COMP 4901Q: High Performance Computing (HPC)

Lecture 4: Introduction to Parallel Programming

Instructor: Shaohuai SHI (shaohuais@cse.ust.hk)
Teaching assistants: Mingkai TANG (mtangag@connect.ust.hk)
Yazhou XING (yxingag@connect.ust.hk)

Course website: https://course.cse.ust.hk/comp4901q/

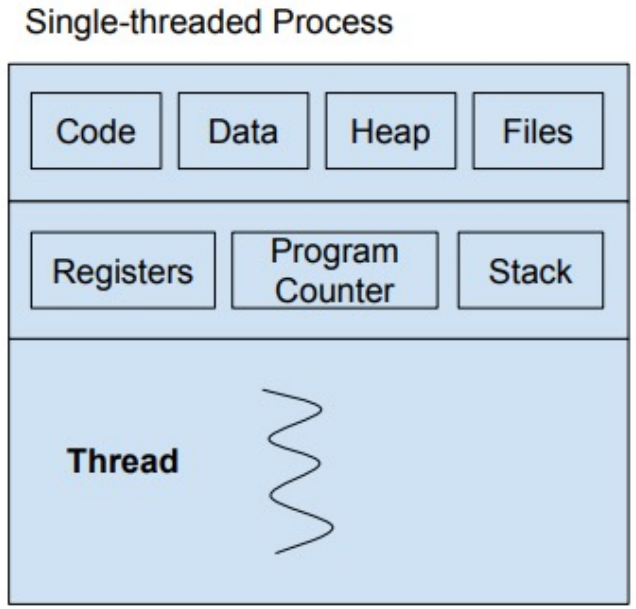
Outline

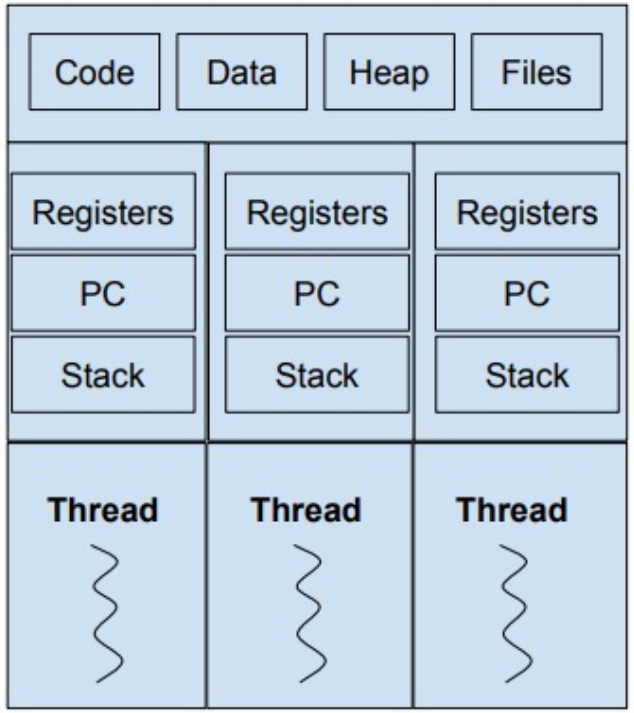
- Parallel Programming Models
  - Shared Memory Model
  - Distributed Memory / Message Passing
  - Data Parallel
  - Hybrid
  - Single Program Multiple Data (SPMD)
  - Multiple Program Multiple Data (MPMD)
- Locality

- Designing Parallel Programs

Program vs. Process vs. Thread

- Program
  - Code stored in computer
    - In disk
    - Or in non-volatile memory
  - Can be executed by the computer
- Process
  - An instance of an executing program is a **process** consisting of an **address space** and **one or more threads** of control
- Thread
  - The unit of execution within a process. A process can have from just one thread to many threads
  - Typically represented by program counter, registers, and stack





