# Lecture 26
# Parallel Programming Meets Architecture

Adapted in part from: https://computing.llnl.gov/tutorials/parallel_comp



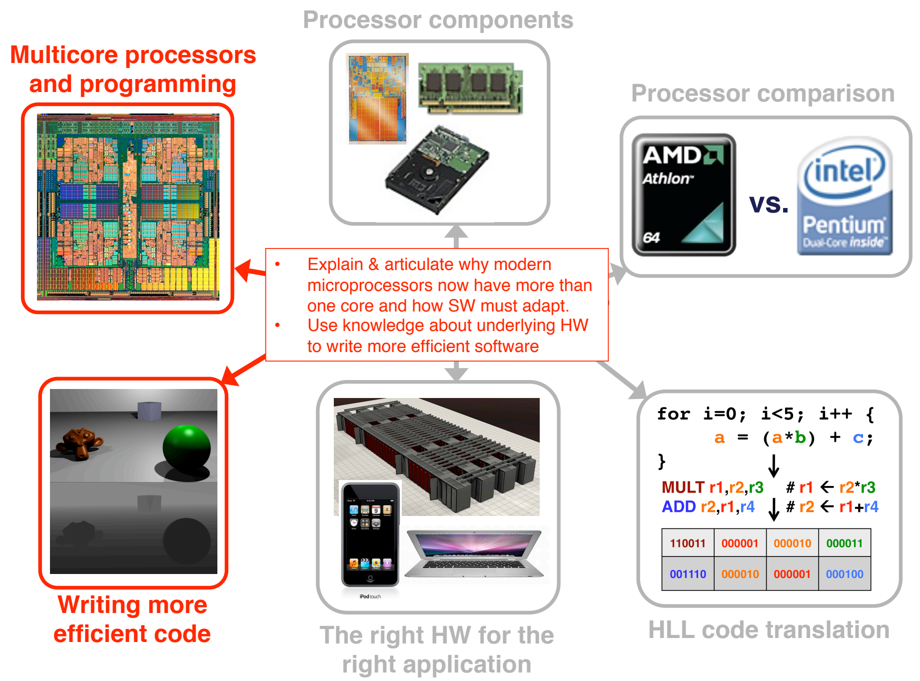


## Suggested Readings

- Readings
  - H&P: Chapter 7 – especially 7.1-7.8
    - (Over next 2 weeks)
  - Introduction to Parallel Computing
    - https://computing.llnl.gov/tutorials/parallel_comp/
  - POSIX Threads Programming
    - https://computing.llnl.gov/tutorials/pthreads/
  - How GPUs Work
    - *www.cs.virginia.edu/~gfx/papers/pdfs/59_HowThingsWork.pdf*









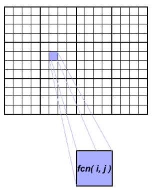



## Understand the Problem and Program

1. To develop parallel software, must first understand if serial code can be parallelized…
   - Consider 2 examples…

- (A) Calculations on 2D array elements
  - Computation on each array element independent from others
    - Serial code:

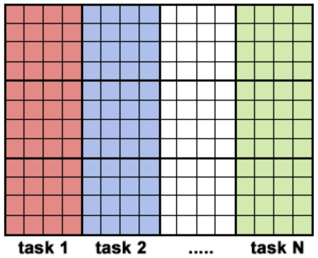
```
for j =1:N
   for i=1:N
      a(i,j) = f(i,j)
```




    - If calculation of elements is independent from one another, problem is "embarrassingly parallel"
      - (usually computationally intensive)

