Lecture 27
Programming parallel hardware

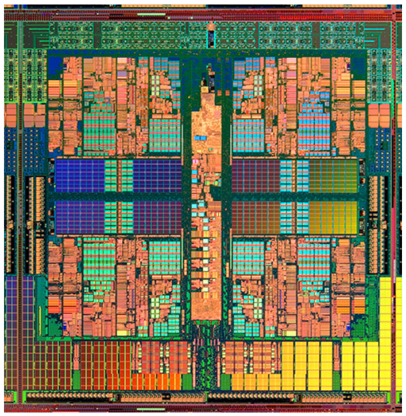
Suggested reading:

(see next slide)

Suggested Readings

- Readings
  - H&P: Chapter 7 – especially 7.1-7.8
  - Introduction to Parallel Computing
    - https://computing.llnl.gov/tutorials/parallel_comp/
  - POSIX Threads Programming
    - https://computing.llnl.gov/tutorials/pthreads/

Multicore processors and programming

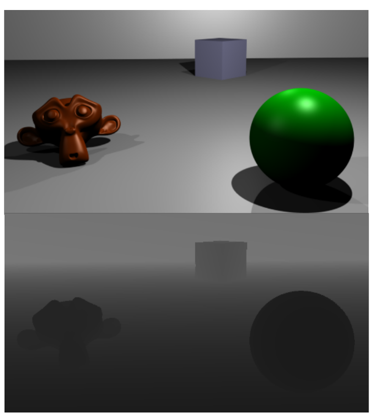


Processor components

Processor comparison

AMD Athlon 64 vs. intel Pentium Dual-Core inside

- Explain & articulate why modern microprocessors now have more than one core and how SW must adapt.
- Use knowledge about underlying HW to write more efficient software



Writing more efficient code

The right HW for the right application

```
for i=0; i<5; i++ {
      a = (a*b) + c;
}
```

```
MULT r1,r2,r3    # r1 ← r2*r3
ADD r2,r1,r4     # r2 ← r1+r4
```

110011	000001	000010	000011
001110	000010	000001	000100

HLL code translation

Fundamental lesson(s)

- Some problems map well to parallel systems, others do not (and demand a fast, single thread).
- In this lecture, we will consider what classes of problems fall into each category

Why it’s important…

- If you are writing software for a multi-core processor, and don't understand the implications / specifics of the underlying hardware, it's possible to write some very bad, ill-performing code.

Writing (good) parallel code:

1. To develop parallel software, must first understand if serial code can be parallelized…

(Understand the problem)

Example: independent array elements

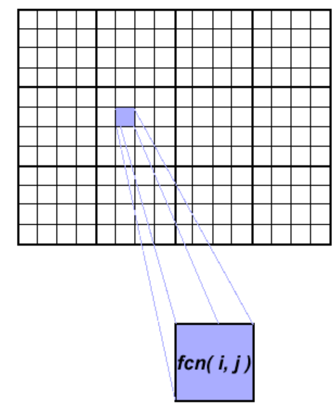
Example:

1. **Calculations on 2D array elements**

– **Computation on each array element independent from others**

- **Serial code:**

```
for j =1:N
  for i=1:N
    a(i,j) = f(i,j)
```

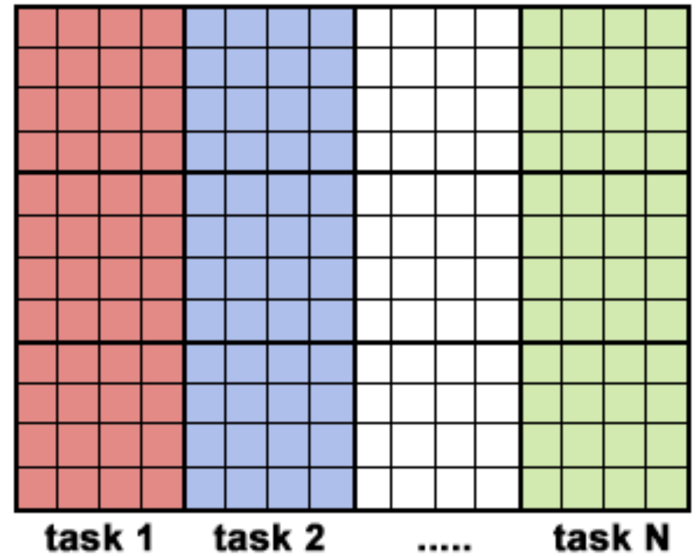


- **If calculation of elements is independent from one another, problem is “embarrassingly parallel”**
  – (usually computationally intensive)

