6-7 Parallel Software

Lecture 6 Parallel Software: Basic

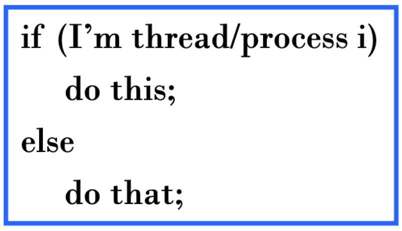
The burden is on software

From now on...

- In shared memory systems:
    - Start a single process and fork threads.
    - Threads carry out tasks.
- In distributed memory systems:
    - Start multiple processes.
    - Processes carry out tasks.
- When using accelerators (e.g. GPUs)
    - Start a process with one or more threads.
    - The thread launches task to be done on the GPU.
    - GPU will do the same task on different data.

SPMD – single program multiple data

SPMD programs consist of a single executable (i.e. single program) forked into different processes/threads, that can behave as if it were multiple different programs through the use of conditional branches.



```
if (I'm thread/process i)
    do this;
else
    do that;
```

Writing Parallel Programs

1. Divide the work among the processes/threads so that each process/thread gets roughly the same amount of work
2. Arrange for the processes/threads to synchronize if needed.
3. Arrange for communication among processes/threads. Reduce communication as much as possible.

Shared Memory Systems (Threads)

Dynamic threads: Master thread waits for work, forks new threads, and when threads are done, they terminate.

+ Efficient use of resources

- Thread creation and termination is time consuming.

Static threads: Pool of threads created and are allocated work, but do not terminate until cleanup.

+ Better performance

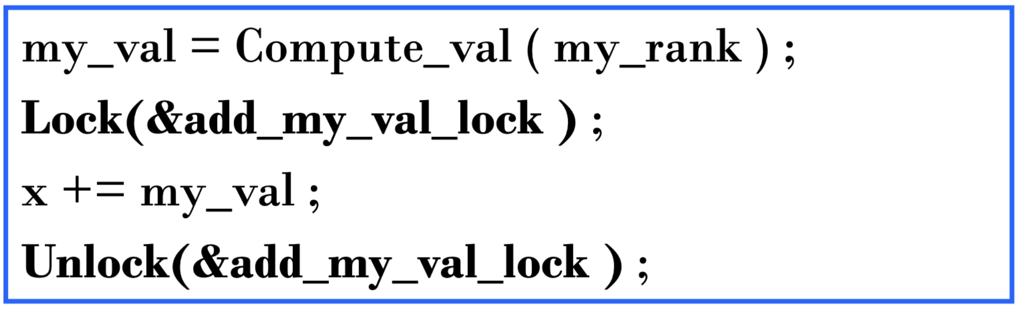
- potential waste of system resources

Effect of Nondeterminism

Race condition

Critical section

- Need to enforce mutual exclusion through locks (mutex)



```
my_val = Compute_val ( my_rank ) ;
Lock(&add_my_val_lock ) ;
x += my_val ;
Unlock(&add_my_val_lock ) ;
```

Isn't cache coherence enough to ensure determinism? Why need critical section?

- Critical section can have more than one line, cache coherence can't
- Cache coherence ensures that values are the same for a variable in different caches
- We need both: hardware gives coherence, software gives locks

Cache coherence is not enough to deal with critical sections. This is why we need programming tools like locks, mutex, etc. State two reasons for that.

- The critical section may consist of more than one instruction. Coherence does not deal with that.
- Coherence ensures that no two cores update their similar cache blocks at the same time. But it does not two threads from updating the same data item. It just serializes them, which will still lead to wrong result.

Distributed Memory Systems (Processes)

Distributed Memory: Message-Passing

Foster's methodology (The PCAM Methodology)

1. Partitioning: divide the computation to be performed and the data operated on by the computation into small tasks.

   The focus here should be on identifying tasks that can be executed in parallel.

   Ideally, an increase in problem size should increase the number of tasks rather than the size of individual tasks.

2. Communication: determine what communication needs to be carried out among the tasks identified in the previous step.

3. Aggregation: combine tasks and communications identified in the first step into larger tasks.

   If task A must be executed before task B, aggregate them into a single composite task.

4. Mapping: assign the composite tasks identified in the previous step to processes/threads.

In steps 1 & 2, you design your program using machine-independent issues: concurrency, scalability, ...