Parallel Programming Models atop Parallel Hardware

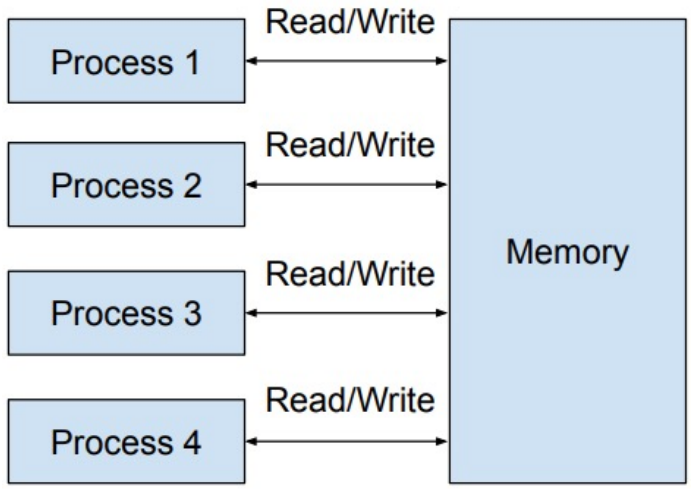
- Parallel programming models are NOT specific to a particular type of machine or memory architecture

- Shared memory model on a distributed memory machine
  - Kendall Square Research (KSR) ALLCACHE approach
  - Also called "virtual shared memory"

- Distributed memory model on a shared memory machine
  - Message Passing Interface (MPI) on SGI Origin 2000

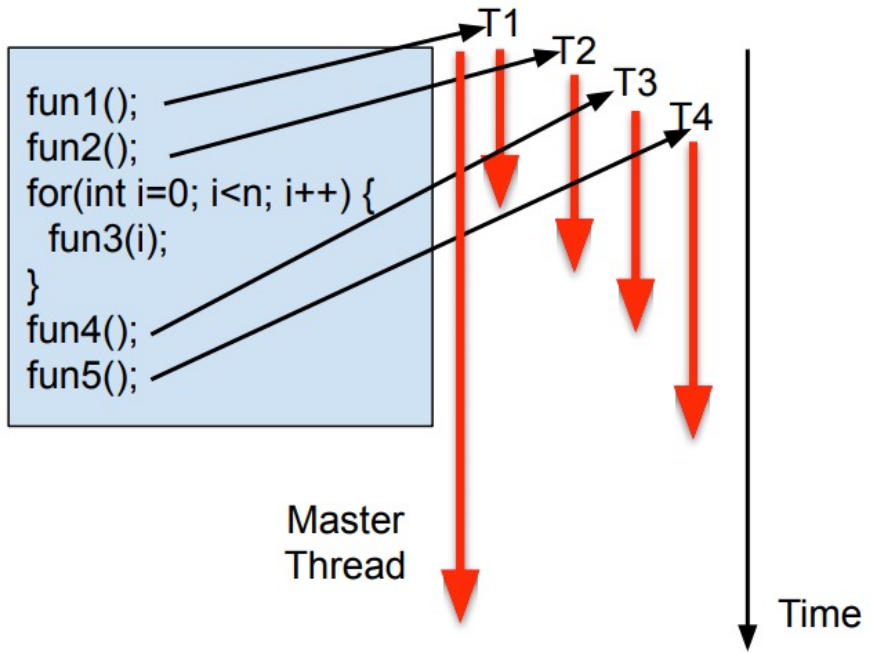
Shared Memory Model (without Threads)

- Processes or tasks share a common address space
  - Read or write asynchronously
  - Locks / semaphores are used to control access
    - resolve contentions
    - prevent race conditions and deadlocks
  - Pros
    - Simple
    - Don's need to specify explicitly communications between tasks
  - Cons
    - Difficult to understand and manage **data locality**
- Implementations
  - On stand-alone shared memory machines
    - the POSIX standard (shm_overview: https://www.man7.org/linux/man-pages/man7/shm_overview.7.html)
    - shared memory segments (e.g., shmget, shmat, shmctl, etc).
  - On distributed memory machines
    - SHMEM (from Cray Research's "shared memory" library)



Shared Memory Model (Threads)

- A single "heavy weight" **process** can have multiple "light weight", concurrent execution paths (**threads**)
  - The master thread in the process creates multiple threads
    - All threads belong to the process
    - All threads share the memory address
  - Multiple threads are scheduled by the native operating system
  - Any thread can execute any subroutine at the same time as other threads
    - T1: fun1()
    - T2: fun2()
    - T3: fun4()
    - T4: fun5()
  - Threads communicate with each other through global memory
  - Threads can be created and destroyed dynamically
- Implementations
  - A library of subroutines: e.g., POSIX threads (pthreads)
    - Very explicit parallelism
    - Requires significant programmer attention to detail
  - Compiler directives: e.g., **OpenMP**
    - Easy and simple to use
    - Portable / multi-platform, including Unix and Windows platforms