# Understand the Problem and Program

- 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
```





# (1) Understand the Problem and Program

- (B) Calculate the numbers in a Fibonacci sequecne
  - (Hint: Part of a project benchmark…)
  - 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*





# (1) Understand the Problem and Program

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





# Example: Binary Search

- Is this problem parallelizable?

# (2) Partitioning

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

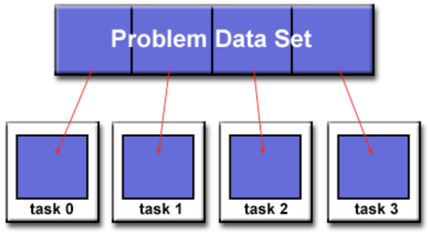




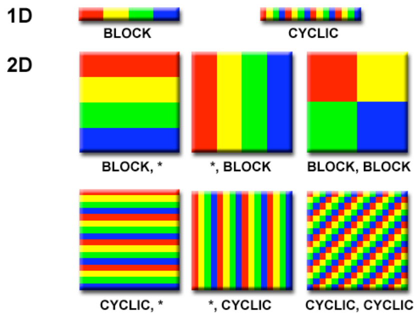
# (2) Domain Decomposition

***Data* associated with a problem is decomposed.**

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




- Different ways to partition data…

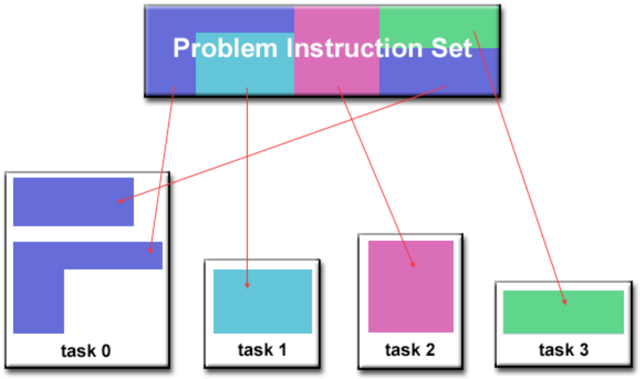







# (2) Functional Decomposition

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

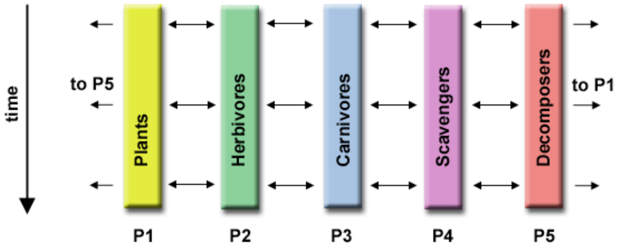







# Functional Decomposition

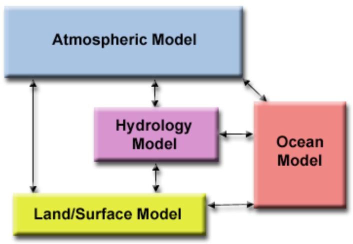
- Lends itself well to problems that can be split into different tasks
  - Example: Ecosystem modeling…
    - Each program calculates population of a given group
    - Each group's growth depends on that of neighbor
    - As time progresses, each process calculates current state
      - Can then exchange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




**Question:**
Is exchange part of parallel operation?

# (2) 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.





# (3) Communication

- FYI…
  - This topic gets its whole lecture
    - (Focus will be the HW/architecture perspective)
  - This topic gets its own homework problem
    - (Focus will be the HW/architecture perspective)
  - Here, we talk a bit about communication in the context of the program itself…





# (3) Communication

- Some problems (programs) don't incur excessive communication overhead
  - Aforementioned 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 (e.g. heat diffusion problem)
    - Changes to neighboring data has a direct effect on task's data





# (3) Communication (costs)

- Inter-task communication implies overhead
- Machine cycles / resources that could be used for computation are instead…
  - …spent packaging and transmitting data
    - (NOT parallelizable)
- Communication usually means that tasks must be synchronized…
  - …so 1 task may wait for another to finish its work
    - (NOT parallelizable)
- Like a highway in a major city, only so much bandwidth for cars that want to use it…
  - …competing communication traffic can further exacerbate performance

# (3) Communication

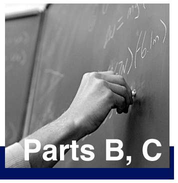
- Knowing which tasks must communicate with each other is critical when writing parallel code
  - Similarly, knowledge about communication vehicle equally important
    - Example:
      - What if each of N nodes needs to send M bit message every Q clock cycles?
      - However, interconnection network 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





# Examples:

- Heat transfer problem
- Loop carried dependence








# (4) Synchronization

- When a task performs a communication operation, some form of coordination 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 parallelizable!)





# (4) 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.

# (4) 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?





# (5) Load balancing

- (Saw example in Lecture 24)
- Idea:
  - Want to keep all tasks busy at all times
    - (i.e. minimize idle time)
- Example:
  - If all tasks subject to barrier synchronization, slowest task determines overall performance: