Synchronization and Load Balancing

Why?

- Synchronization
  - Data Dependency
  - Task Dependency
- Load Balancing
  - Reduce communication
  - Improve performance

Data Dependencies

- **Definition:**
  - A ***dependence*** exists between program statements when the order of statement execution affects the results of the program.
  - A ***data dependence*** results from multiple use of the same location(s) in storage by different tasks.
  - Dependencies are important to parallel programming because they are one of the *primary inhibitors* to parallelism.

Loop Carried Dependency

```
DO 500 J = MYSTART,MYEND
A(J) = A(J-1) * 2.0
500 CONTINUE
```

- The value of A(J-1) must be computed before the value of A(J), therefore A(J) exhibits a data dependency on A(J-1). Parallelism is inhibited.
- If Task 2 has A(J) and task 1 has A(J-1), computing the correct value of A(J) necessitates:
  - ***Distributed memory architecture*** - task 2 must obtain the value of A(J-1) from task 1 after task 1 finishes its computation
  - ***Shared memory architecture*** - task 2 must read A(J-1) after task 1 updates it

Loop Independent Data Dependence

```
task 1        task 2
------        ------
X = 2         X = 4
  .             .
  .             .
Y = X**2      Y = X**3
```

- As with the previous example, parallelism is inhibited. The value of Y is dependent on:
  - ***Distributed memory architecture*** - if or when the value of X is communicated between the tasks.
  - ***Shared memory architecture*** - which task last stores the value of X.
- Although all data dependencies are important to identify when designing parallel programs,
- loop carried dependencies are particularly important since loops are possibly the most common target of parallelization efforts.

How to Handle Data Dependencies

- Distributed memory architectures
  - communicate required data at synchronization points.
- Shared memory architectures
  - synchronize read/write operations between tasks.

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.

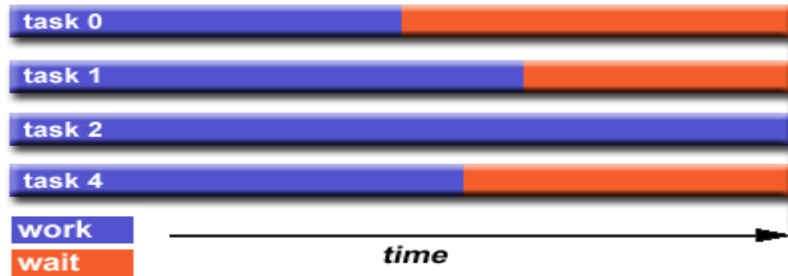
Types of Synchronization: Lock

- 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.
- Other tasks must wait until the task that owns the lock releases it.
- Can be blocking or non-blocking

Synchronous communication

- Involves only those tasks executing a communication operation
- When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication.
- For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.

Load Balancing



- Load balancing refers to the practice of distributing work among tasks so that ***all*** tasks are kept busy ***all*** of the time.
- It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons.
  - For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.

How to Achieve Load Balance

- Equally partition the work each task receives (statically)
- Use dynamic work assignment

Load balance: Equal Partition

- For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
- For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.

Load balance: Equal Partition …

- If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool to detect any load imbalances.
- Adjust work accordingly.

Load balance: Dynamic Work Partitioning

- Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:
  - ***Sparse arrays*** - some tasks will have actual data to work on while others have mostly "zeros".
  - ***Adaptive grid methods*** - some tasks may need to refine their mesh while others don't.
  - ***N-body simulations*** - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks.

Load balance: Dynamic Work Partitioning

- When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a ***scheduler - task pool*** approach. As each task finishes its work, it queues to get a new piece of work.
- It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

QUESTIONS??