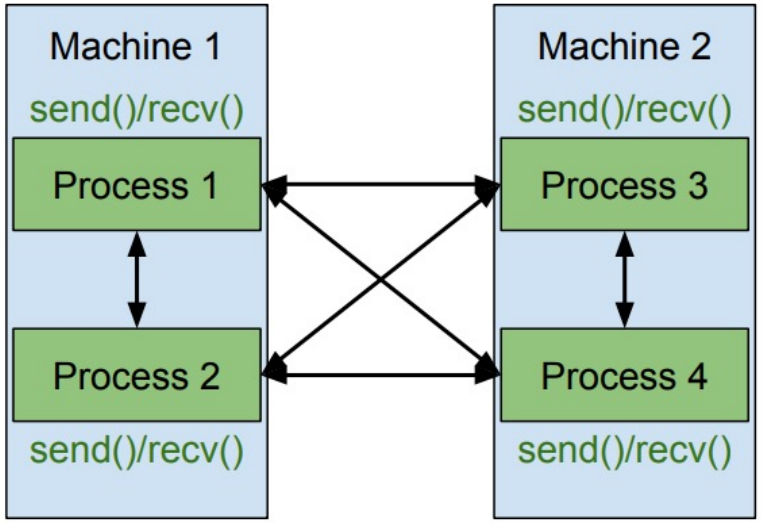


Distributed Memory / Message Passing Model

- A set of tasks (processes) that use their own local memory during computation
  - Multiple tasks can reside on the same physical machine
  - Multiple tasks can also reside across an arbitrary number of machines

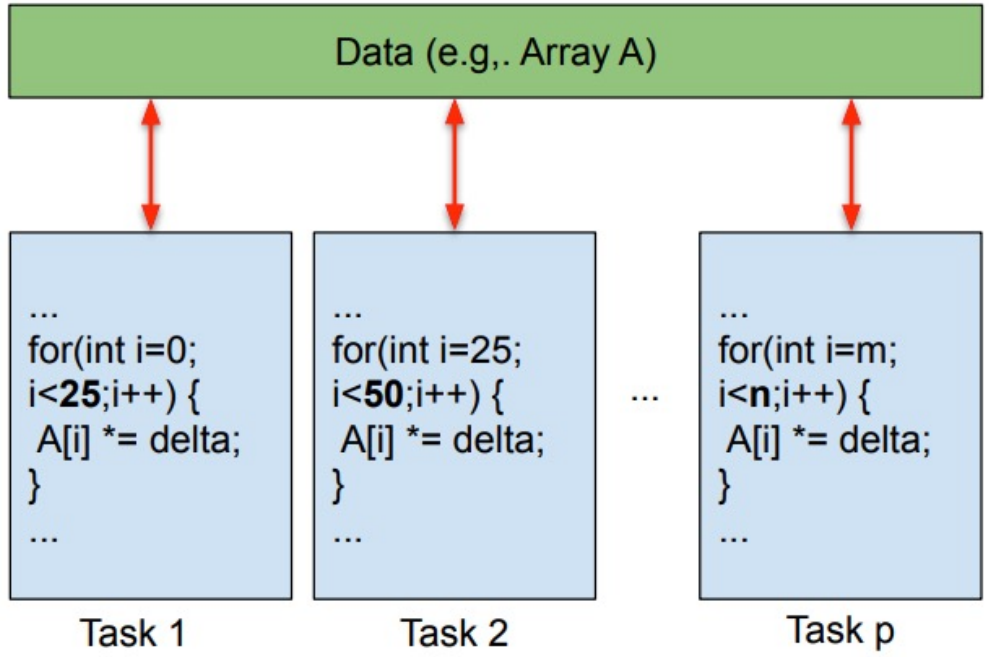
- Tasks exchange data through communications by sending and receiving messages
  - Data transfer usually requires cooperative operations
  - E.g., a send operation must have a matching receive operation

- Implementations
  - TCP/UDP Socket programming
    - Difficult for programmers to develop applications
  - Message Passing Interface (MPI) since 1994
    - The "de facto" industry standard



