Programação com MPI

Ementa

- Introdução
- Plataformas de Hardware
- Modelos de Programação
- Comunicação Assíncrona e Síncrona
- Avaliação de desempenho - Conceitos básicos
- Apresentação do MPI

Bibliografia

https://www.casadocodigo.com.br/products/livro-programacao-paralela

Google Colab

- Os exemplos, notebooks e slides do minicurso estão disponíveis no seguinte endereço:
- https://github.com/Programacao-Paralela-e-Distribuida/MPI/

Arquiteturas Paralelas

Arquiteturas Paralelas

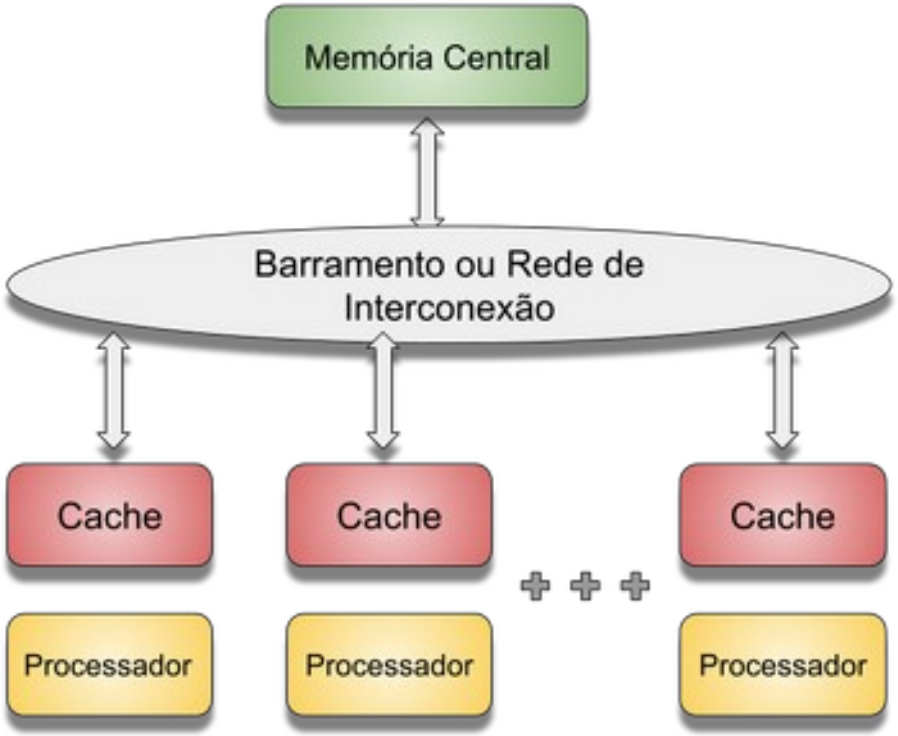
- As arquiteturas paralelas são aquelas capazes de explorar o paralelismo existente nas aplicações e podem ser organizadas de diversas maneiras.
- As arquiteturas paralelas podem explorar o paralelismo em diversos níveis, algumas vezes de forma transparente ao usuário, outras vezes não.
- Para cada tipo de arquitetura existe um paradigma de programação mais adequado, de modo a obter a maior aceleração possível, quando comparado com a execução sequencial da aplicação.

Classificação de Flynn

- SISD (Single Instruction Single Data Stream)
  - São os processadores convencionais, onde apenas um processo ou *thread* é executado por vez, podendo fazer uso de técnicas de exploração de paralelismo temporal (*pipeline, superpipelined)* ou espacial (superescalares) no nível de instrução, de forma transparente ao usuário.
- SIMD (Single Instruction Multiple Data Streams)
  - São os processadores vetoriais (supercomputadores Cray, NEC), unidades funcionais vetoriais (Intel AVX-512), arquiteturas SIMD (processamento de imagem), *tensor cores* (Google TPUs).
- MIMD (Multiple Instruction Multiple Data Streams)
  - Podem ser de memória compartilhada, como os multiprocessadores (*multicores*), o de memória distribuída, como os multicomputadores (*clusters*).

Arquiteturas de Memória Compartilhada

- Cada processador tem acesso a um espaço de endereçamento global comum a todos.
- Processos ou *threads* trocam informações diretamente na memória, sem necessidade de comunicação explícita com troca de mensagens.
- Os processadores fazem uso da memória cache para diminuir a contenção no acesso à memória.



Arquiteturas de Memória Compartilhada

- As suas principais vantagens são:
  - O código de programas sequenciais podem ser facilmente adaptados para versões paralelas.
  - A comunicação entre processos ou *threads* é bastante eficiente.
  - Acesso rápido à memória comparado a arquiteturas distribuídas.
- As suas principais desvantagens são:
  - Uso de primitivas de sincronização para assegurar um resultado correto para a computação.
  - Necessidade de mecanismos de coerência de cache para manter os dados atualizados entre caches locais.
  - O aumento de processadores pode causar contenção de memória e gargalos no barramento.