(Understand the problem)

Example: independent array elements

– Can distribute elements so each processor owns its own array (or subarray)
  - This type of problem can lead to "superlinear" speedup
    – (Entire dataset may now fit in cache)



– Parallel code might look like…

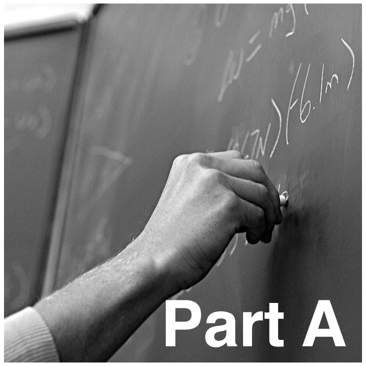
```
find out if I am MASTER or WORKER

if I am MASTER

  initialize the array
  send each WORKER info on part of array it owns
  send each WORKER its portion of initial array

  receive from each WORKER results

else if I am WORKER
  receive from MASTER info on part of array I own
  receive from MASTER my portion of initial array

  # calculate my portion of array
  do j = my first column,my last column
  do i = 1,n
    a(i,j) = fcn(i,j)
  end do
  end do

  send MASTER results

endif
```

(Understand the problem)

Example: loop carried dependence

- (B) Calculate the numbers in a Fibonacci sequecne
  - Fibonacci number defined by:
    - F(n) = F(n-1) + F(n-2)
      - Fibonacci series (1,1,2,3,5,8,13,21,…)
  - This problem is non-parallelizable!
    - Calculation of F(n) dependent on other calculations
    - F(n-1) and F(n-2) cannot be calculated independently

(Understand the problem)



Other concerns…

- Identify program "hotspots"
  - Most work – i.e. in scientific or technical code – done in just a few places
- Identify program bottlenecks
  - Are there areas that are disproportionately slow?
    - (I/O usually slows program down)
  - Solution?
    - Restructure program to tolerate latencies
- Identify other inhibitors
  - Again, data dependence is example

(Understand the problem)

Writing (good) parallel code

2. Break up program into chunks of work that can be distributed to multiple processing nodes
   - 2 types:
     - Domain decomposition
     - Functional decomposition

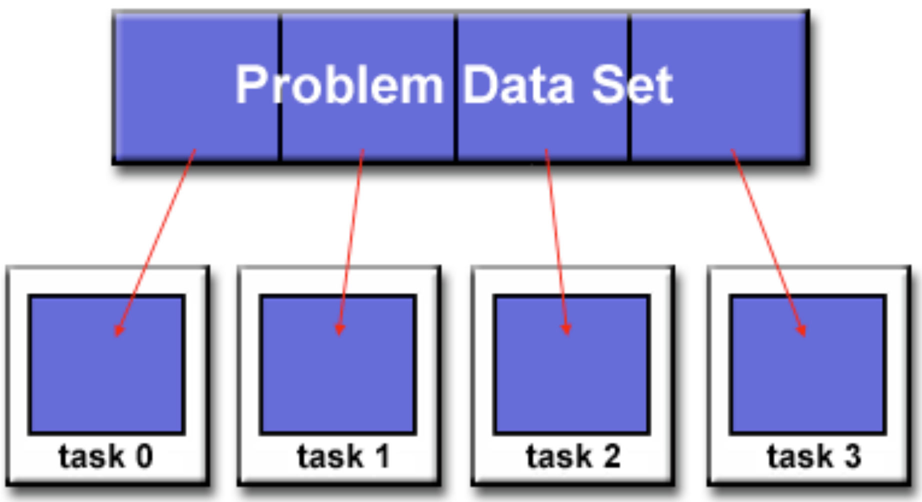
} Any idea what these mean?

(Partitioning)

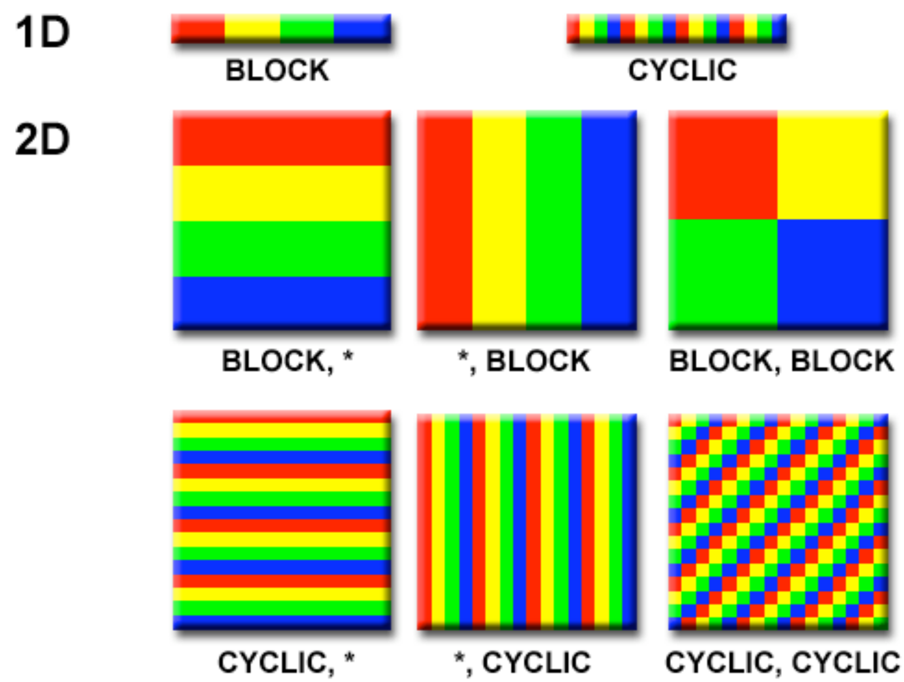
Domain decomposition

Data associated with a problem is decomposed.

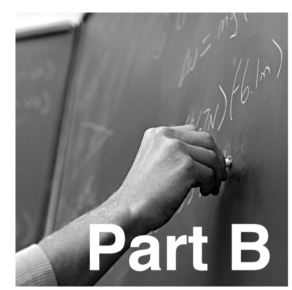
- Each parallel task then works on a portion of the data.



