ERAD SP 2013

IV Escola Regional de
Alto Desempenho de São Paulo
São Carlos / SP

Minicurso

Programando para múltiplos processadores: Pthreads, OpenMP e MPI

Liria Matsumoto Sato (EP/USP) e
Helio Crestana Guardia (DC/UFSCar)

Necessidade de Poder Computacional

Demanda crescente por poder computacional:

- Desejo de resolver **novos** problemas
- Necessidade de resolver problemas computacionais já tratados:
  - com **melhor precisão**
  - de maneira **mais rápida**

Problemas computacionais

Aplicações atuais em muitos casos envolvem a manipulação de **grandes volumes de dados**, o que requer processamento extensivo.

Exemplos:

- Bancos de dados paralelos e *data mining*
- Processamento de imagens e sinais
- Máquinas de busca na Web
- Serviços baseados na Web
- Suporte para diagnósticos auxiliados por computador
- Gerenciamento de grandes sistemas de informação
- Computação gráfica e realidade virtual
- Suporte para tecnologias multimídia
- Ambientes para trabalho cooperativo
- …
- Jogos 3D :-)

Grand Challenge Problems

"A grand challenge is a fundamental problem in science or engineering, with broad applications, whose solution would be enabled by the application of high performance computing resources that could become available in the near future."

Examples:

- *Computational fluid dynamics for the design of hypersonic aircraft, efficient automobile bodies, and extremely quiet submarines, weather forecasting for short and long term effects, efficient recovery of oil, and for many other applications;*
- *Electronic structure calculations for the design of new materials such as chemical catalysts, immunological agents, and superconductors;*
- *Plasma dynamics for fusion energy technology and for safe and efficient military technology;*
- *Calculations to understand the fundamental nature of matter, including quantum chromodynamics and condensed matter theory;*
- *Symbolic computations including*
  - *speech recognition,*
  - *computer vision,*
  - *natural language understanding,*
  - *automated reasoning, and*
  - *tools for design, manufacturing, and simulation of complex systems.*

http://en.wikipedia.org/wiki/Grand_Challenge

Limites e tendências

- **Limitações físicas** dificultam criar computadores mais rápidos:
  - **Transmissão de dados**
  - **Miniaturização**
  - **Economia**: custo para desenvolver processador mais rápido é cada vez maior.
- **Vantagens** do paralelismo:
  - Processadores **comerciais** oferecem desempenho cada vez melhor, incluindo múltiplos processadores no mesmo ***chip*** (*multi-core*) e suporte para execução simultânea de várias atividades (*Hiper-Threading*).
  - **Redes rápidas**: tecnologias de rede oferecem interligação da ordem de Gbps em preços acessíveis para grupos de máquinas.
- Uso de **N** processadores comuns **interligados** é mais **barato** que 1 processador **N** vezes mais rápido com **mesmo desempenho**.

Computação Paralela

Problemas computacionais complexos normalmente apresentam as seguintes características:

- Podem ser **divididos** em partes distintas que podem ser executadas simultaneamente
- Podem ter **diversas instruções** sendo executadas ao mesmo tempo
- São executados em **menor tempo** quando **múltiplos** recursos computacionais são utilizados

Programação Paralela

- Programação paralela trata da programação de **múltiplos computadores**, ou de computadores com **múltiplos processadores**, para resolver um problema mais rapidamente do que é possível com um único processador.
- **Ideia**: resolução de um problema normalmente pode ser **dividida** em atividades menores, que podem ser executadas simultaneamente com alguma coordenação.
- Motivos:
  - Tratar problemas maiores, que requerem mais **processamento** ou mais **memória** que a normalmente disponível em um único sistema.
  - Prover tolerância a falhas, ...
- **Questão**: ***N*** processadores trabalhando de maneira simultânea podem obter um resultado ***N*** vezes mais depressa?

