Lecture 03: Concurrency Decomposition

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

- Why multicores?
  - Moore's law and Dennard scaling
  - Multicore saves power
- Pthread based implementation of parallel Fibonacci
- Wait/Notify sequence using pthread

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  uint64_t input;
  uint64_t output;
} thread_args;

void *thread_func(void *ptr) {
  uint64_t i =
    ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

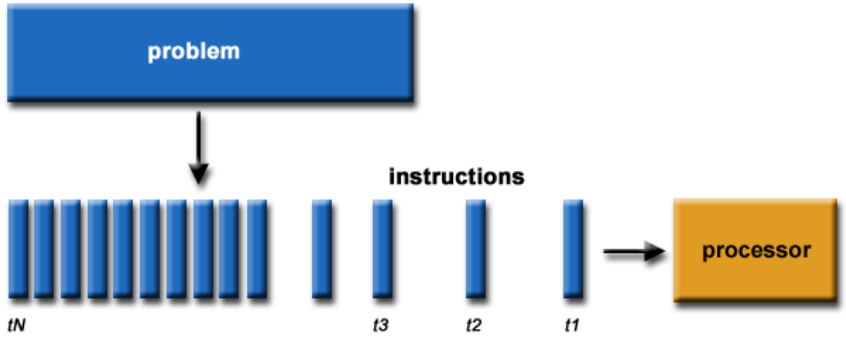
```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  uint64_t result;

  if (argc < 2) { return 1; }
  uint64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
   result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // Wait for the thread to terminate.
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRIu64 " is %" PRIu64 ".\n",
         n, result);
  return 0;
}
```

Today’s Class

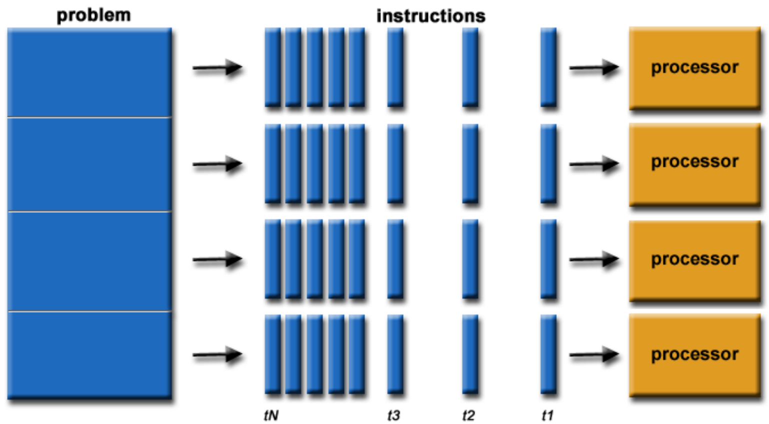
- Decomposition of sequential program into parallel program
    - Tasks and decomposition
    - Amdahl’s law
    - Tasks and mapping
    - Decomposition techniques
        - Recursive
        - Data
        - Exploratory
        - Speculative