- Different ways to partition data…

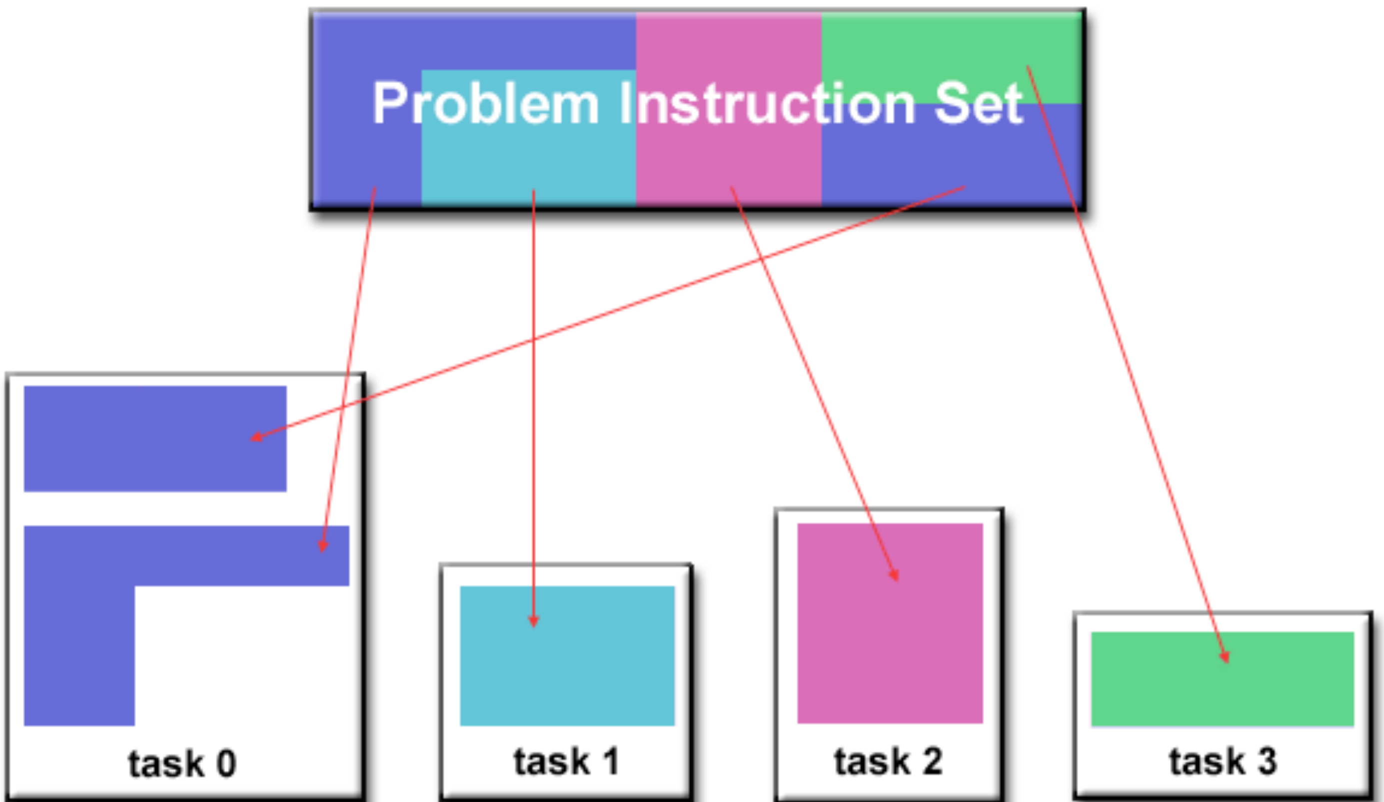


(Partitioning)



Functional Decomposition

- **Focus is on computation performed, not data manipulated**
  - Problem decomposed according to work that is done
  - Each task performs a portion of overall work