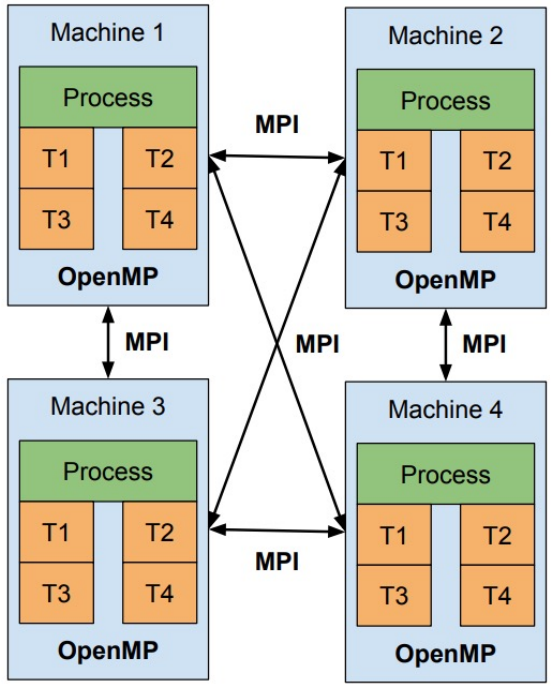
Data Parallel Model

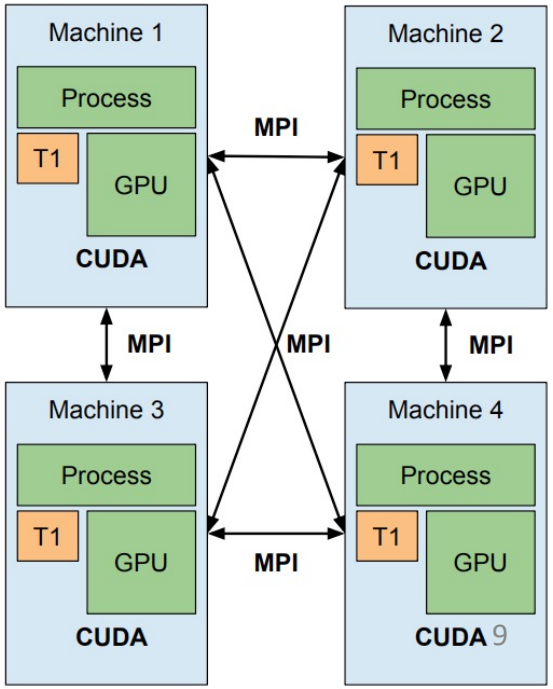
- Also called Partitioned Global Address Space (PGAS)
- Most of the parallel work focuses on performing operations on a data set
  - The data set is typically organized into a common structure, such as an array or cube.
- A set of tasks work collectively on the same data structure
  - each task works on a different partition of the same data structure
  - performs the same operation on their partition of work
- Data partition
  - On shared memory architectures, all tasks may have access to the data structure through global memory
  - On distributed memory architectures, the global data structure can be split up logically and/or physically across tasks.
- Implementations
  - Coarray Fortran: a small set of extensions to Fortran 95
  - Unified Parallel C (UPC): an extension to the C programming language for SPMD parallel programming
  - Global Arrays: provides a shared memory style programming environment
  - X10: a PGAS based parallel programming language
  - Chapel: an open source parallel programming language
  - CUDA on GPUs



Hybrid Model

- A hybrid model combines more than one of the previously described programming models
  - MPI+OpenMP
    - Threads perform computationally intensive kernels using local, on-node data
    - Communications between processes on different nodes occurs over the network using MPI
  - MPI+CUDA
    - Computationally intensive kernels are off-loaded to GPUs on-node
    - Data exchange between node-local memory and GPUs uses CUDA
    - MPI tasks run on CPUs using local memory and communicating with each other over a network
  - MPI+CUDA+OpenMP
    - Some computationally intensive subroutines are computed by multiple CPUs or CPU cores
    - Some computationally intensive kernels are off-loaded to GPUs on-node
    - MPI tasks run on CPUs using local memory and communicating with each other over a network





Single Program Multiple Data (SPMD)

- SPMD
  - All tasks execute their copy of the **same program** simultaneously
  - All tasks may use **different data**

- SPMD is actually a "high level" programming model
  - can be built upon any combination of the previously mentioned parallel programming models

- Different from SIMD
  - SPMD is a subcategory of MIMD
  - SIMD requires the same instructions on different data, while SPMD can run with different instructions (e.g., different subroutines) on different data

Multiple Program Multiple Data (MPMD)