Concurrency v/s Parallelism

- Concurrency
  - "**Dealing**" with lots of things at once



- Parallelism
  - "**Doing**" with lots of things at once



Concurrency v/s Parallelism

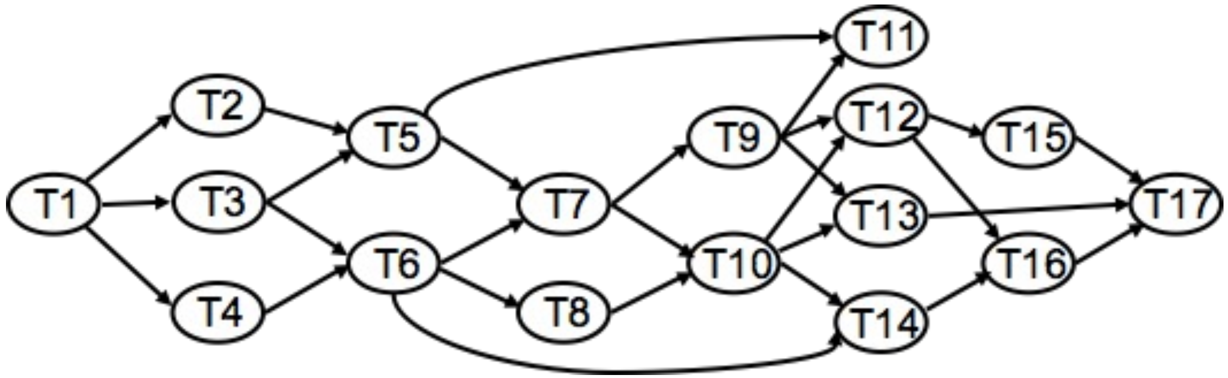
- Concurrency
  - Refers to tasks that appear to be running simultaneously, but which may, in fact, actually be running serially
- Parallelism
  - Refers to concurrent tasks that actually run at the same time
  - Always implies multiple processors
  - Parallel tasks always run concurrently, but not all concurrent tasks are parallel

Recipe to Solve a Problem using Parallel Programming

- **Typical** steps for constructing a parallel algorithm
  - identify what pieces of work can be performed concurrently
  - partition concurrent work onto independent processors
  - distribute a program's input, output, and intermediate data
  - coordinate accesses to shared data: avoid conflicts
  - ensure proper order of work using synchronization
- Why "**typical**"? Some of the steps may be omitted.
  - if data is in shared memory, distributing it may be unnecessary
  - the mapping of work to processors can be done statically by the programmer or dynamically by the runtime

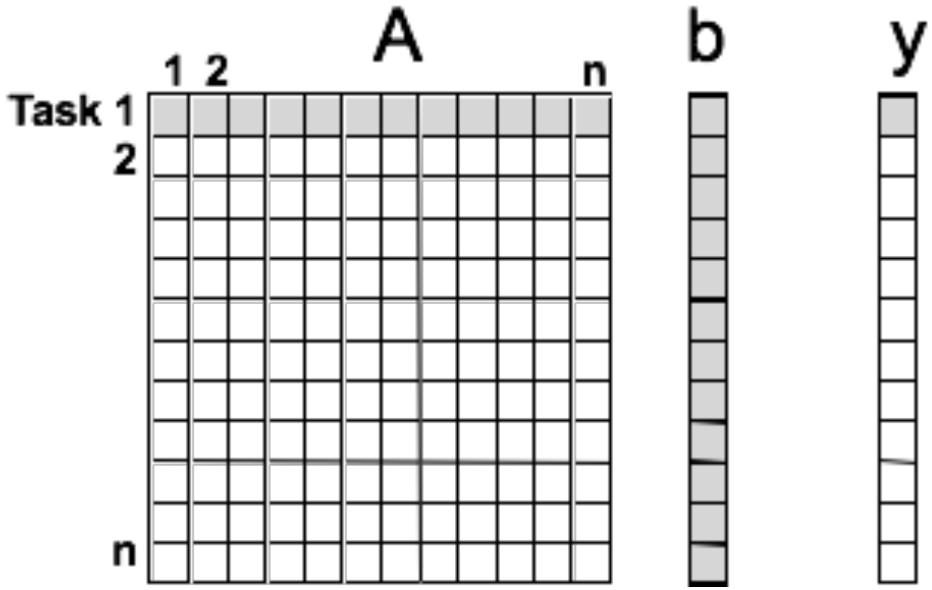
Decomposing Work for Parallel Execution

- Divide work into tasks that can be executed concurrently
- Many different decompositions possible for any computation
- Tasks may be same, different, or even indeterminate sizes
- Tasks may be independent or have non-trivial order
    - Conceptualize tasks and ordering as computation graph
        - Node = task
        - Edge = control dependency