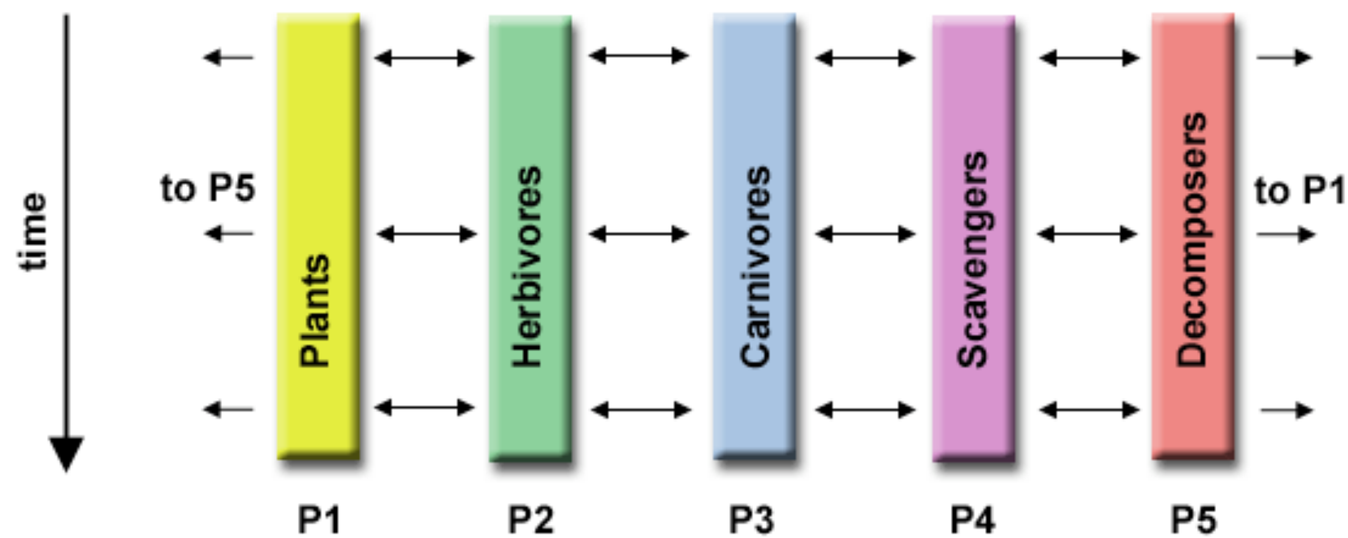


(Partitioning)

Functional Decomposition

- **Lends itself well to problems that can be split into different tasks**
  - **Example: Ecosystem modeling…**
    - **Each program calculates population of a given group**
    - **Each group's growth depends on that of neighbor**
    - **As time progresses, each process calculates current state**
      - **Can then exchange information with neighbors…**
      - **…and begin again…**

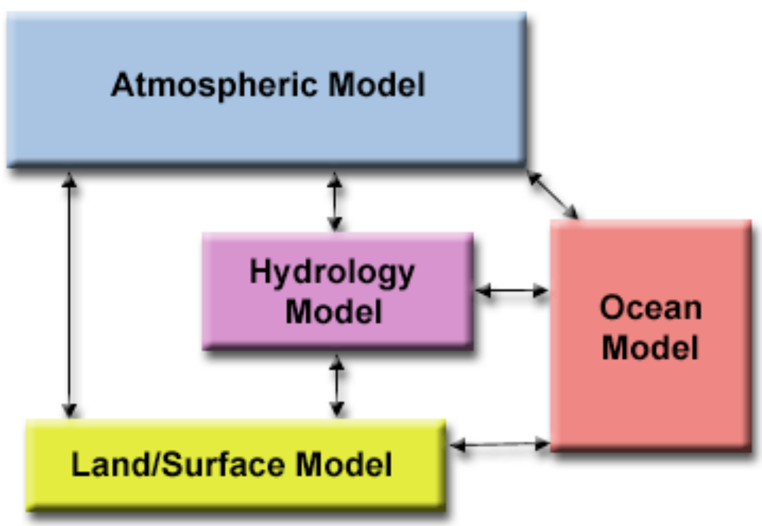
Question:
Is exchange part of parallel operation?



(Partitioning)

Functional Decomposition

- Example: Climate modeling
  - Each model thought of as separate task
  - Arrows represent exchanges of data…
    - Atmosphere model generates wind velocity data → wind velocity data used by ocean model → ocean model generates sea surface temperature data → sea surface temperature data used by atmosphere model



Questions:

1. Do coarse grain dependencies exist too?
2. Are there potential load balancing issues to contend with?

- Within each model, may have embarrassingly parallel functions, data dependencies, etc.

(Partitioning)

Writing (good) parallel code:

3. We must account for the time required for different processing nodes to *communicate*.
   - (As seen from last lecture, this can increase computation time from N to N+M)

(Communication)

Communication

- **Some problems (programs) don't incur excessive communication overhead**
  - **Image processing good example**
    - **i.e. take every pixel and change its color**
      - **No communication overhead required**

- **Most parallel programs / problems do involve tasks that must share data with one another**
  - **Could be practical (distributed memory)**
  - **Could be algorithmic**
    - **Changes to neighboring data has a direct effect on task's data**
      - **(e.g. heat diffusion problem to be discussed, ecosystem modeling, etc.)**

(Communication)

Communication costs

- Inter-task communication implies overhead
- Machine cycles / resources that could be used for computation are instead…
  - …spent packaging and transmitting data
    - (From N cycles to N+M)
- Communication usually means that tasks must be synchronized…
  - …so 1 task may wait for another to finish its work
    - (ecosystem modeling problem)
- Like a highway in a major city, only so much bandwidth for cars that want to use it…
  - See Lecture 26 case studies – can flood network