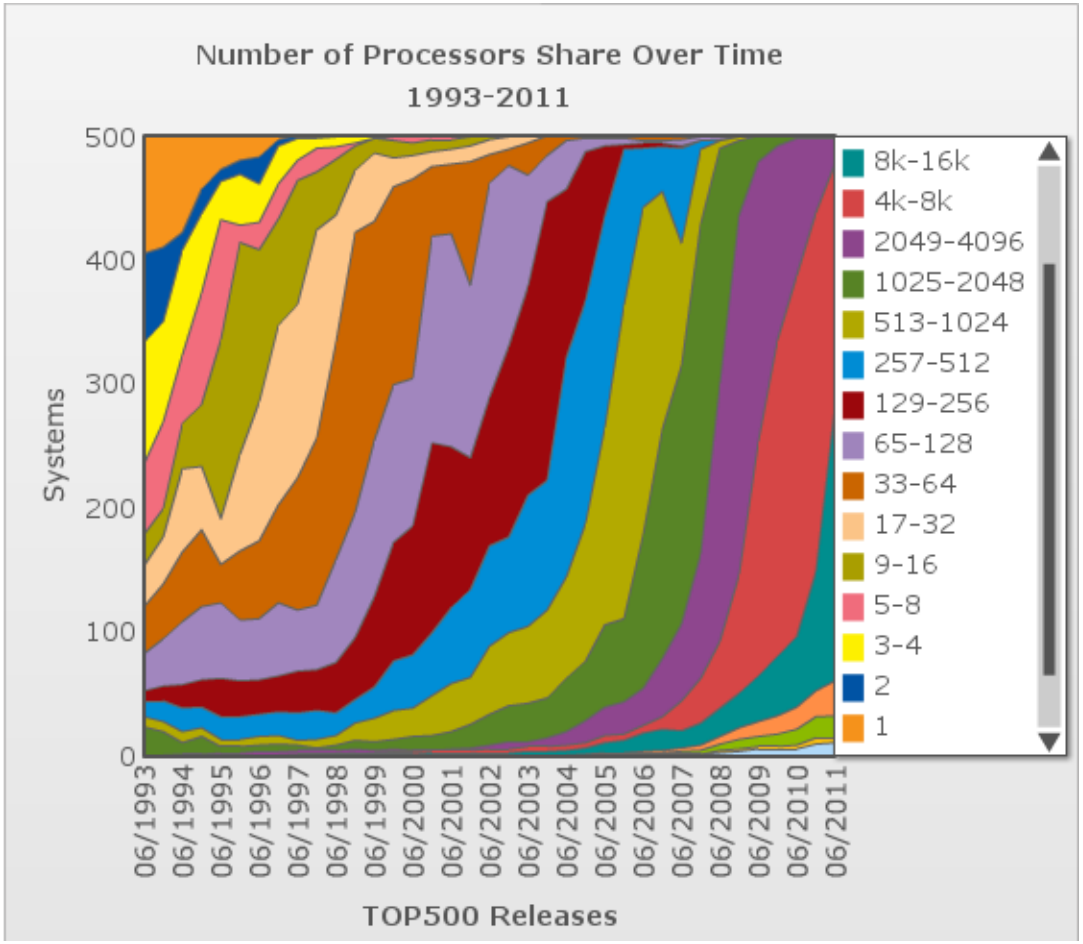
Programação Paralela

- **Decomposição** dos programas ou dos dados
- **Distribuição** dos programas ou dados
- **Coordenação** do processamento e das comunicações

- Aspectos:
  - Arquitetura paralela
  - Formas de comunicação

www.top500.org

Histórico dos maiores sistemas computacionais [*www.top500.org*]:



Tipos de Computadores Paralelos

- Multi**processadores** com memória compartilhada
- Multi**computadores** com memória distribuída