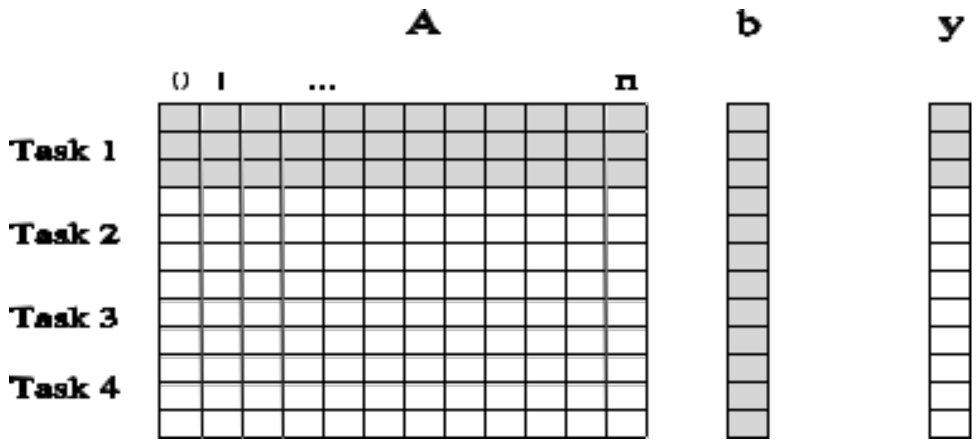
Example: Dense Matrix Vector Product

- Computing each element of output vector y is independent
- Easy to decompose dense matrix-vector product into tasks
  - one per element in y
- Observations
  - task size is uniform
- no control dependences between tasks
  - tasks share b



Granularity of Task Decomposition

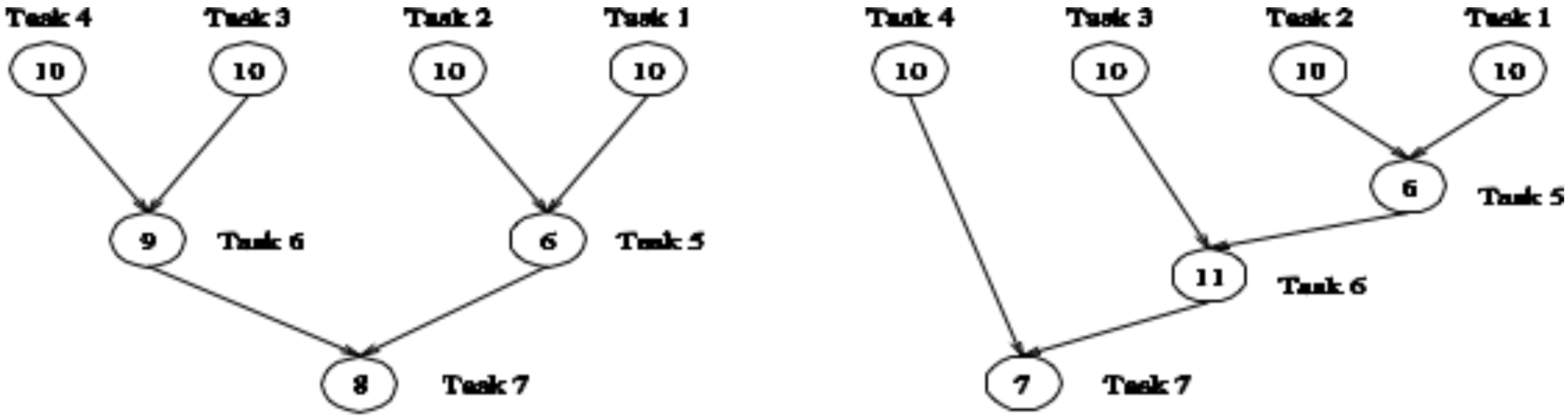
- Granularity = task size
  - depends on the number of tasks
- Fine-grain = large number of tasks
- Coarse-grain = small number of tasks
- Granularity examples for dense matrix-vector multiply
  - fine-grain: each task represents an individual element in y
  - coarser-grain: each task computes 3 elements in y



Critical Path

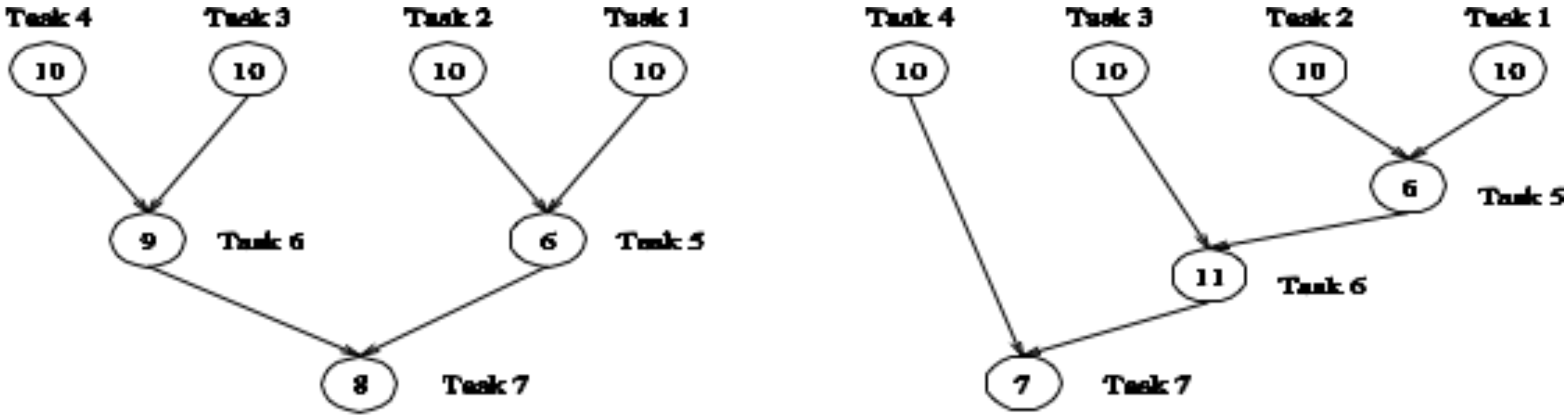
- Edge in computation graph represents task serialization
- Critical path = longest weighted path though graph
- Critical path length = lower bound on parallel execution time

Critical Path Length



Note: number in vertex represents task cost

Critical Path Length



Note: number in vertex represents task cost

Questions:

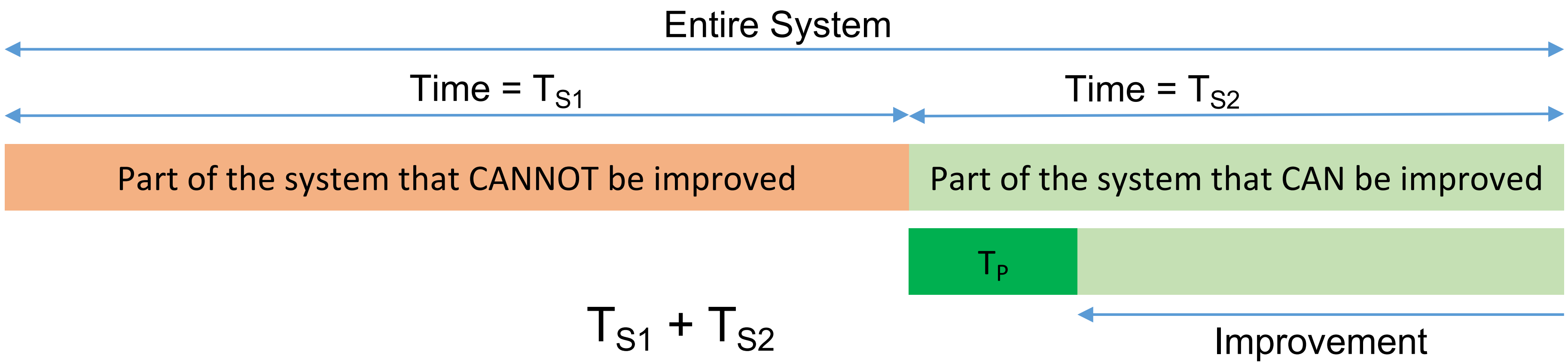
- What are the tasks on the critical path for each dependency graph?
- What is the shortest parallel execution time for each decomposition?