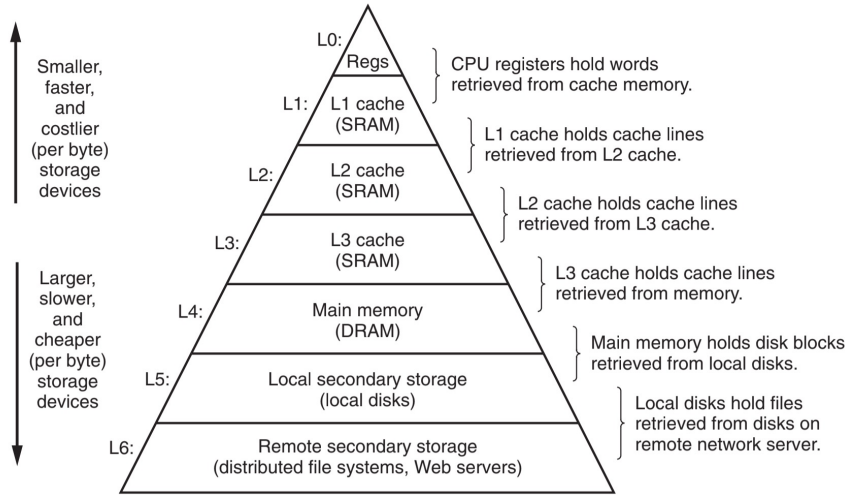
- MPMD
  - Tasks may execute different programs simultaneously
  - All tasks may use different data

- MPMD is also a "high level" programming model
  - can be built upon any combination of the previously mentioned parallel programming models

- Not that common as SPMD
  - but may be better suited for certain types of problems

Locality

- Targets at making processors access data from fast memory
  - Data movement is expensive especially from/to slow memory



- Temporal locality
  - If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in **the near future**
- Spatial locality
  - If a particular storage location is referenced at a particular time, then it is likely that **nearby memory locations** will be referenced in the near future
  - Data elements are brought into cache **one cache line** at a time

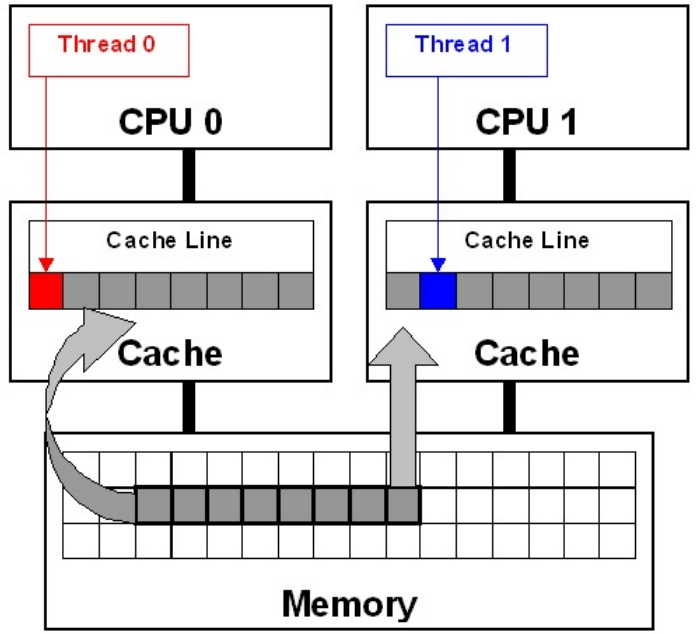
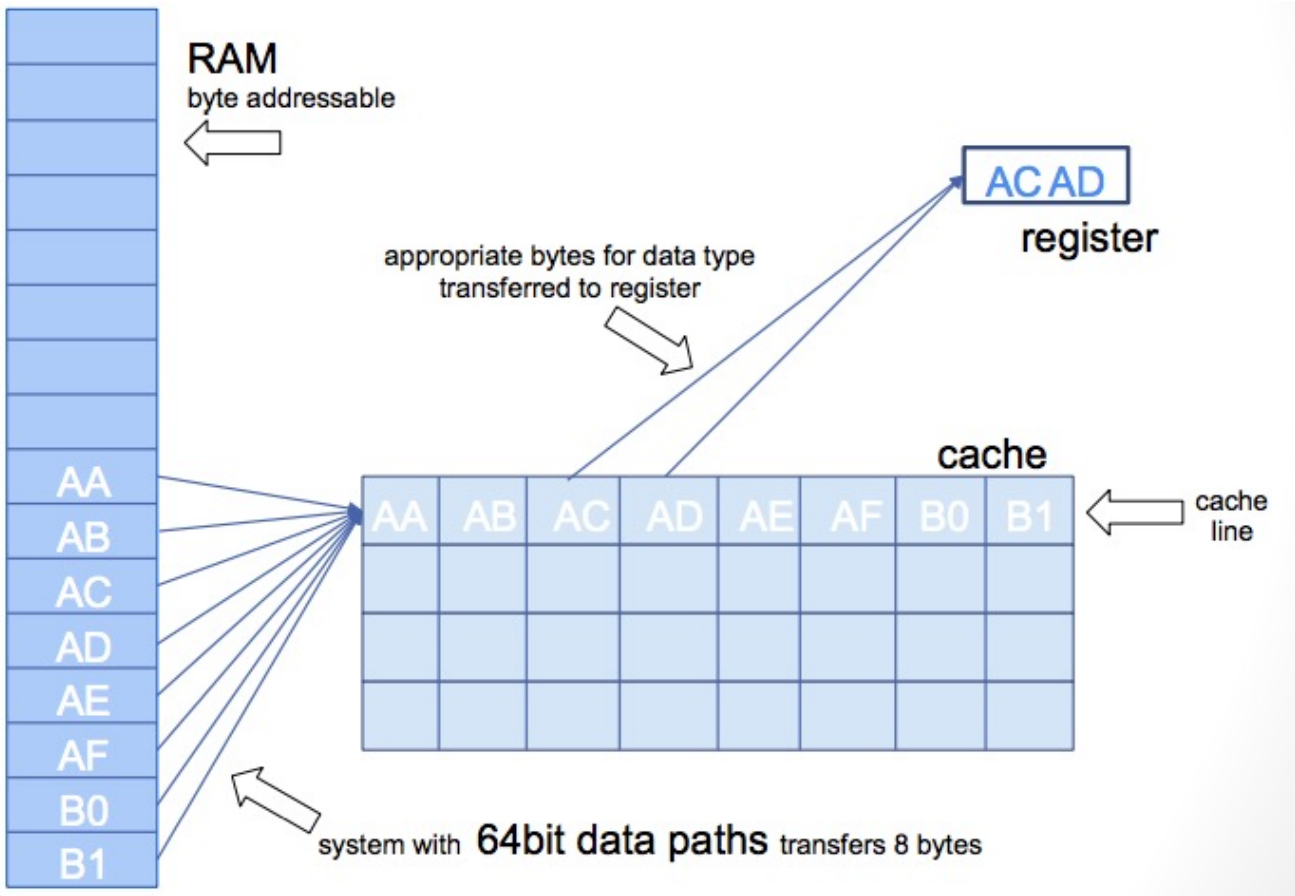