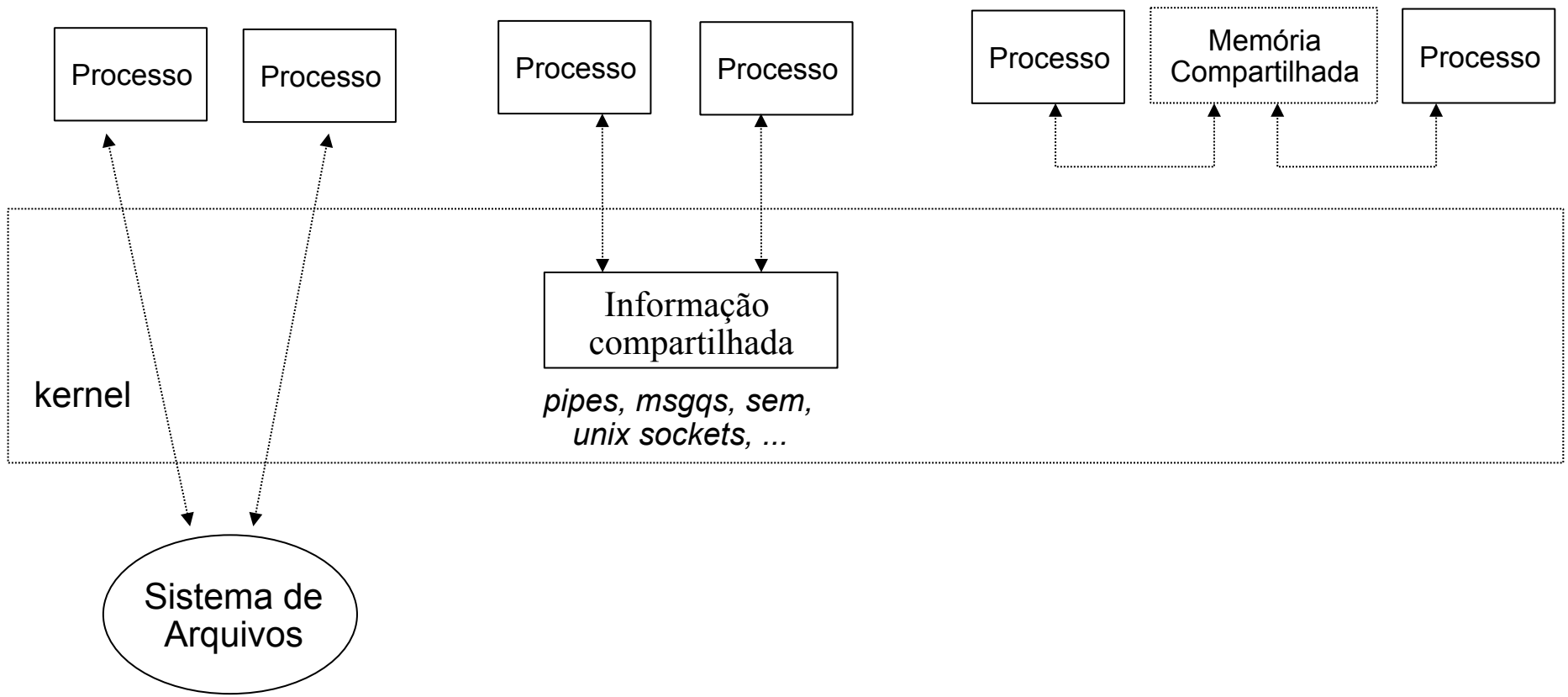
Programação para multicore

- Computadores paralelos com múltiplos processadores ou núcleos
- **Memória** fisicamente **compartilhada** entre os processadores

Processos

- **Criação: *fork( )***
  - Cópia de memória de pai para filho
  - Duplicação dos descritores
  - *Copy-on-write* pode atrasar cópia até que operação de escrita
  - ***vfork***( ) não copia dados, aguardando ***exec***( )
  - Mecanismos de comunicação (IPC) são usados para passar informações entre pai e filho(s)
- **Comunicação e sincronização:**
  - **Arquivos**: arquivos comuns e *mmap'ed*
  - **BSD IPC**: *sockets* (UNIX e INET), *pipes*
  - **System V IPC**: *shared memory,* semáforos, filas de mensagem