Limits on Parallel Performance

- What bounds parallel execution time?
  - minimum task granularity
    - e.g. dense matrix-vector multiplication ≤ n^2 concurrent tasks
  - dependencies between tasks
  - parallelization overheads
    - e.g., cost of communication between tasks
  - fraction of application work that can’t be parallelized
    - Amdahl’s law

Amdahl's Law

- Gives an estimate of maximum expected improvement S to an overall system when only part of the system F_E is improved by a factor F_I



$$\text{Speedup using P processes} = \frac{T_{S1} + T_{S2}}{T_{S1} + T_P}$$

Speedup Analysis

1. Do disproportionately less work
2. Harness disproportionately more resources

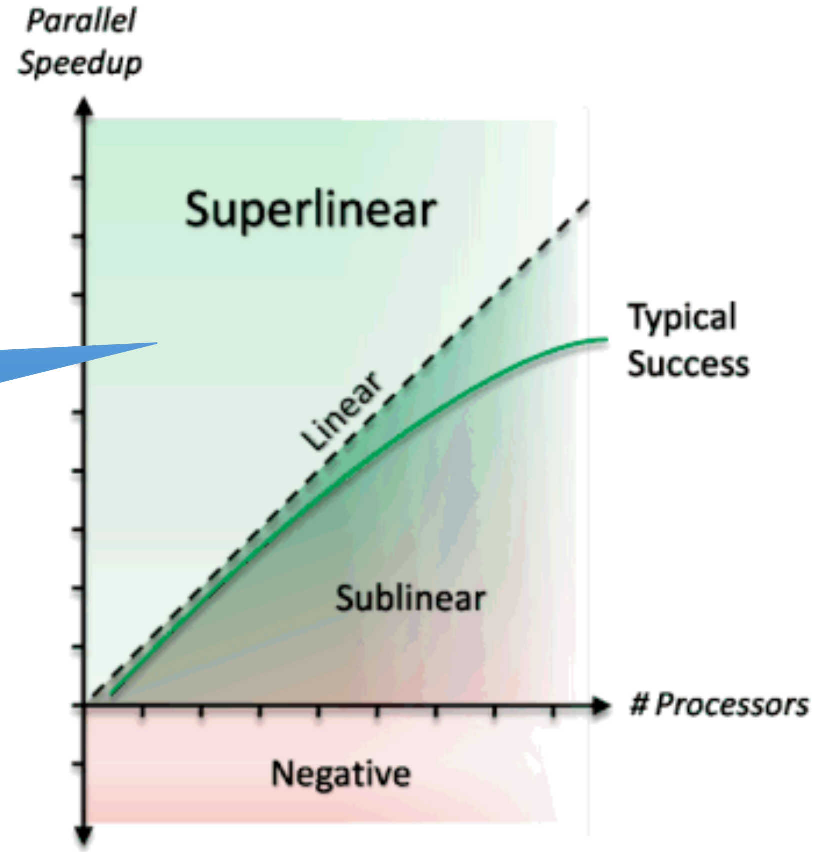