Arquiteturas de Memória Compartilhada

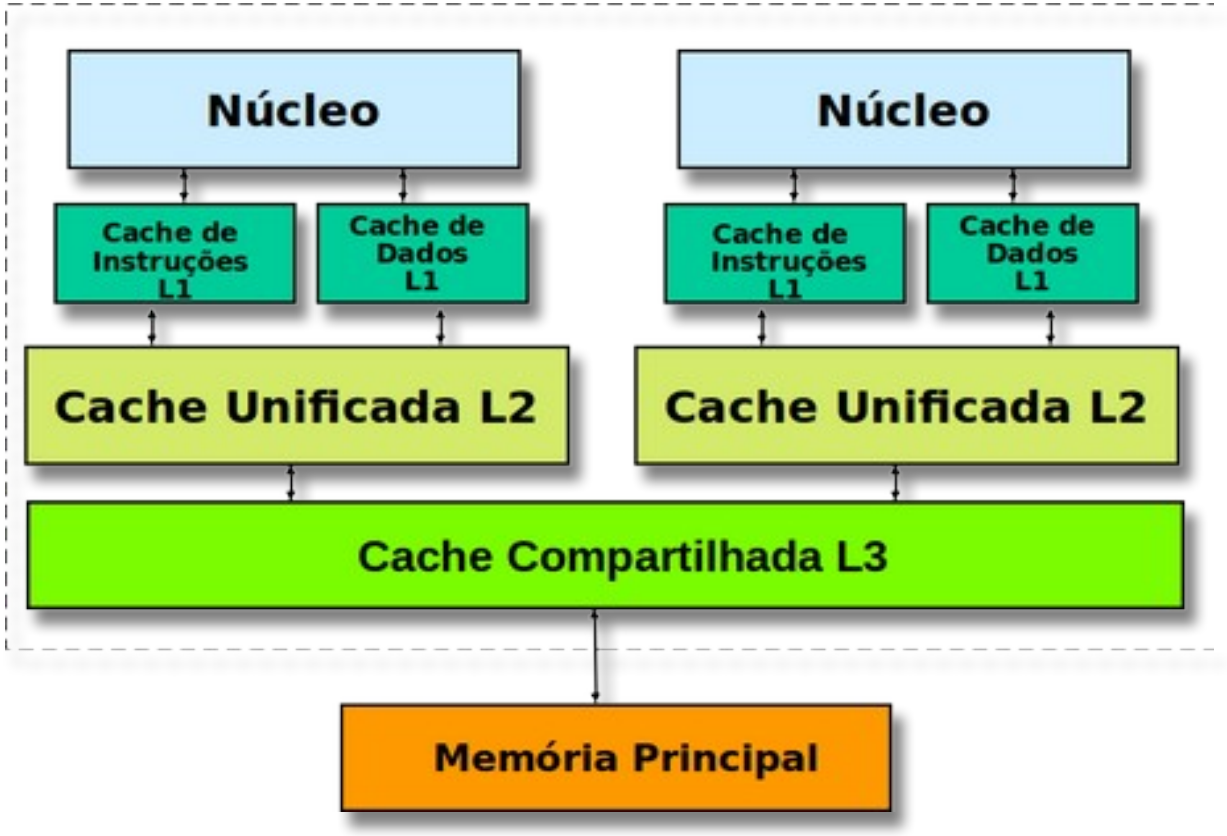
- Essas arquiteturas possuem diversas classificações e variações em sua construção, tais como:
  - UMA (Uniform Memory Access): São arquiteturas com memória única global, onde o tempo de acesso à memória é uniforme para todos os nós de processamento.
  - NUMA (Non-uniform Memory Access): A memória é dividida em blocos locais a cada processador do sistema, ligados por uma rede de interconexão. O acesso aos dados na memória local é muito mais rápido do que o acesso aos dados em blocos de memória remotos.

Arquiteturas Multicore

- Conhecidas como *chip multiprocessing*, são caracterizadas pela existência de diversos *cores* (núcleos) processadores em um mesmo encapsulamento, compartilhando uma memória cache e a memória principal.
- Esse tipo de arquitetura pode ser expandido para vários chips, cada um com vários núcleos, compartilhando uma mesma memória global, no que é chamado multiprocessamento simétrico (SMP).
- A comunicação entre as *threads* é feita através de variáveis na memória compartilhada (ou na memória cache no caso dos multicore), necessitando de intervenção explícita do programador para a coordenação do paralelismo.

Arquiteturas Multicore

- Vários níveis de cache existem dentro da mesma pastilha, com os níveis mais altos (L1 e L2) exclusivos de cada núcleo e o nível mais inferior compartilhado por todos os núcleos.
- Os processadores mais modernos possuem caches de até 36 MB e velocidade de relógio com até 5,0 GHz.