Compartilhamento de Informações entre processos



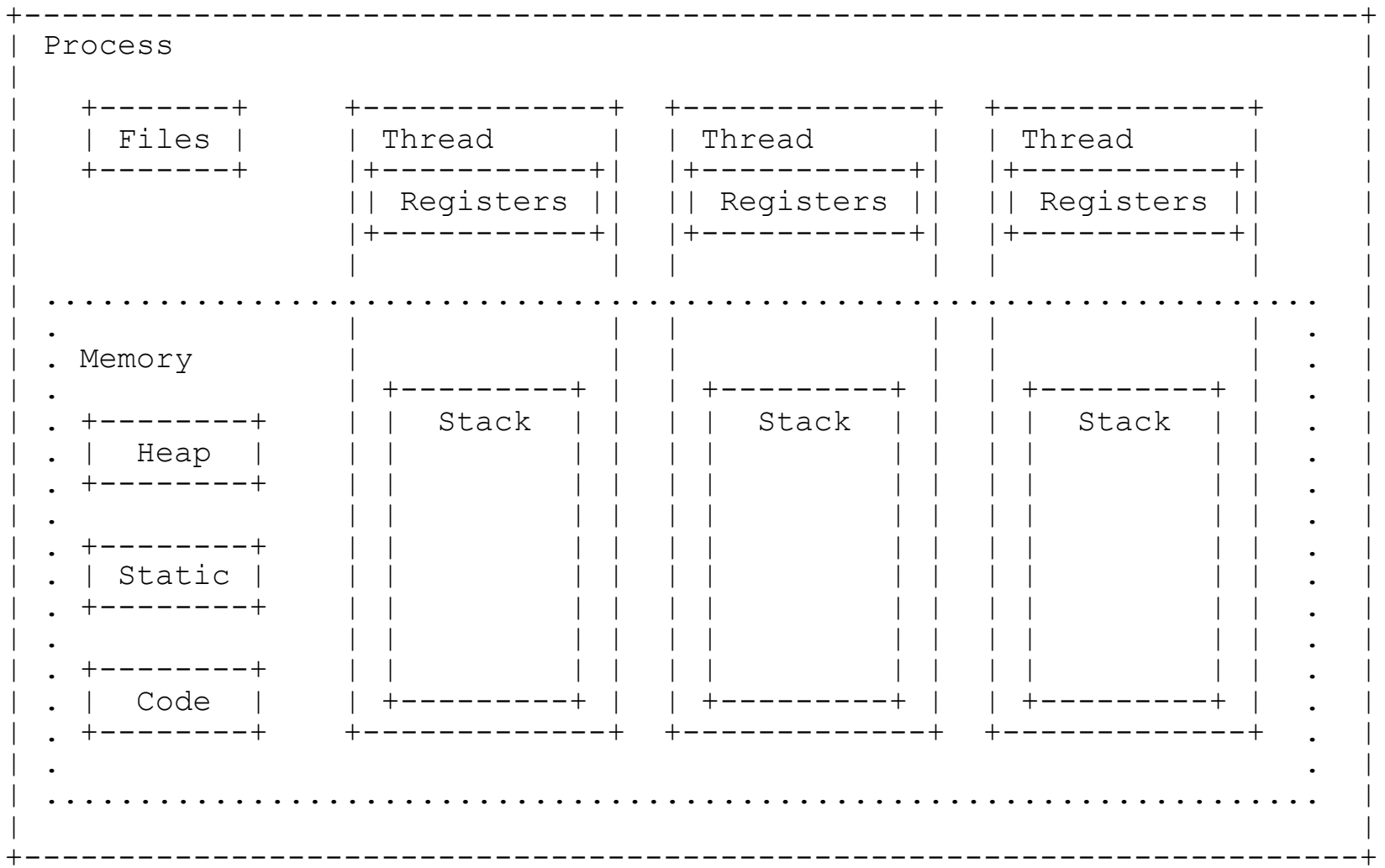
Threads

- Em ambientes multiprocessados com **memória compartilhada,** como SMPs, ***threads*** podem ser usadas para **execução paralela**
- Uma ***thread*** pode ser definida como um **fluxo de instruções** (**função**) independente, **associado a um processo**
- Processos contêm ao menos uma *thread*, iniciada com a função ***main()***
- Em um programa ***multi-threaded***, diversas funções podem ser selecionadas para execução simultânea
- Execução simultânea é possibilitada pela replicação de alguns recursos para cada *thread*
- *Threads* compartilham a **mesma memória global** (dados e *heap*), mas cada uma possui sua própria **pilha** (alocação de variáveis locais da função, etc.)