Image credit: http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

Locality - Cache Line

- Direct-Mapped Cache
- Cache performance metrics
  - **Miss rate**
    - Fraction of memory references not found in cache
  - Hit time
    - Time to deliver a line in the cache to the processor
    - Typical numbers
      - 1 clock cycle for L1
      - 3-8 clock cycles for L2
  - Miss penalty
    - Additional time required because of a miss
    - Typically 25-100 cycles for main memory



- Overall cache performance
  - AMAT (Avg. Mem. Access Time) = t_hit + prob_miss * penalty_miss

Image credit: http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

Layout of C/C++ Arrays in Memory

- C/C++ arrays allocated in row-major order
  - Each row in contiguous memory locations
  - Stepping through columns in one row
    - for(int i=0; i<N; i++) sum += A[0][i];
    - Accesses successive elements
    - If block size (B) > 4 bytes, exploit spatial locality
  - Stepping through rows in one column
    - for(int i=0; i<N; i++) sum += A[i][0];
    - accesses distant elements
    - no spatial locality!

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

a	b	c	d	e	f		o	p

Row-major order

Locality Example 1

- Example: Summing all elements of a 2D array
  - Assume: 4-byte words, 4-word cache line

```
int sum_array_rows(int A[N][N]) {
  int i, j, sum = 0;
  for(i=0; i<N; i++) {
    for(j=0; j<N; j++) {
      sum += A[i][j];
    }
  }
}
```

```
int sum_array_cols(int A[N][N]) {
  int i, j, sum = 0;
  for(j=0; j<N; j++) {
    for(i=0; i<N; i++) {
      sum += A[i][j];
    }
  }
}
```