Fig. source: http://www.drdobbs.com/cpp/going-superlinear/206100542

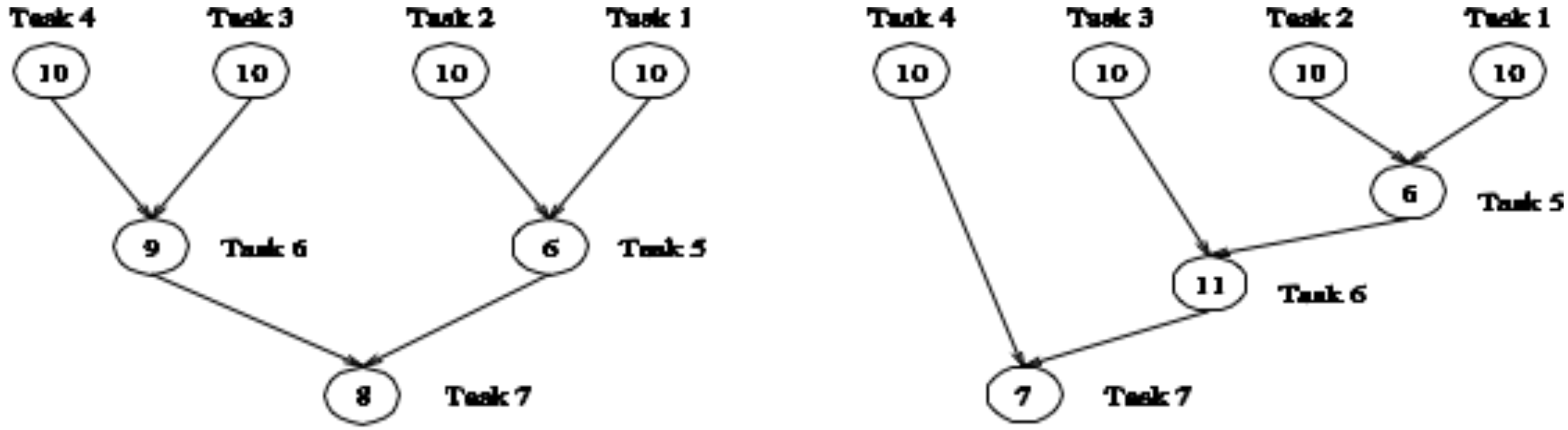
Today's Class

- Decomposition of sequential program into parallel program
  - Tasks and decomposition
  - Amdahl's law
  - Tasks and mapping
  - Decomposition techniques
    - Recursive
    - Data
    - Exploratory
    - Speculative

Mapping Tasks to Cores

- Generally
  - # of tasks > # threads available
  - parallel algorithm must map tasks to threads
  - schedule independent tasks on separate threads (consider computation graph)
  - threads should have minimum interaction with one another

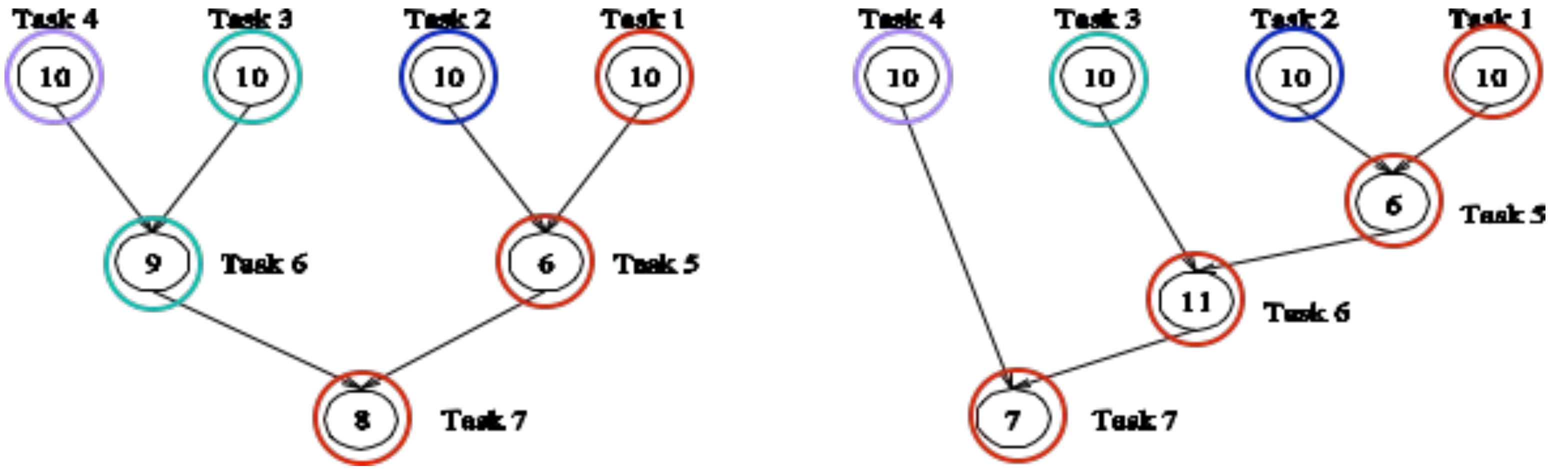
Tasks, Threads, and Mapping Example



Note: number in vertex represents task cost

- How to best map these tasks on threads?