Threads: implementações

- Padrão IEEE POSIX 1003.1c (POSIX.1): **interface** comum para manipulação de *threads*, chamada ***POSIX threads***, ou ***Pthreads***

Threads: Recursos



(From the DECthreads manual)

Threads: dados compatilhados

POSIX.1 especifica que *threads* de um processo devem compartilhar os seguintes atributos:

- *process ID*
- *parent process ID*
- *process group ID and session ID*
- *controlling terminal*
- *user and group IDs*
- *open file descriptors*
- *record locks (see fcntl(2))*
- *signal dispositions*
- *file mode creation mask (umask(2))*
- *current directory (chdir(2)) and root directory (chroot(2))*
- *interval timers (setitimer(2)) and POSIX timers (timer_create())*
- *nice value (setpriority(2))*
- *resource limits (setrlimit(2))*
- *measurements of the consumption of CPU time (times(2)) and resources (getrusage(2))*

http://pubs.opengroup.org/onlinepubs/000095399/basedefs/xbd_chap03.html

3.289 Process

An address space with one or more threads executing within that address space, and the required system resources for those threads.

Note: *Many of the system resources defined by IEEE Std 1003.1-2001 are shared among all of the threads within a process. These include the process ID, the parent process ID, process group ID, session membership, real, effective, and saved set-user-ID, real, effective, and saved set-group-ID, supplementary group IDs, current working directory, root directory, file mode creation mask, and file descriptors.*

Threads: dados replicados

Além da **pilha** (***stack***), POSIX.1 determina que os seguintes atributos devem ser distintos para cada *thread*:

- *thread ID (the pthread_t data type)*
- *signal mask (pthread_sigmask())*
- *the errno variable*
- *alternate signal stack (sigaltstack(2))*
- *real-time scheduling policy and priority (sched_setscheduler(2) and sched_setparam(2))*

No **Linux**, os seguintes atributos também são mantidos **por *thread***:

- *capabilities (capabilities(7))*
- *CPU affinity (sched_setaffinity(2))*

Threads: Aspectos

- *Threads* são associadas a um processo, do qual compartilham recursos:
  - Alterações globais são visíveis por todas as *threads* (manipulação de arquivos, e.g.)
  - **Ponteiros iguais** apontam para a **mesma área** de memória
  - Acessos simultâneos à mesma área de memória são possíveis e requerem **sincronização**
  - *Threads* de um processo devem ser alocadas na mesma memória, normalmente no mesmo processador

Threads: Aspectos

Reentrância de código

- **Funções reentrantes** apresentam comportamento correto mesmo se chamadas simultaneamente por diversas *threads*
- Funções que manipulam **informações globais** do processo (memória, arquivos, etc.) devem ser cuidadosamente projetadas para garantir a reentrância de código
- Abordagens para prover reentrância:
  - passagem de parâmetros
  - uso de dados internos da *thread* (*thread-specific*), alocados na pilha

Thread-safety

- ***Thread-safety*** está relacionada com **condições de disputa** (***race conditions***) e com o comprometimento de dados globais do processo, produzindo **resultados incorretos** ou **imprevisíveis** em função da **ordem** em que *threads* são executadas
- **Função** é ***thread-safe*** quando seu comportamento é correto mesmo quando executada simultaneamente por diversas *threads*
  - Garantia é normalmente obtida **encapsulando** (***wrapping***) a função original em uma nova, que utiliza um mecanismo de controle de acesso (***mutex***)
  - Funções e chamadas de sistema são ***non-thread-safe*** a menos que atestado o contrário

Asynchronous-safe

- Função ***asynchronous-safe*** é aquela que pode ser usada em tratadores de sinal (***signal handlers***)