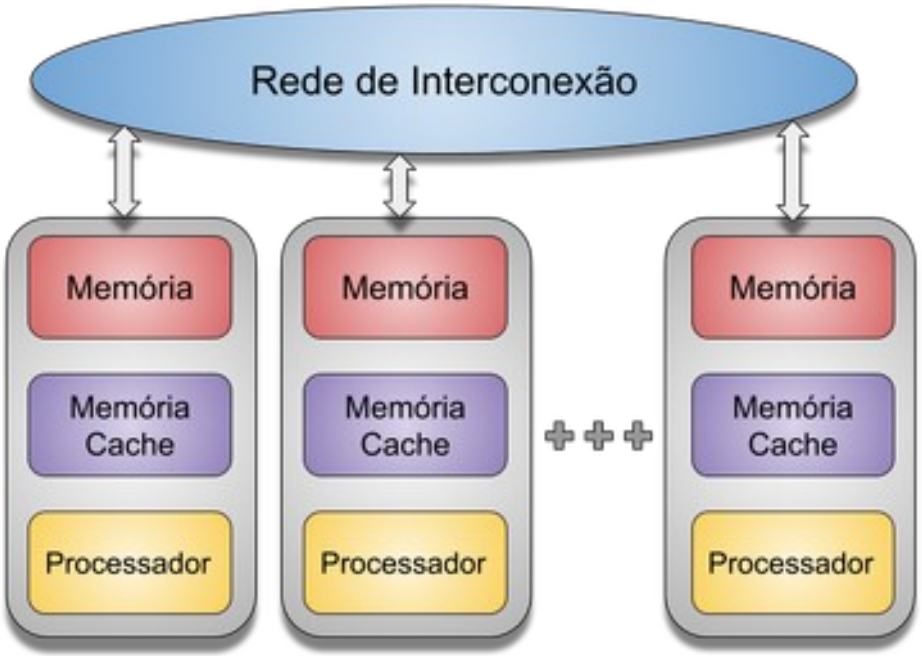


Linguagens e Frameworks

- Linguagem ‘C/C++’ + pThreads: Uma biblioteca padrão para criar e gerenciar threads em sistemas POSIX.
- Linguagem ‘C/C++’ ou Fortran + OpenMP: Uma API que adiciona diretivas ao código C/C++ para especificar paralelismo, gerenciamento de threads e sincronização.
- Java: Possui suporte nativo para multithreading através da biblioteca java.lang.Thread e frameworks como java.util.concurrent.
- Python: Embora seja uma linguagem interpretada, oferece bibliotecas como threading e multiprocessing para programação paralela. No entanto, o GIL (Global Interpreter Lock) limita o paralelismo verdadeiro em código puramente computacional. Existem bibliotecas com NumPy e SciPy para programação científica.

Arquiteturas de Memória Distribuída

- Cada processador enxerga apenas o seu espaço de memória.
- Apresenta como vantagens serem altamente escaláveis e permitirem a construção de processadores maciçamente paralelos.
- Nesse tipo de arquitetura a comunicação entre os processadores é feita através de troca de mensagens.
- A troca de mensagens resolve tanto o problema da comunicação dos processadores como o da sincronização.



Arquiteturas de Memória Distribuída

- As suas grandes desvantagens são a necessidade de se fazer uma boa distribuição de carga entre os processadores, quer seja automaticamente, quer seja manualmente.
- É necessário evitar as situações de *deadlock*, tanto no nível de aplicação como no nível de sistema operacional, possíveis de ocorrer quando há espera circular pelo envio e recebimento de mensagens.
- Além disso, é um modelo de programação menos natural, demandando diversas modificações em relação ao código convencional sequencial.

Programação Paralela

- MPI (Message Passing Interface):
  - API padrão de fato para programação em clusters e supercomputadores.
  - Oferece um conjunto rico de funções para comunicação ponto-a-ponto e coletiva, além de mecanismos de sincronização.
  - Suporta uma ampla variedade de linguagens, incluindo C, C++, Fortran e outras.
- PVM (Parallel Virtual Machine):
  - Uma das primeiras APIs para programação distribuída, ainda utilizada em alguns ambientes.
  - Oferece funcionalidades similares ao MPI, mas com uma sintaxe diferente.
  - Utilizada hoje em dia apenas em contextos educacionais.

Lista de Supercomputadores

- A lista Top500 (https://www.top500.org) relaciona os computadores mais rápidos do mundo.
- Essa lista contém detalhes sobre o desempenho, consumo e características dos computadores mais rápidos da atualidade.
- Detalhes como o tipo de processador, uso de aceleradores, quantidade de memória e rede de interconexão são apresentados.
- Os computadores listados apresentam uma mistura dos conceitos de memória compartilhada e distribuída.

Programação Paralela

Programação Paralela

- A programação paralela cuida da execução coordenada de tarefas para a solução de um problema.
- Apenas existência de vários processos ou *threads* executando simultaneamente não caracteriza a programação paralela.
- Para isso é necessário que os processos ou *threads* realizem atividades coordenadas, comunicando-se e trocando as informações necessárias.
- Essa comunicação pode ser feita através de uma memória compartilhada, acessível a todos os processos e *threads* ou, quando isso não for possível, através de troca de mensagens enviadas por uma rede de comunicação.

Programação por Memória Compartilhada

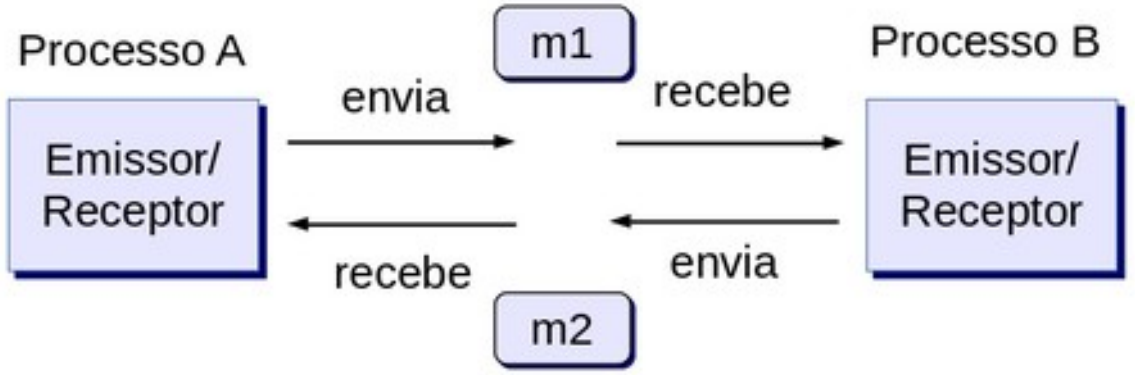
- Neste paradigma, a comunicação entre as tarefas é realizada por meio de variáveis armazenadas em um espaço de memória compartilhado. As tarefas da aplicação podem ser mapeadas em processos ou *threads*.
- No caso de processos, é necessário requisitar ao sistema operacional a definição de um espaço de memória compartilhada por mais de um processo. Este compartilhamento entre processos pode ser complexo e pouco eficiente.
- No caso do uso de *threads*, já existe um espaço de memória compartilhado entre as *threads* no mesmo processo que as criou. Esse acesso é rápido e de baixa complexidade. Por isso, o uso de *threads* é frequente no paradigma de memória compartilhada.