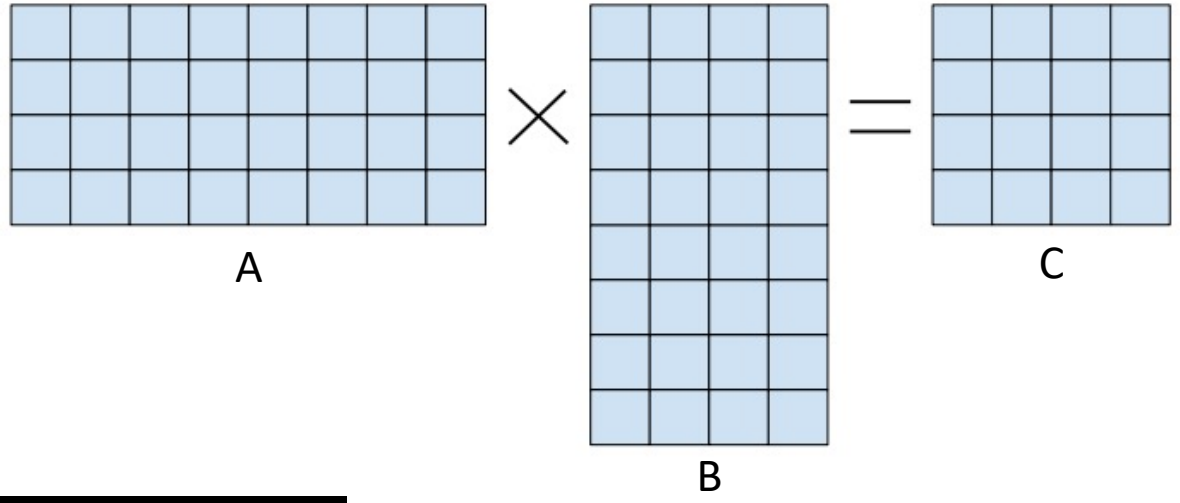
a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

a	b	c	d	e	f		o	p

Row-major order

Locality Example 2

```
const int m=4096;
const int n=4096;
const int k=4096;
float C[m][n];
float B[k][n];
float A[m][k];
```




```
for(int i=0; i<m; i++) {
  for(int j=0; j<n; j++) {
    for(int p=0; p<k; p++) {
      C[i][j] += A[i][p]*B[p][j];
    }
  }
}
```

```
for(int i=0; i<m; i++) {
  for(int p=0; p<k; p++) {
    for(int j=0; j<n; j++) {
      C[i][j] += A[i][p]*B[p][j];
    }
  }
}
```

Designing Parallel Programs

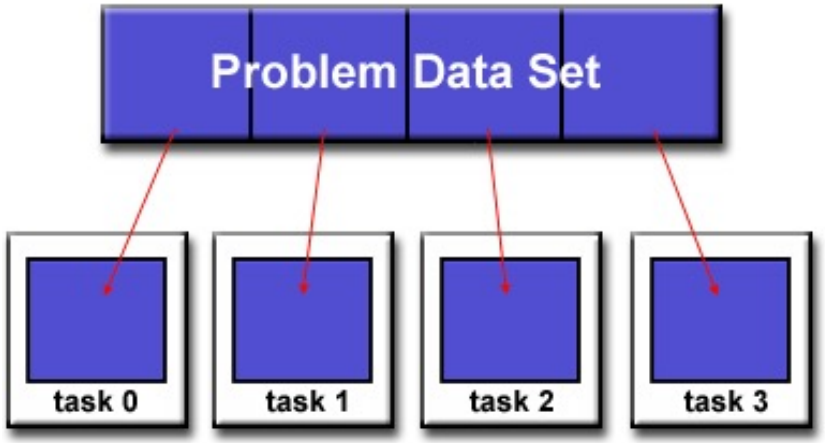
- Automatic vs. Manual Parallelization
  - Designing and developing parallel programs has characteristically been a very manual process
  - The programmer is typically responsible for both identifying and actually implementing parallelism
- Manually developing parallel programs
  - Time consuming, complex, error-prone
- Automatic with a parallelizing compiler in two ways
  - Fully automatic
    - The compiler analyzes the source code and identifies opportunities for parallelism
    - Loops (do, for) are the most frequent target for automatic parallelization
  - Programmer directed
    - Using "compiler directives" or possible compiler flags
- Manual design and automatic tools are what we need!
  - Manual design: Understand the problem and the program
  - Automatic tools: make use of parallel programming frameworks (e.g., OpenMP, CUDA, and MPI)