Aspectos da Lib C

- *Reentreant Routines: a function whose effect, when called by two or more threads, is guarenteed to be as if the threads each executed the function one after another in an undefined order, even if the actual execution is interleaved.*
- *All routines which use static data (**strtok**, **ctime**, **gethostbyname**) are nonreentrant*
- *Non reentrant routines have a _r reentrant counterpart*
- *Some implementation reimplement functions that return static data to use thread specific data.*
- *Errno*
  - *The variable errno is now per thread*
  - *Declaring errno extern int is no longer valid*
  - *Must include errno.h to get the correct declaration*
- *Stdio*
  - *All stdio calls inherintly lock the FILE*
  - *A recursive locking mechanizim is provided to lock accross calls.*
  - *For speed previous macro routines have a new name*

- *void flockfile(FILE * file);*
- *int ftrylockfile(FILE * file);*
- *void funlockfile(FILE * file);*
- *int getc_unlocked(FILE * file);*
- *int getchar_unlocked(void);*
- *int putc_unlocked(FILE * file);*
- *int putchar_unlocked(void);*

Porque usar *Threads*

Velocidade:

- Comunicação entre *threads* é mais rápida, baseada em memória compartilhada

Economia de recursos:

- Compartilhamento de recursos favorece economia de memória e diminui tempo das operações de criação de *threads* em relação aos processos

***Responsiveness* (Interatividade)**:

- Uso de múltiplas *threads* em aplicação interativa permite que programa continue respondendo, mesmo que parte esteja bloqueada ou realizando operação demorada

Desempenho de E/S (*I/O throughput*):

- E/S tradicional bloqueia processo
- Com *threads*, apenas a *thread* responsável pela operação é bloqueada, permitindo a sobreposição de processamento com E/S

Portabilidade:

- POSIX *threads* são portáveis

Quando usar *Threads*

- Para poder beneficiar-se do uso de *threads*, programas devem ser organizados em **tarefas independentes**, que podem ser executadas de maneira **concorrente**
- Aspectos das **aplicações** que favorecem o uso de *threads*:
  - **Bloqueio** por possivelmente longos períodos de tempo
  - Uso de muitos **ciclos** de CPU
  - Possibilidade de sobrepor processamento e E/S
  - Tratamento de **eventos assíncronos**
    - Espera por eventos (aplicações de tempo real, e.g.)
    - **Interação** (servidores de rede, e.g.)
    - Uso da função ***select( )***
  - Possuem **prioridades** incomuns (maiores ou menores)
  - Podem ser executadas em **paralelo** com outras tarefas

Modelos de Programação

- ***Manager/worker*** (**mestre/escravo**):
  - *Thread* **mestre** (***manager***) atribui operações para outras *threads* **escravas**
  - **Mestre** normalmente trata operações de **entrada de dados** (solicitações), distribuindo o serviço entre os demais
  - **Escravos** podem ser criados estática ou dinamicamente
- ***Pipeline***:
  - Tarefa é quebrada em suboperações, executadas em série, mas de maneira **concorrente**, por *threads* diferentes
- ***Peer***:
  - Semelhante ao modelo mestre/escravo mas, depois de criar os escravos, *thread* principal participa na execução do trabalho

Paralelização de Algoritmos

- Projeto de algoritmos paralelos é tradicionalmente feito de maneira **manual**
- Programador é responsável por **identificar** e **implementar** o paralelismo
- Paralelização é um processo **iterativo**, **complexo** e sujeito a **erros**.
- Ferramentas podem auxiliar na extração de paralelismo de programas sequenciais:
  - **Compiladores paralelizantes**
  - **Pré-processadores paralelizantes**

Paralelização Automática

Compiladores paralelizantes usam 2 abordagens:

- Completamente **automatizada**
  - Compilador analisa o código fonte e identifica oportunidades para paralelização
  - Análise inclui a identificação de fatores de inibição do paralelismo e uma avaliação dos benefícios da paralelização do aumento do desempenho
  - *Loops* (*do*, *for*) são os principais candidatos a paralelização automática
- **Controlada** pelo programador
  - Diretivas de compilação permitem ao programador explicitar como (e onde) paralelizar a aplicação, podendo ser combinadas com mecanismos de paralelização automática

Restrições da paralelização **automática**:

- Possibilidade de geração de resultados incorretos
- Possibilidade de perda de desempenho
- Pouca flexibilidade na paralelização
- Limitado a trechos de código, tipicamente *loops*
- Pode não paralelizar, caso análise detecte dependência de dados ou código seja muito complexo