Programação por Memória Compartilhada

- Quando duas ou mais *threads* de um programa precisam trocar informações entre si, o programador define variáveis de acesso compartilhado entre elas.
- Desse modo, quando uma *thread* tem um valor novo para ser enviado para as outras *threads*, esse valor é simplesmente escrito em uma variável compartilhada, independentemente do fato de as demais *threads* estarem esperando por esse valor naquele momento.
- Isso pode gerar um problema de condição de corrida, exigindo o uso de primitivas de sincronização para resolvê-lo.

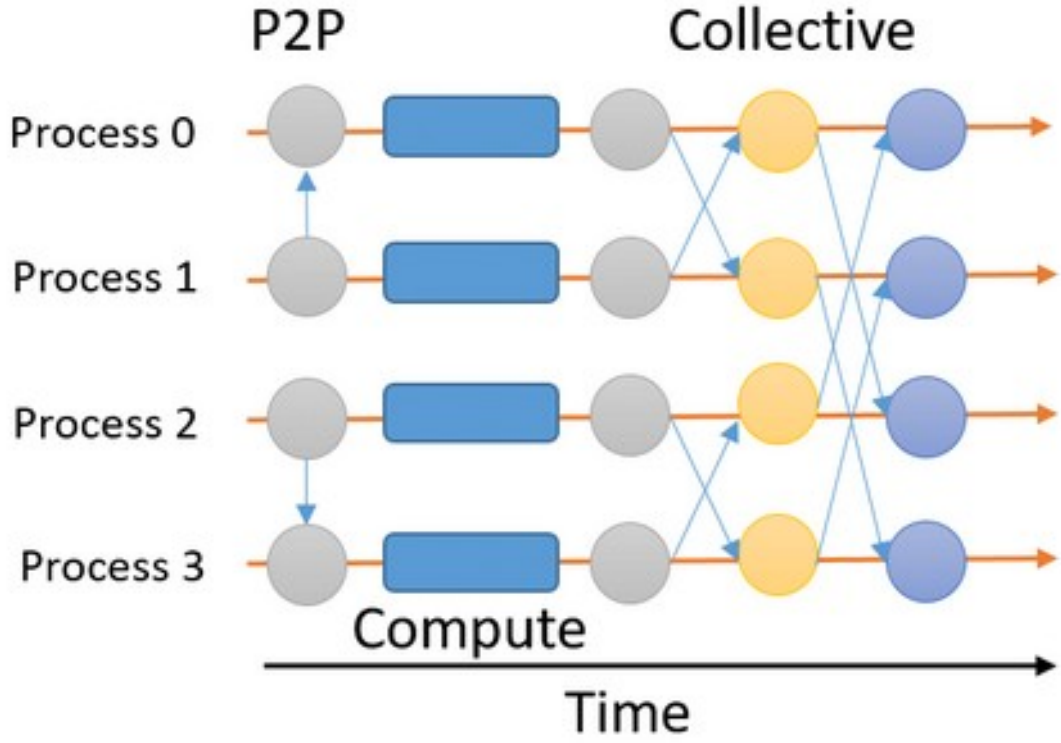
Programação por Troca de Mensagens

- Neste paradigma, a comunicação entre as tarefas da aplicação é feita por meio de rotinas de envio e recebimento de mensagens.
- As tarefas são normalmente mapeadas em processos, cada um com sua memória privada.



Programação por Troca de Mensagens

- A figura ilustra diversos processos em execução, trocando informações para a solução de um problema.
- A comunicação pode ser ponto-a-ponto, envolvendo dois processos, ou coletiva, quando mais de dois processos trocam mensagens.