Tasks, Threads, and Mapping Example



- No tasks in a level depend upon each other
- Assign all tasks within a level to different threads

Mapping Techniques

Static vs. dynamic mappings

- Static mapping
  - *a-priori* mapping of tasks to threads or processes
    - requirements
      - a good estimate of task size
      - even so, computing an optimal mapping may be hard
- Dynamic mapping
    - map tasks to threads or processes at runtime
    - why?
      - tasks are generated at runtime, or
      - their sizes are unknown

Static Mapping

- Data partitioning
- Computation graph partitioning

Dynamic Mapping

- Dynamic mapping AKA dynamic load balancing
  - load balancing is the primary motivation for dynamic mapping
- Styles
  - centralized
  - distributed

Today’s Class

- Decomposition of sequential program into parallel program
  - Tasks and decomposition
  - Amdahl’s law
  - Tasks and mapping
  - Decomposition techniques
    - Recursive
    - Data
    - Exploratory
    - Speculative

Decomposition Techniques

How should one decompose a task into various subtasks?

- No single universal recipe
- In practice, a variety of techniques are used including
  - Data decomposition
  - Recursive decomposition
  - Exploratory decomposition
  - Speculative decomposition

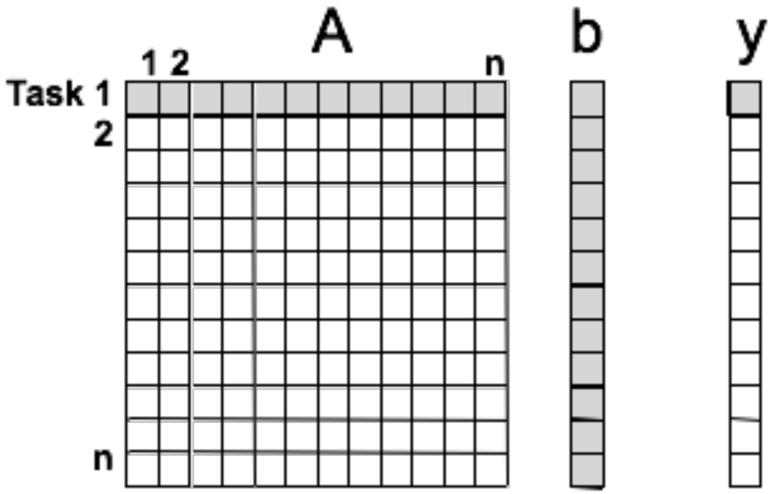
Data Decomposition

- Steps
  1. identify the data on which computations are performed
  2. partition the data across various tasks
     - partitioning induces a decomposition of the problem