Paralelização de Algoritmos

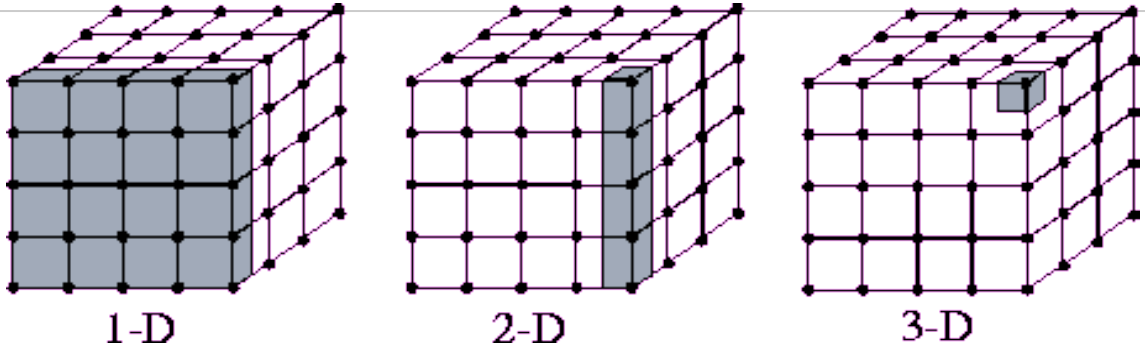
Decomposição de programas para paralelização usa 2 abordagens principais:

- ***Domain decomposition*** (**decomposição de domínio** ou **paralelização de dados**):
  - **Dados** associados ao problema são particionados e mapeados aos processadores
  - Cada tarefa executa **mesmo** conjunto de instruções sobre **dados distintos**
  - Localidade dos acessos é considerada
- ***Functional decomposition*** (**decomposição funcional** ou **paralelismo funcional**):
  - Atividades a serem realizadas são divididas em blocos funcionais
  - Tarefas manipulam **diferentes** partes dos dados ou mesmos dados

Domain Decomposition

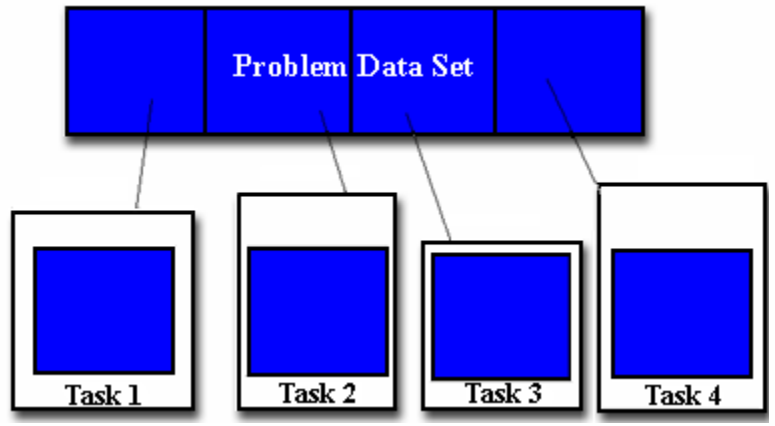
- Objetivo é particionar os **dados** associados com um problema.
- É o mecanismo adotado pela maioria das aplicações paralelas.
- Preferencialmente, dados são decompostos em **pequenos** pedaços de **mesmo** tamanho.
- **Computações** são particionadas a seguir, associando-as aos dados que irão manipular.
- Particionamento resulta em conjunto de tarefas, cada uma contendo parte dos dados e um conjunto de operações que deverão ser realizadas sobre eles.
- Operações podem requerer dados de diversas tarefas, impondo a necessidade de mecanismos de **comunicação** para transferências entre elas.
- Diferentes particionamentos podem ser possíveis, baseados em diferentes estruturas de dados. Abordagens principais consistem em:
  - focar na **maior estrutura** de dados, ou
  - na estrutura que é acessada **mais frequentemente**.
- Diferentes fases do processamento podem manipular diferentes estruturas de dados, ou demandar decomposições diferentes para as mesmas estruturas de dados.
  - Nesse caso, cada fase é tratada separadamente e depois determina-se como as decomposições e algoritmos paralelos são relacionados.

