcount,MPI_Datatype recvtype,
MPI_Comm comm)
```

sendbuf
Endereço do *buffer* para envio dos dados.

sendcount
Número de elementos a ser enviado (número inteiro e não negativo).
sendtype
Tipo do dado.
recvbuf
Endereço do *buffer* para recebimento dos dados.
recvcount
Número de elementos no *buffer* de recebimento (número inteiro e não negativo).
recvtype
Tipo do dado.
comm
Comunicador. Em SIMD (MPI_COMM_WORLD), Em MIMD (O comunicador criado pela função MPI_Comm_create) .

Listagem 2.11. MPI_Allgather

2.8.13 Tipos derivados de dados.

Os mecanismos básicos de comunicação do MPI podem ser usados para enviar ou receber sequências de elementos de mesmo tipo, que podem ser ou não contíguos em memória. O objetivo é amortizar o custo desta operação e prover ganhos no tempo total de execução do seu algoritmo.

Diferentemente da abordagem de declaração em tempo de desenvolvimento, no MPI os tipos de dados são criados em tempo de execução, entre as funções MPI_Init e MPI_Finalize. O ciclo de vida destes tipos possuem as seguintes etapas: declaração, alocação, o uso efetivo do tipo durante a execução do programa e, finalmente, a liberação dos recursos. Os métodos de construção destes tipos são:

- MPI_Type_contiguous: é o construtor mais simples dentre os disponibilizados no MPI. Gera um novo tipo contínuo a partir de cópias de um tipo de dados existente.
- MPI_Type_vector: semelhante ao MPI_Type_contiguous, porém permite a configuração de lacunas (*strides*) nos deslocamentos. Há ainda a função MPI_Type_hvector que é um MPI_Type_vector, porém definido em bytes.
- MPI_Type_indexed: gera uma matriz de deslocamentos do tipo de dados como um mapa para o novo tipo de dados. Há ainda a sua variação em bytes: MPI_Type_hindexed.
- MPI_Type_struct: dos tipos derivados, este é o mais genérico. O novo tipo é formado de acordo com cada componente da estrutura de dados.

Antes de ser utilizado, é necessário informar aos processadores da comunicação coletiva que o tipo está disponível. Esta operação é feita através da função MPI_Type_commit. Para a liberação dos recursos alocados, a função é a MPI_Type_free. Como o free no C, o MPI_Type_free evita o consumo de memória desnecessário, para os tipo de dados que não são mais utilizados.

A utilização de tipos derivados é recomendada por prover uma solução portável e elegante para o envio de dados não contíguos, maximizando a eficiência dos métodos de envio e recebimento de dados, promovendo ganhos no tempo total de execução do seu programa.

2.9. Conclusões

Neste minicurso buscamos explorar uma visão geral sobre arquiteturas paralelas e o modelo de programação baseado no paradigma de *message passing*. A partir das funções e ideias apresentadas, é possível construir códigos explorando paralelismo com o MPI. As *features* mais complexas do MPI não abordadas retratam questões de refinamento de desempenho e manipulação de arquivos de forma paralela. Estas características podem ser conhecidas através do site do MPI-Forum(http://www.mpi-forum.org) ou por intermédio das bibliografias listadas nas referências.

Em Computação Paralela, há ainda outras tecnologias que podem ser utilizadas em conjunto ao MPI. Em Memória Compartilhada (*Shared Memory*), por exemplo, o modelo do OpenMP fornece os mesmos benefícios de portabilidade do OpenMPI para este tipo de arquitetura. Há ainda abordagens que exploram paralelismo em placas de vídeo , como o modelo livre chamado OpenCL, definido pelo consórcio de empresas chamado Khronos Group, além dos modelo proprietários propostos pela NVIDIA (CUDA) e o OpenACC(http://www.openacc-standard.org/), encabeçado atualmente pela PGI, Nvidia, Caps e Cray.

Referências

Chen, Yen-Kuang. (2008), "Convergence of Recognition, Mining, and Synthesis Workloads and Its Implications", http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4479863&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D4479863, Maio.

Dantas, Mario. (2005), Computação Distribuída de Alto Desempenho, Axcel, 1º edição.

De Rose, César A. F. Navaux, Phillippe O. A. (2005), Arquiteturas Paralelas, Bookman, 1º edição.

Gebali, Fayez. (2011), Algorithms and Parallel Computing, Wiley, 1º edição.

Hennesy. John L. Patterson, David A. (2012), Computer Architecture, Morgan Kaufamann, 5º edição.

Kirk, David B. Hwu, Wen-mei W. (2010), Programming Massively Parallel Processors, Morgan Kaufmann, 1º edição.

Martin, Milo M. K. Hill,Mark D. Sorin,Daniel J. (2012), "Why on-chip cache coherence is here to stay", http://doi.acm.org/10.1145/2209249.2209269, Julho

Mattson, T.G, Sanders. B.A., Massingill, B.L. (2010), Patterns for Parallel Programming. Addison-Wesley, 1º edição.

Mishra, Bhabani Shankar. and Dehuri, Satchidananda (2011). "Parallel Computing Environments: A Review"., http://tr.ietejournals.org/article.asp?issn=0256-4602;year=2011;volume=28;issue=3;spage=240;epage=247;aulast=Mishra;t=6, Maio

MPI Forum. (2012), "MPI-3.0 Standard Specification", http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, Setembro.

Peiqiang, L. Zeng, Y. Kotoulas, S. Urbani, J. and Zhong, N. (2009) "The Quest for Parallel Reasoning on the Semantic Web", http://dx.doi.org/10.1007/978-3-642-04875-3_45, Novembro.

Pitt-Francis, J. and Whiteley, J. (2012), An Introduction to Parallel Programming Using MPI - Guide to Scientific Computing in C++ ,Springer, 1º edição

Quammen, Cory. (2005). "Introduction to programming shared-memory and distributed-memory parallel computers", http://doi.acm.org/10.1145/1144382.1144384 , Outubro

Quinn, Michael J. (2003), Parallel Programming in C with MPI and OpenMP, Mc Graw Hill, 1º edição.

Watanabe, K. Otsuka, T. Tsuchiya, J. Nishi, H. Yamamoto, J. Tanabe, N. Kudoh, T. and Amano, H. (2007), "A Network Interface Controller Chip for High Performance Computing with Distributed PCs, *Parallel and Distributed Systems",* http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4288127&isnumber=4288116, Setembro.