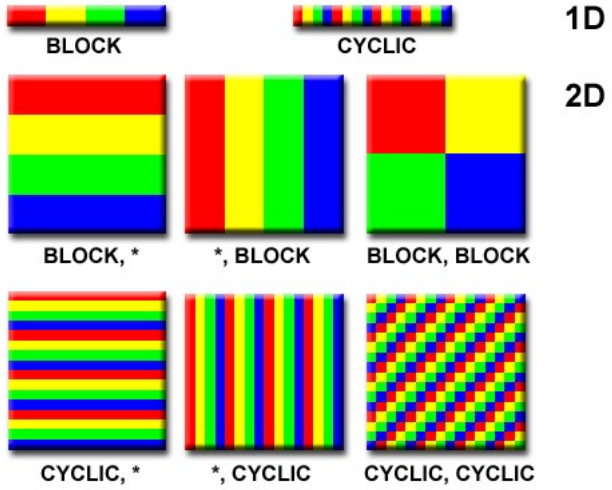
Understand the Problem and the Program

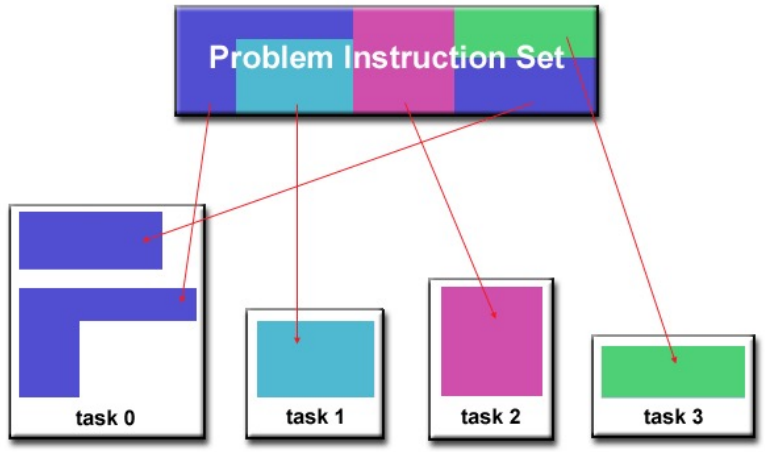
- The first step in developing parallel software is to first understand the problem that you wish to solve in parallel
  - Determine whether or not the problem is one that can actually be parallelized
    - Understand the existing serial code
  - Identify the program's hotspots
  - Identify bottlenecks in the program
  - Identify inhibitors to parallelism
  - Investigate other algorithms if possible
  - Take advantage of optimized third party parallel software and highly optimized math libraries available from leading vendors

Partitioning

- Break the problem into discrete "chunks" of work that can be distributed to multiple tasks
- Domain decomposition and functional decomposition
  - Domain decomposition: the data associated with a problem is decomposed
  - Functional decomposition: the problem is decomposed according to the work that must be done







Understand the Problems - Examples

- Example 1: Calculate the potential energy for each of several thousand **independent conformations** of a molecule. When done, find the minimum energy conformation
  - Each of the **molecular conformations** is independently determinable
  - The calculation of the **minimum energy conformation** is also a parallelizable problem

- Example 2: Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula: F(n) = F(n-1) + F(n-2), F(0)=0, and F(1)=1
  - The calculation of the F(n) value uses those of both F(n-1) and F(n-2), which must be computed first
  - It is difficult to parallel!
  - How if we use the closed form of F(n)?

Designing Parallel Programs - Communications

- Who Needs Communications?
  - The need for communications between tasks depends upon your problem
- No-communication problems
  - Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data
  - These types of problems are often called **embarrassingly parallel** - little or no communications are required
- Communication problems
  - Most parallel applications are not quite so simple, and do require tasks to share data with each other
- Factors to Consider
  - Communication overhead
    - Latency vs. bandwidth
  - Visibility of communications
  - Synchronous vs. asynchronous

Summary

- Different types of parallel programming models
  - Shared Memory Model
  - Distributed Memory / Message Passing
  - Data Parallel
  - Hybrid
  - Single Program Multiple Data (SPMD)
  - Multiple Program Multiple Data (MPMD)
- Locality
  - Memory hierarchy
  - Cache line
- How to design parallel programs

Reading List

- Pacheco, Peter. An introduction to parallel programming. Elsevier, 2011. **Chapter 2.4-2.8.** [PDF: http://www.e-tahtam.com/~turgaybilgin/2013-2014-guz/ParalelProgramlama/ParallelProg.pdf]