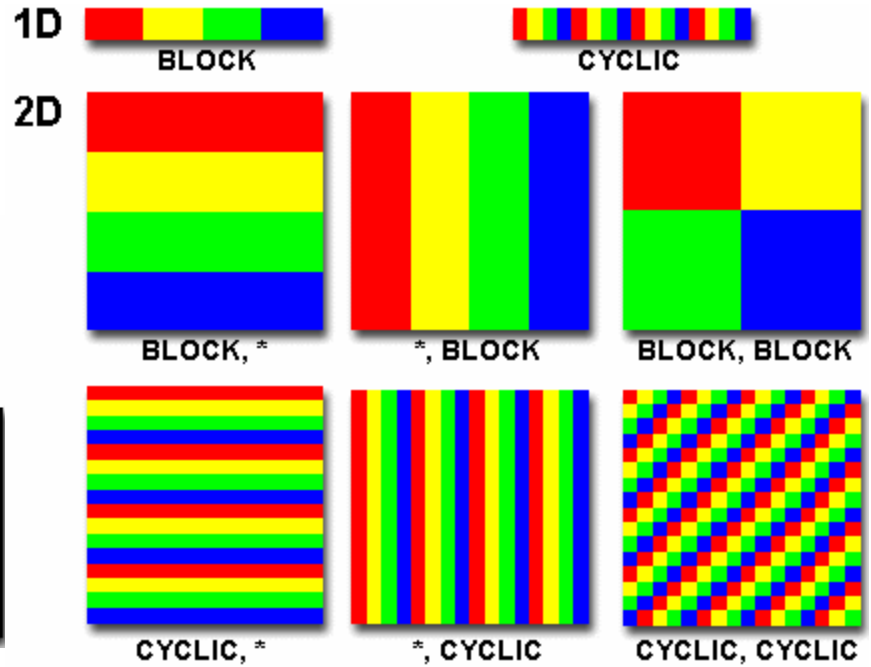
Domain Decomposition



Decomposição de dados de estrutura em 3 dimensões.

http://www-unix.mcs.anl.gov/dbpp



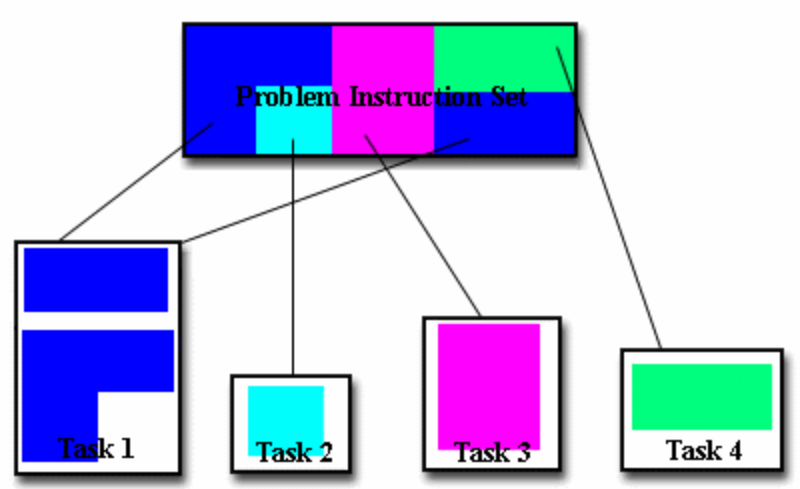


Decomposição e distribuição de dados.

http://www.llnl.gov/computing/tutorials/parallel_com

Functional Decomposition

- Foco nas **computações** a serem realizadas, ao invés de nos **dados** manipulados.
- Computações devem ser divididas em tarefas.
- Idealmente, dados manipulados devem ser também disjuntos.
- Havendo sobreposições nos dados, comunicações podem ser necessárias entre as tarefas, ou dados podem ser replicados.
  - Neste caso, decomposição de dados pode ser mais adequada.



Decomposição funcional

http://www.llnl.gov/computing/tutorials/parallel_com