Programação por Troca de Mensagens

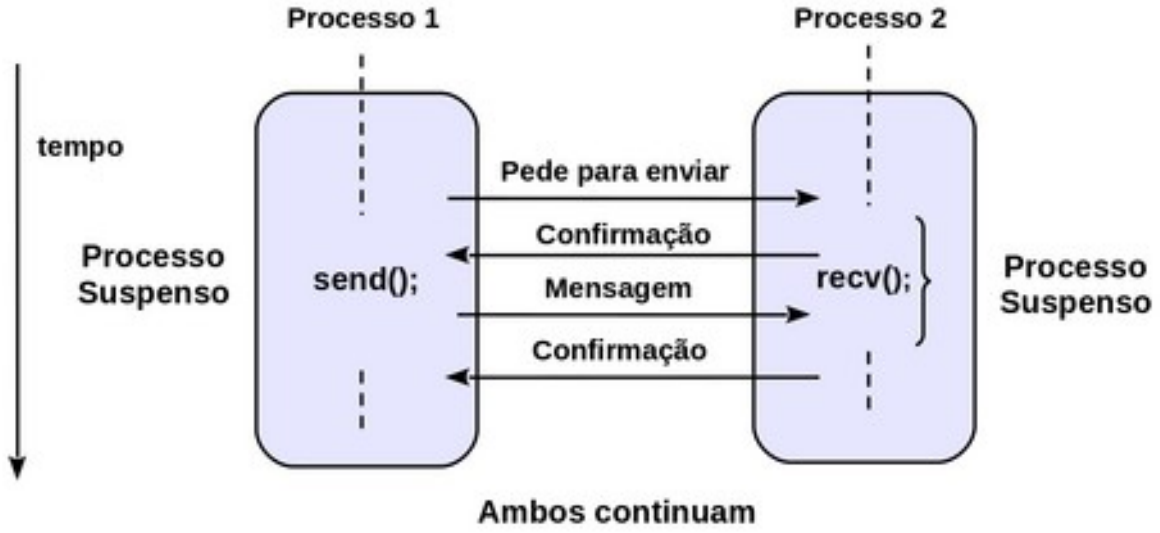
- Como não há um espaço de memória compartilhada, o problema de condição de corrida não ocorre, embora também possa haver deadlock.
- O deadlock acontece quando há um ou mais processos esperando por uma mensagem de um processo, que por sua vez está esperando por uma mensagem a ser enviada pelos demais processos.
- Há dois modos principais de comunicação no paradigma de programação por troca de mensagens: síncrono e assíncrono.

Comunicação Síncrona

- No modo de comunicação síncrona tanto o envio como a recepção de mensagens são bloqueantes.
- Ou seja, a rotina de envio não retorna para o programa principal enquanto a mensagem não for completamente enviada.
- E a função de recepção também não retorna enquanto a mensagem não estiver disponível para o usuário no espaço de memória reservado na aplicação para isso.
- Ao se concluir a operação de envio, as estruturas de dados utilizadas no programa do usuário para o envio da mensagem podem ser reutilizadas com segurança.
- Ao término da operação de recepção temos a certeza que os dados recebidos estão disponíveis para a aplicação.

Comunicação Síncrona

- Esse modelo de comunicação requer, portanto, um protocolo de quatro fases para a sua implementação:
  - pedido de autorização para transmitir;
  - recebimento da autorização;
  - transmissão da mensagem;
  - recebimento da mensagem de reconhecimento.



Comunicação Síncrona

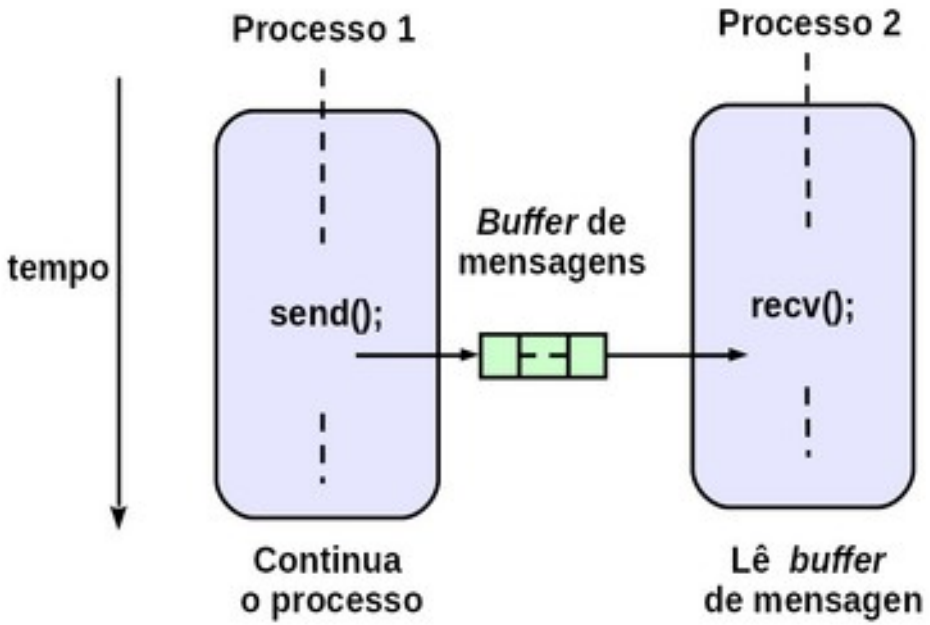
- Esse modo de comunicação é simples e seguro.
- Também permite a sincronização entre os processos, ou seja, quando um processo acaba de transmitir uma mensagem, sabe que o processo de destino acabou de receber essa mensagem.
- Esse modo, contudo, elimina a possibilidade de haver superposição entre a computação e a transmissão das mensagens, diminuindo o desempenho da aplicação paralela.

Comunicação Assíncrona

- A principal característica da comunicação assíncrona é a desvinculação do instante de tempo no qual ocorrem as operações de envio e recepção da mensagem.
- Isso pode ser feito com o uso de áreas (*buffers*) destinadas ao armazenamento da mensagem enviada.
- A computação prossegue, enquanto a transmissão dos dados é realizada através da rede de comunicação.
- A operação de envio (*send)* se completa a partir da cópia de todo o conteúdo da mensagem do espaço de endereçamento do usuário para uma área correspondente (*buffer*) da biblioteca ou do sistema operacional.