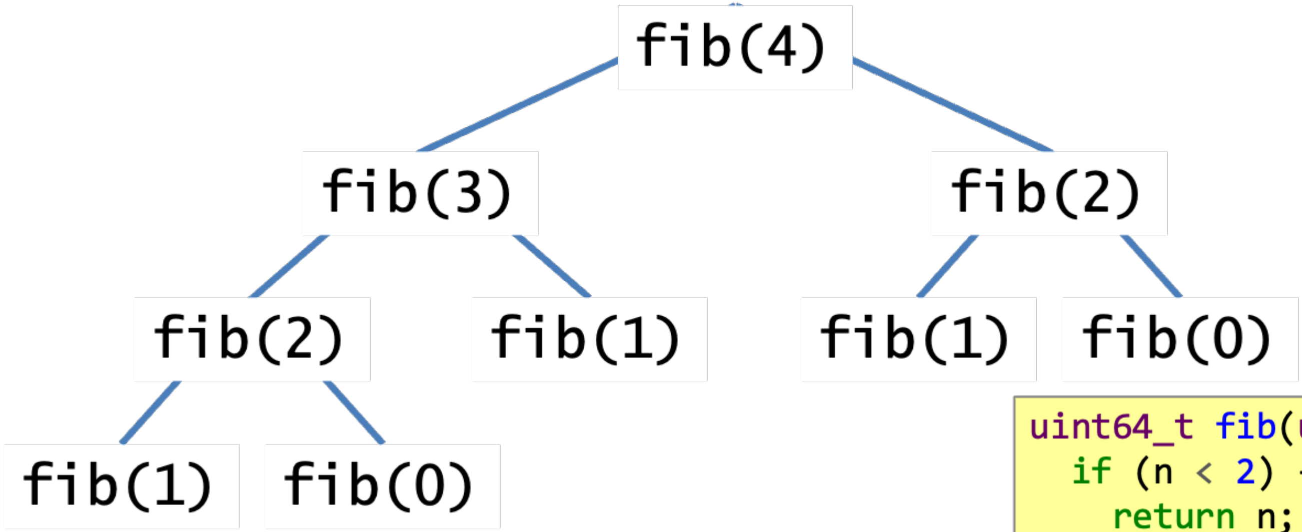
Data Decomposition Example

- If each element of the output can be computed independently
- Partition the output data across tasks
- Have each task perform the computation for its outputs

Example:
dense matrix-vector multiply



Recursive Decomposition




```
uint64_t fib(uint64_t n) {
  if (n < 2) {
    return n;
  } else {
    uint64_t x = fib(n-1);
    uint64_t y = fib(n-2);
    return (x + y);
  }
}
```

Question: what kind of mapping is suited for this scenario?

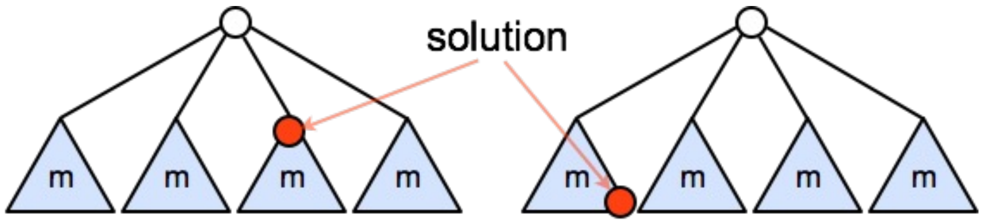
DAG Source: http://www.cs.ucsb.edu/projects/jicos/tutorial/fibonacci/index.html

Exploratory Decomposition

- Exploration (search) of a state space of solutions
  - Problem decomposition reflects shape of execution
  - Parallel formulation may perform a different amount of work

Exploratory Decomposition Speedup

- Parallel formulation may perform a different amount of work
  - **Can cause super- or sub-linear speedup**
- Assume each vertex of the triangles represents a computation that takes 'T' unit of time to compute and execution begins from leftmost triangle to the rightmost



- Serial execution time = 7 T
- Parallel execution time using 4 threads to compute each triangle in parallel = T
- Speedup (4 threads) = 7T/T = 7
- **Super**-linear speedup

- Serial execution time = 3 T
- Parallel execution time using 4 threads to compute each triangle in parallel = 3T
- Speedup (4 threads) = 3T/3T = 1
- **Sub**-linear speedup

Question

- How exploratory decomposition (ED) differs from data decomposition (DD)?
  1. Unlike ED, **all** partial tasks contribute to final result in DD
  2. Unlike DD, unfinished tasks in ED can be terminated once final solution is found

Speculative Decomposition

- Example: when program may take one of many possible compute-intensive branches depending on the output of preceding computation

```
int val = T1        //compute intensive
switch(val) {       // cases may be computed speculatively
        case 0: T2; break;
        case 1: T3; break;
        …..
        case n: Tn; break;
}
```

Next Lecture (#04)

- Productivity in parallel programming (tasks based parallel programming model)
- Quiz-1
  - Syllabus: Lectures 02 and 03