Communication

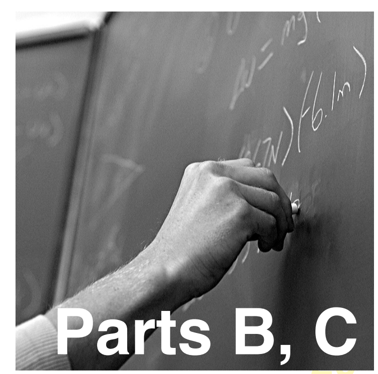
- Knowing which tasks must communicate with each other is critical when writing parallel code
  - Similarly, knowledge about communication vehicle equally important
    - Example:
      - What if each of N nodes needs to send M bit message every Q clock cycles?
      - However, interconnection network can only support N, (M / 4) bit messages every Q cycles…
    - May have written correct code, but performance will suffer b/c hardware cannot support implicit communication demands

- May even want to manually map problem parts to cores
  - Idea: think about which node will talk to which…

Communication examples:

- Heat transfer problem
- Loop carried dependence

(Communication)



Writing (good) parallel code:

4. When a task performs a communication operation, some form of coordination (or *synchronization)* is required with the other task(s) participating in the communication
   - Example:
     - Before task can perform send, must first receive an acknowledgment from the receiving task that it is OK to send
       - (May not always be the case … but this is NOT useful "computation")

(Synchronization)

Types of synchronization

- Barrier
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
  - When the last task reaches the barrier, all tasks are synchronized.
    - What happens from here varies.
      - Often, a serial section of work must be done.
      - In other cases, the tasks are automatically released to continue their work.

Types of synchronization

- **Semaphore**
  - Can involve any number of tasks
  - Typically used to serialize (protect) access to global data or a section of code.
  - Only one task at a time may use (own) the lock / semaphore / flag.
    - The first task to acquire the lock "sets" it.
    - This task can then safely (serially) access the protected data or code.
    - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.

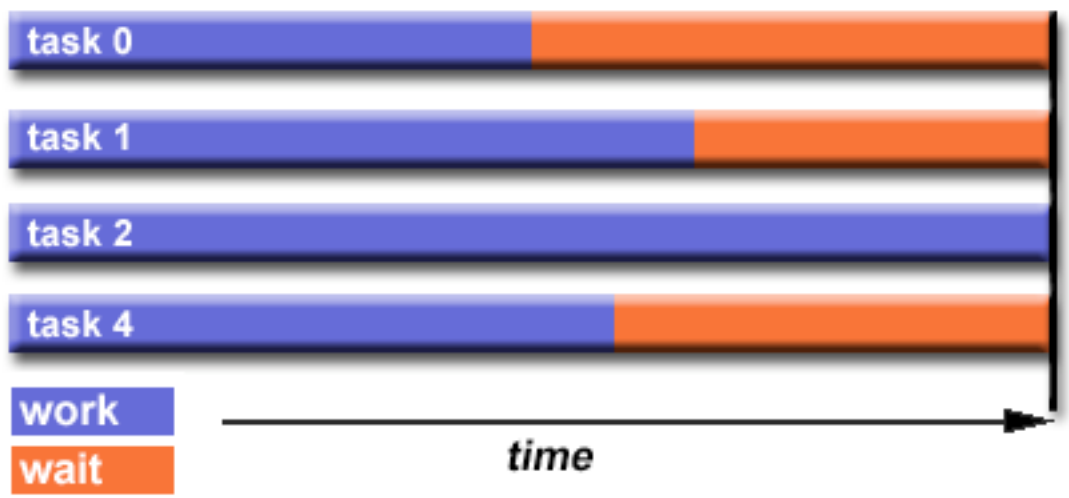
Questions:

1. In context of CSM, DSM, why is synchronization needed?
2. Does synchronization demand architectural support?

Writing (good) parallel code:

5. **Load balancing: keep all cores busy at all times**
   - (i.e. minimize idle time)

- **Example:**
  - If all tasks subject to barrier synchronization, slowest task determines overall performance:



(Load balancing)