Comunicação Assíncrona

- Isso libera as estruturas de dados do programa do usuário para serem utilizadas por novas operações de envio de mensagens.
- Há outras formas de implementação da comunicação assíncrona, como por exemplo o uso de rotinas de envio e/ou recepção não-bloqueantes.



Comunicação Assíncrona

- Quando a operação de envio é não-bloqueante, ela retorna imediatamente, dando apenas início ao processo de transmissão da mensagem.
- Isso nada garante e não autoriza a reutilização das estruturas de dados do programa para o envio de uma nova mensagem.
- Quando a operação de recepção é não-bloqueante, ela apenas indica a intenção de realizar uma operação de recepção, retornando também imediatamente, e nada se garante em relação aos dados recebidos já estarem disponíveis para uso pela aplicação do usuário.

Comunicação Assíncrona

- Tanto no envio como na recepção, ainda é necessário uma segunda fase, onde é verificado se as operações de troca de mensagem iniciadas anteriormente pelas primitivas de envio e recepção já foram concluídas.
- Somente após essa verificação é possível se alterar com segurança a área de dados utilizada pelo processo emissor e se ter certeza que os dados recebidos estão disponíveis para uso no processo receptor da mensagem.
- O modo de comunicação assíncrono permite uma maior superposição no tempo entre o processamento da aplicação e a transmissão das mensagens, aumentando o desempenho da aplicação paralela.

Avaliação de Desempenho

Avaliação de Desempenho

- Ao desenvolver uma aplicação paralela, é fundamental avaliar seu desempenho com precisão.
- Além de obter tempos de execução melhores que a versão sequencial, é necessário considerar métricas que indiquem se os recursos disponíveis estão sendo utilizados de forma eficiente.
- Em outras palavras:
  - A aplicação paralela está sendo executada de maneira eficiente?
  - É possível melhorar seus tempos de execução?
  - Onde estão os gargalos que impedem essa melhoria?

Avaliação de Desempenho

- A avaliação de desempenho da aplicação paralela, pode ser feita com métricas muito simples:
  - *speedup* – também conhecido como ganho de desempenho ou aceleração;
  - eficiência;
  - escalabilidade.
- Diversos fatores influenciam essas métricas, tais como o algoritmo paralelo utilizado; os custos de sincronização e comunicação; a forma de distribuição das tarefas; e o percentual do programa que efetivamente pode ser paralelizado.

Speed-up (Aceleração)

- O *speedup*, ou aceleração, mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação sequencialmente, com um único processador – T(1), e o tempo gasto na execução com P processadores – T(P), conforme a equação a seguir:

$$S(P) = \frac{T(1)}{T(P)} \qquad (2.1)$$

- Por exemplo, se o tempo de execução sequencial T(1) foi 10 segundos, e o tempo de execução paralelo T(P) foi 1 segundo, o *speed-up* S(P) é 10.

Eficiência

- A eficiência mede o quão eficaz é a adição de novos processadores para auxiliar na resolução de um problema.
- Seu valor é calculado pela razão entre o speedup S(P) e o número de processadores P utilizados para alcançar esse *speedup*, conforme mostra a equação a seguir:

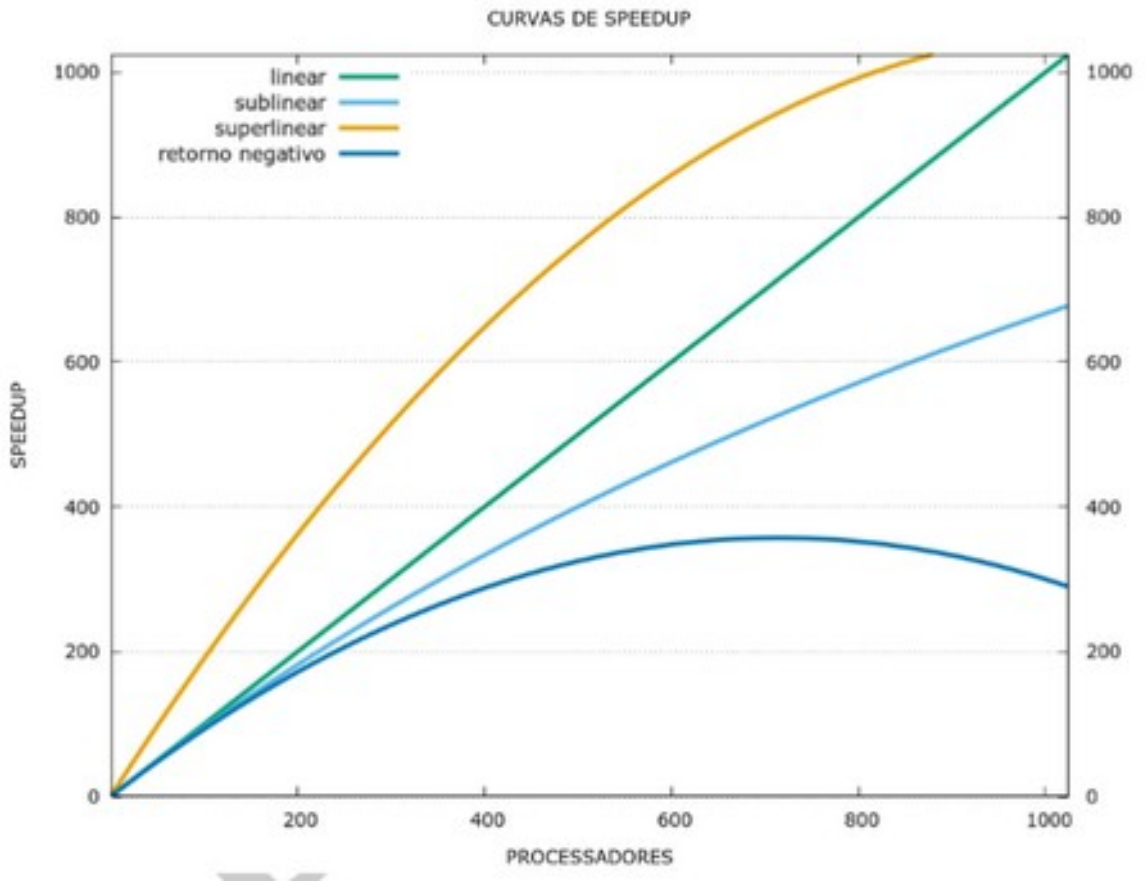
$$E(P) = \frac{S(P)}{P} \qquad (2.3)$$

Eficiência

- Então, considerando o exemplo anterior, se o *speedup* igual a 10 foi obtido com 10 processadores, a eficiência será igual a 1, o que é um valor ótimo.
- Contudo, se o mesmo speedup foi obtido com o uso de 20 processadores, então a eficiência é igual a 0,5, o que pode ser considerado um valor baixo.
- Normalmente, o valor da eficiência é menor do que 1 devido aos seguintes fatores, já elencados:
  - O percentual de código paralelizável na aplicação;
  - A distribuição de carga entre os processadores;
  - A quantidade de comunicação e sincronização da aplicação.

Speedup vs Eficiência

- O speedup pode apresentar os comportamentos da figura ao lado, de acordo com os valores de eficiência:
  - Superlinear E → E > 1
  - Linear → E = 1
  - Sublinear →E < 1
  - Retorno negativo → E < 0



Speedup versus Eficiência

- A eficiência pode ser interpretada graficamente como a inclinação da curva de *speedup*.
- Em condições ideais, a inclinação da curva será de 45º , ou seja, o *speedup* será sempre igual a P, onde P indica o número de processadores em uso. Isso caracteriza o speedup linear, no qual a eficiência é igual a 1.
- Entretanto, o speedup sublinear é o caso mais comum, ocorrendo devido a ineficiências no processo de paralelização dos programas já mencionados:
  - O percentual de código que pode ser paralelizado na aplicação;
  - A distribuição desigual de carga entre os processadores;
  - A quantidade de comunicação e sincronização necessária na aplicação.

Speedup versus Eficiência

- O *speedup* superlinear ocorre apenas em situações muito específicas, tal como, quando a paralelização altera a distribuição dos dados na hierarquia de memória.
- Isso pode permitir, por exemplo, que os dados de cada processador caibam inteiramente na memória cache, em vez de precisarem ser buscados na memória principal.
- O retorno negativo ocorre quando a adição de mais processadores diminui o *speedup*, aumentando o tempo de execução da aplicação, possivelmente por conta de um aumento excessivo nos custos de comunicação e/ou de sincronização.

Escalabilidade

- Uma outra métrica importante é a escalabilidade. Ou seja, como o meu programa se comporta conforme aumenta o número de processadores envolvidos na computação.
- Um sistema é considerado escalável quando sua eficiência se mantém constante à medida que o número de processadores P aplicados à solução do problema aumenta.
- Quando o tamanho do problema é mantido constante e o número de processadores cresce, dois fenômenos acontecem: o *overhead* de comunicação tende a aumentar, resultando na diminuição da eficiência; e granularidade de cada tarefa executada diminui.

Escalabilidade

- Na prática, uma análise de escalabilidade deve levar em conta a possibilidade de aumentar proporcionalmente o tamanho do problema a ser resolvido conforme P cresce, de modo a compensar o aumento natural da sobrecarga de comunicação e sincronização.
- Considere, por exemplo, um problema de tamanho S. Utilizando P processadores, esse problema leva um tempo T para ser executado. O sistema é considerado escalável se um problema de tamanho 2S executado em 2P processadores, também levar o mesmo tempo T.
- Assim, se esses cuidados forem tomados, uma análise mais adequada do sistema pode ser realizada.

MPI

MPI

- É um padrão de troca de mensagens portátil que facilita o desenvolvimento de aplicações paralelas.
- Usa o paradigma de programação paralela por troca de mensagens e pode ser usado tanto em *clusters* como em sistemas de memória compartilhada.
- É uma biblioteca de funções utilizável com programas escritos em C, C++ ou Fortran.
- O MPI foi fortemente influenciado pelo trabalho no IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex e PARMACS. Outras contribuições importantes vieram do Zipcode, Chimp, PVM, Chameleon e PICL.

Características

- Suporte para comunicação entre processos ponto a ponto ou coletiva.
- Suporte para vários modelos de comunicação, incluindo comunicação síncrona e assíncrona.
- Suporte para vários tipos de topologias de rede virtuais, incluindo anel, árvore e malha.
- Suporte para tipos de dados básicos e estruturados.
- Suporte para operações de comunicação coletiva como difusão, redução e dispersão.
- Portabilidade entre diferentes arquiteturas de computação paralela e sistemas operacionais.