In steps 3 & 4, you tweak your program to make the best use of the underlying hardware.

Lecture 7 Parallel Software: Advanced

Concurrency vs Parallelism

Concurrency: At least two tasks are making progress at the same time frame.

- Not necessarily at the same time
- Include techniques like time-slicing
- Can be implemented on a single processing unit
- Concept more general than parallelism

Parallelism: At least two tasks execute literally at the same time.

- Requires hardware with multiple processing units

Amdahl's Law

How much of a speedup one could get for a given parallelized task?

If F is the fraction of a calculation that is sequential then the maximum speed-up that can be achieved by using P processors is $\frac{1}{(F+(1-F)/P)}$.

Other significant factors: sequential-to-parallel synchronization, inter-core communication

In applications with high degree of data sharing and high inter-core communication requirements: needs a smaller number of larger cores

DAG Model for Multithreading

A vertex is a unit of execution.

For example: an instruction (we assume this later), a basic block, a function.

An edge indicates dependency.

For example, an edge from vertex A to vertex B means A must execute first before B.

Work: total amount of time spent on all instructions

T_p = The fastest possible execution time on P processors

Work Law: $T_p \geq T_1 / P$

Span: The longest path of dependence in the DAG $= T_\infty$

Span Law: $T_p \geq T_\infty$

Parallelism can be defined as: T_1 / T_∞

The whole challenge of parallel programming is to make the best use of the underlying hardware to exploit the different type of parallelisms.

Sources of Performance Loss in Parallel Programs

Extra overhead

- synchronization
- communication

Artificial dependencies

- Hard to find
- May introduce more bugs
- A lot of effort to get rid of

Contention due to hardware resources

Coherence

- Extra bandwidth (scarce resource)
- Latency due to the protocol
- False sharing

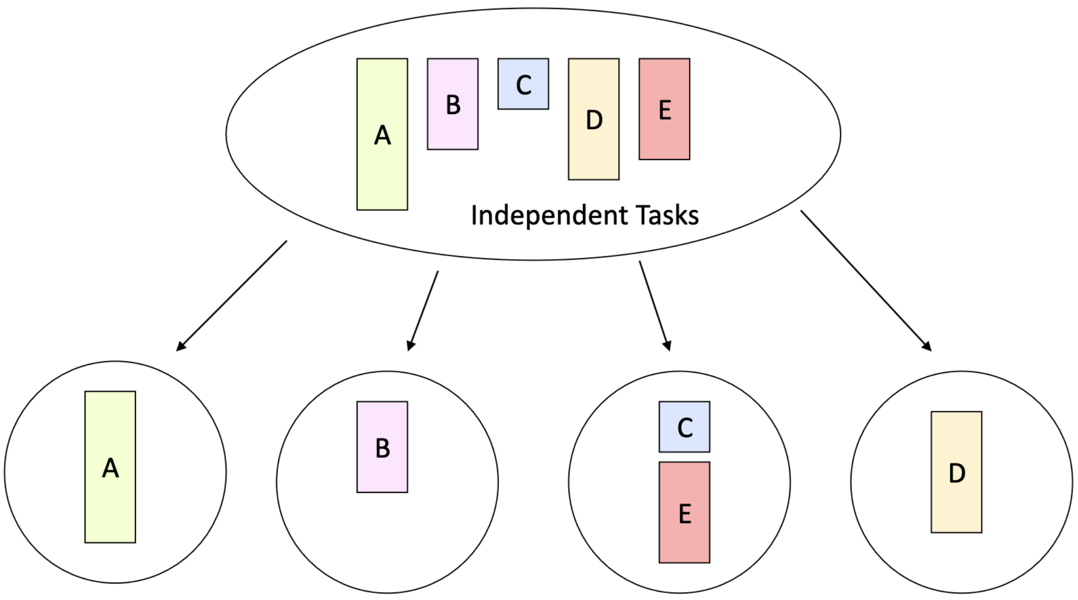
Load imbalance

- Load imbalance is more severe as the number of synchronization points increases.
- Static assignment of work to threads vs. Dynamic assignment of works (has its overhead)