Características

- O MPI pode ser combinado com outros modelos de programação.
- Assim, podemos combinar MPI com OpenMP, que apresenta vantagens em alguns casos, como por exemplo:
  - Códigos com escalabilidade MPI limitada, quer seja pelo algoritmo ou pelas rotinas de comunicação coletiva utilizadas.
  - Códigos limitado pelo tamanho de memória, tendo muitos dados replicados em cada processo MPI.
  - Códigos com problemas de desempenho pela ineficiência da implementação da comunicação intra-nó em MPI.
- O MPI também pode ser combinado com OpenMP, OpenACC ou CUDA para uso de aceleradores.

Linguagens Suportadas

- C, C++ e Fortran: O MPI foi originalmente desenvolvido para ser usado com o C, C++ e Fortran.
- Python: Existem várias bibliotecas MPI para Python, como mpi4py e pyMPI, que permitem que os programas Python se comuniquem usando MPI.
- Java: Existe uma biblioteca MPI para Java chamada MPJ Express, que permite que os programas Java se comuniquem usando MPI.
- Outras linguagens: Além das linguagens mencionadas acima, também existem bibliotecas MPI para outras linguagens como Perl, Ruby, Lua, entre outras.

Histórico de versões

- MPI-1.1: primeira versão funcional lançada em 1998.
- MP1-1.3: documento final que consolida a versão 1.0 do MPI, lançada apenas em 2008.
- MPI-2.1: lançada oficialmente em 2008 como um livro com diversos exemplos e orientações para os usuários.
- MPI-2.2: lançada em 2009. A última versão desta série.
- MPI-3.1: a versão final do padrão 3.0, foi lançada em 2015.
- MPI 4.0: a versão mais recente do padrão foi lançada em 2021.
- Todas as versões estão disponíveis em https://www.mpi-forum.org/docs/

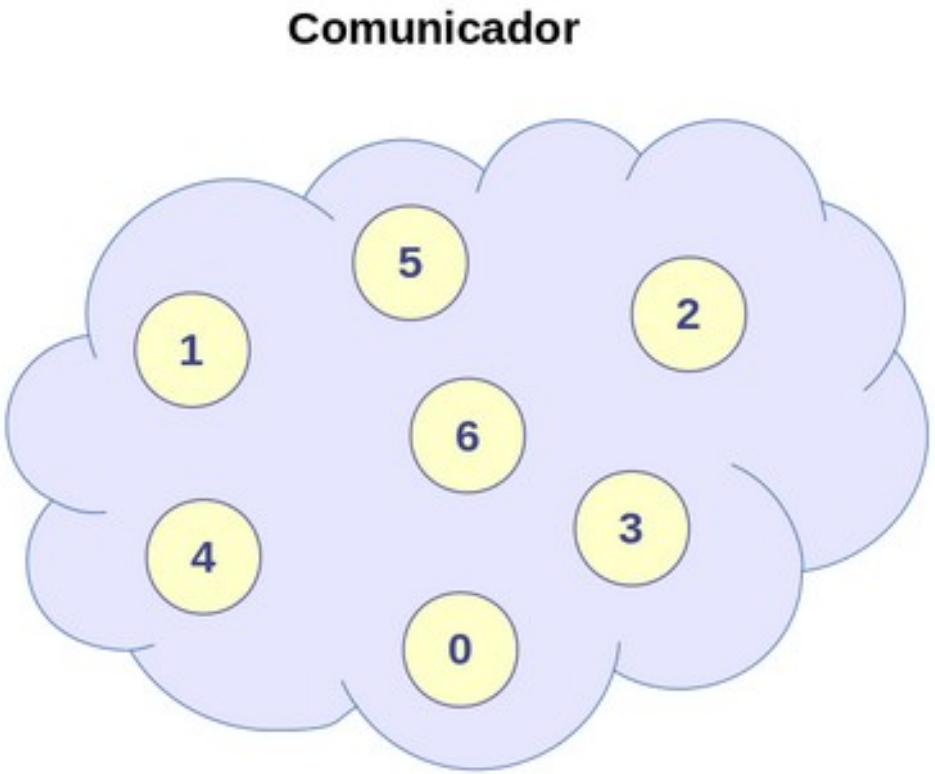
Objetivos

- Um dos objetivos do MPI é oferecer possibilidade de uma implementação eficiente da comunicação:
  - Evitando cópias de memória para memória.
  - Permitindo superposição de comunicação e computação.
- Permitir implementações em ambientes heterogêneos.
- Supõe-se que a interface de comunicação é confiável:
  - Falhas de comunicação devem ser tratadas pelo subsistema de comunicação da plataforma.

Comunicadores

- A biblioteca MPI trabalha com o conceito de comunicadores para definir o universo de processos envolvidos em uma operação de comunicação, através dos atributos de grupo e contexto:
  - Dois processos que pertencem a um mesmo grupo e usando um mesmo contexto podem se comunicar diretamente.
  - O comunicador padrão recebe o nome de MPI_COMM_WORLD e contém todos os processos que iniciados na execução de um programa.
  - Cada processo possui um identificador único chamado de ranque (rank), que vai de 0 até P-1, onde P é o número de processos em um comunicador.

Comunicadores



Comunicadores

- Comunicadores são utilizados para definir o universo de processos envolvidos em uma operação de comunicação através dos seguintes atributos:
  - Grupo: Conjunto ordenado de processos. A ordem de um processo no grupo é chamada de ranque.
  - Contexto: tag definido pelo sistema.
- Dois processos pertencentes a um mesmo grupo e usando um mesmo contexto podem se comunicar.
- O comunicador padrão MPI_COMM_WORLD contém todos os processos existentes no início da execução de um programa.