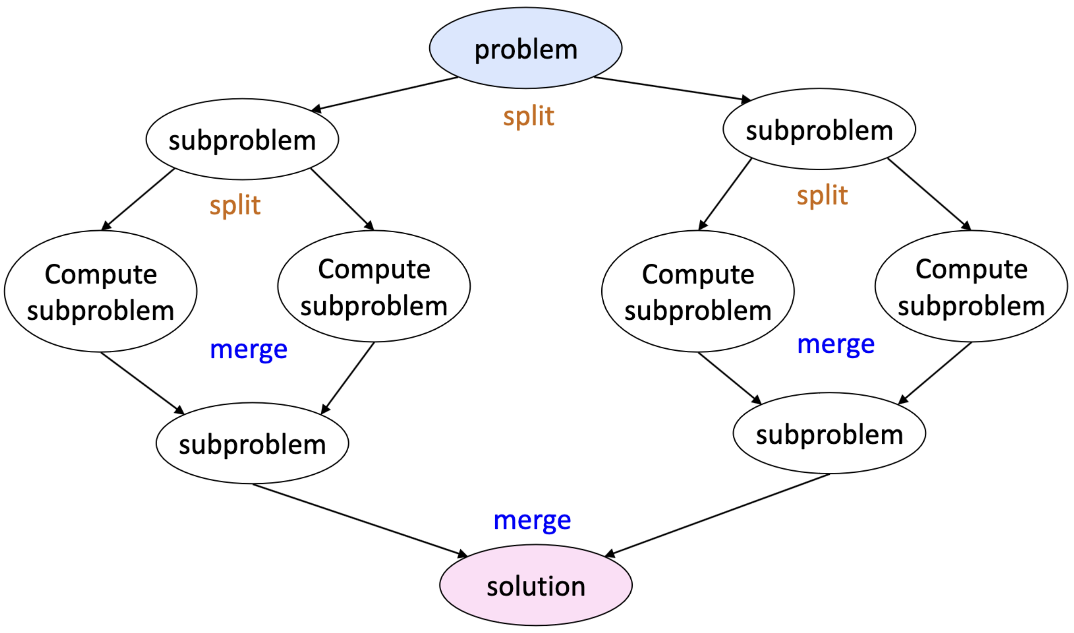
Patterns in Parallelism

Task-level (e.g. Embarrassingly parallel)

- Break application into tasks, decided offline (a priori).
- Generally, this scheme does not have strong scalability. (since we may have more cores than tasks)



Divide and conquer



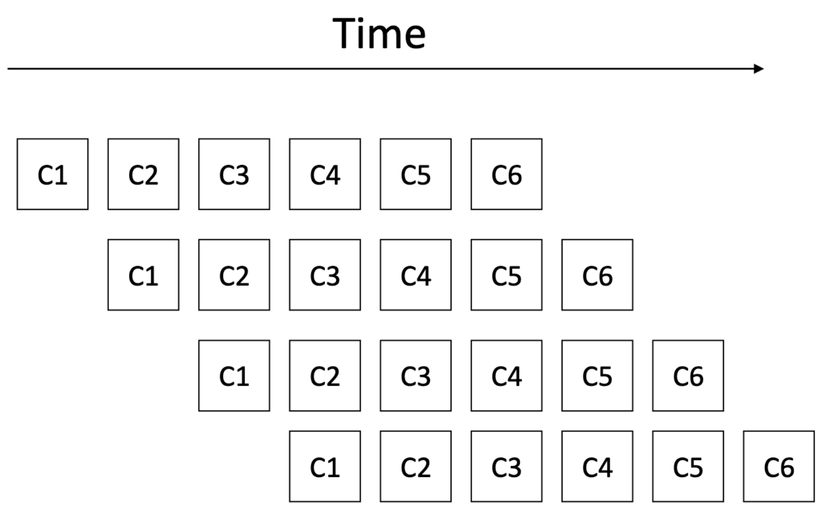
Pipeline

A series of ordered but independent computation stages need to be applied on data.

Useful for

- streaming workloads
- Loops that are hard to parallelize (due to inter-loop dependence)

1. Split each loop iteration into independent stages (e.g. C1, C2, C3, ...)
2. Assign each stage to a thread (e.g. T1 does C1, T2 does C2, ...)
3. When a thread is done with each stage, it can start the same stage for the following loop iteration (e.g. T1 finishes C1 of iteration 0, then start C1 of iteration 1, etc.)



Iterations (loops)

Client-server (repository model)

Geometric (usually domain dependent)

Hybrid (different program phases)