KOÇ UNIVERSITY

Task Graphs

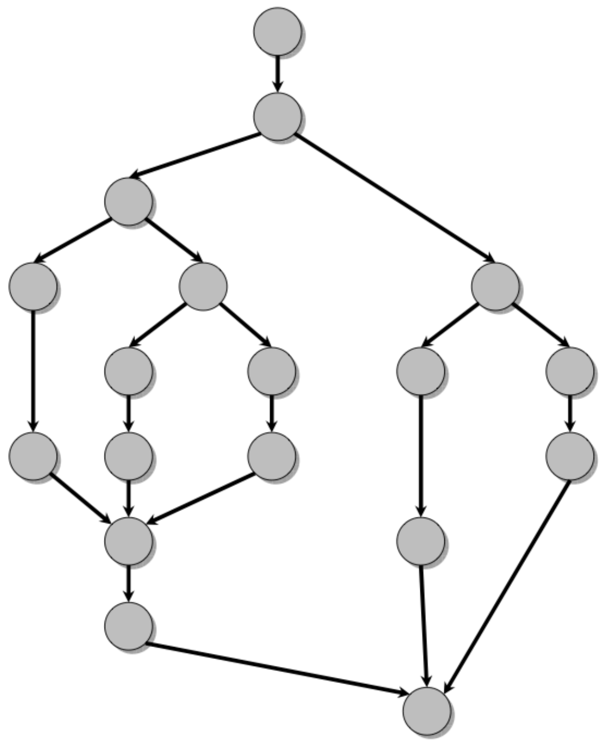
Didem Unat

COMP 429/529 Parallel Programming

Tasks Dependency Graph

- In task parallelism, the problem is decomposed into ***tasks*** that are candidates for parallel execution

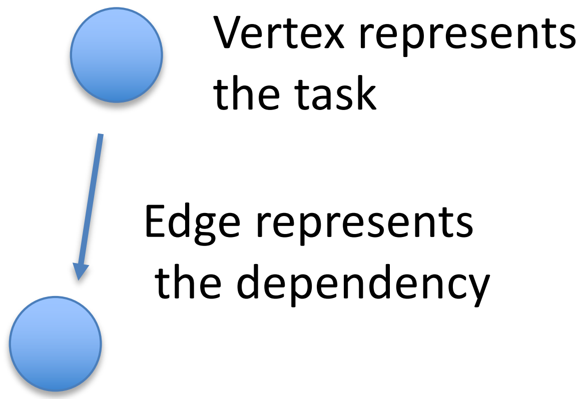
- A decomposition can be illustrated in the form of a **directed graph** with nodes
  - Such a graph is called a ***task dependency graph***.



Tasks Dependency Graph

Nodes(vertices)

- correspond to tasks
- Represents the computation



Edges

- Represents the dependency
- indicate that the result of one task is required for processing the next.

Critical Path

- A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other.

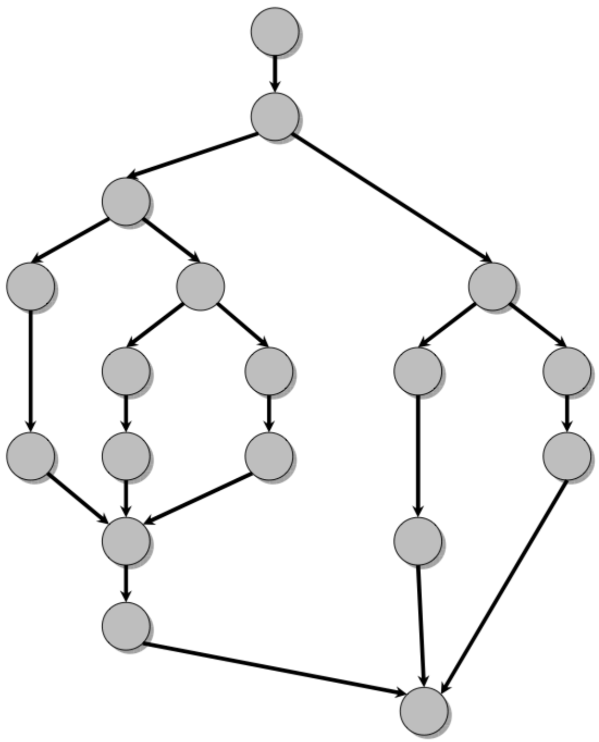
- The longest such path determines **the shortest time** in which the program can be executed in parallel.

- The length of the longest path in a task dependency graph is called **the *critical path length*.**

- The ratio of the total amount of work to the critical path length is the ***average degree of concurrency*.**

Algorithmic Complexity Measures

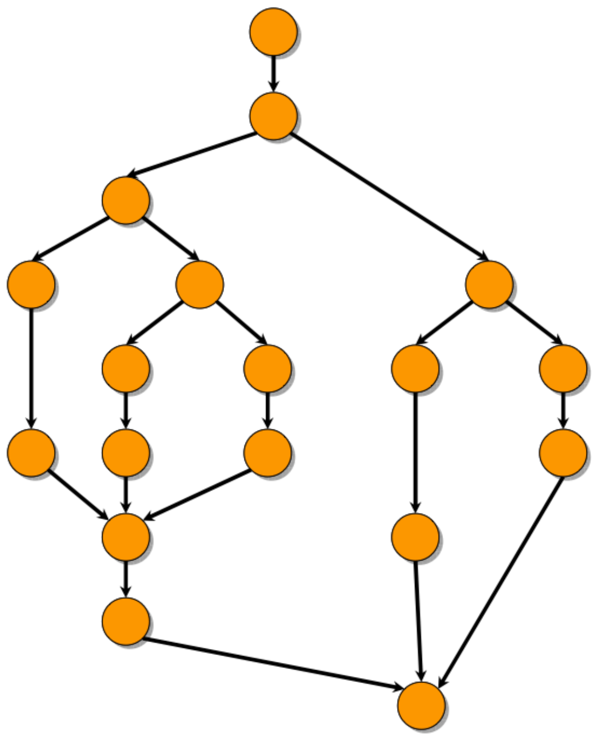
- Ignoring Communication Overhead
- ***Tp*** = execution time on ***p*** processors



Slide Src: Charles E. Leiserson

Algorithmic Complexity Measures

- Ignoring Communication Overhead
- ***Tp*** = execution time on ***p*** processors

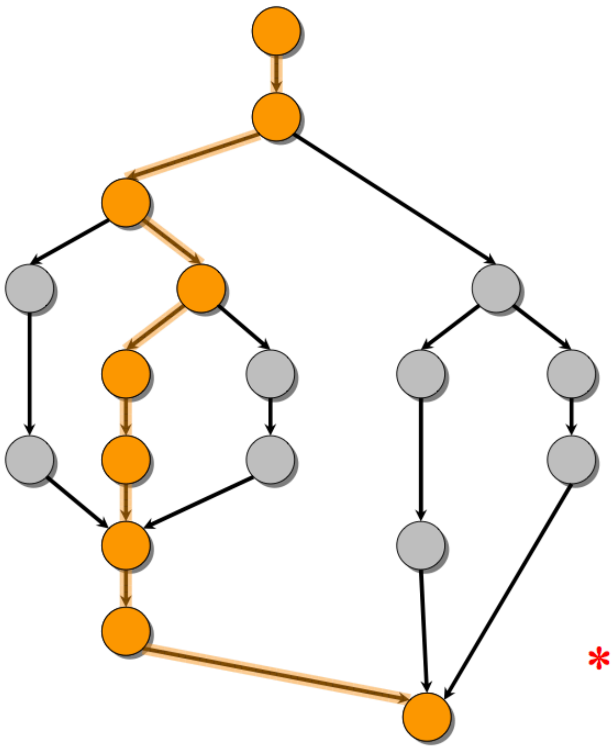


T_1 = ***work***

Slide Src: Charles E. Leiserson

Algorithmic Complexity Measures

- Ignoring Communication Overhead
- ***Tp*** = execution time on ***p*** processors



T_1 = ***work***

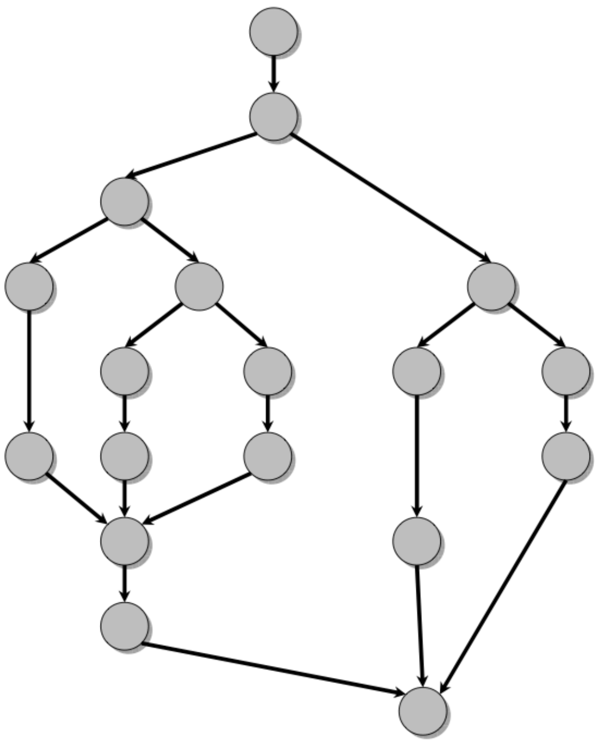
T_∞ = ***span****

* Span is also called ***critical-path length.***

Slide Src: Charles E. Leiserson

Algorithmic Complexity Measures

- Ignoring Communication Overhead
- ***Tp*** = execution time on ***p*** processors



T_1 = ***work***

T_∞ = ***span****

LOWER BOUNDS

- $T_P \geq T_1/P$
- $T_P \geq T_\infty$

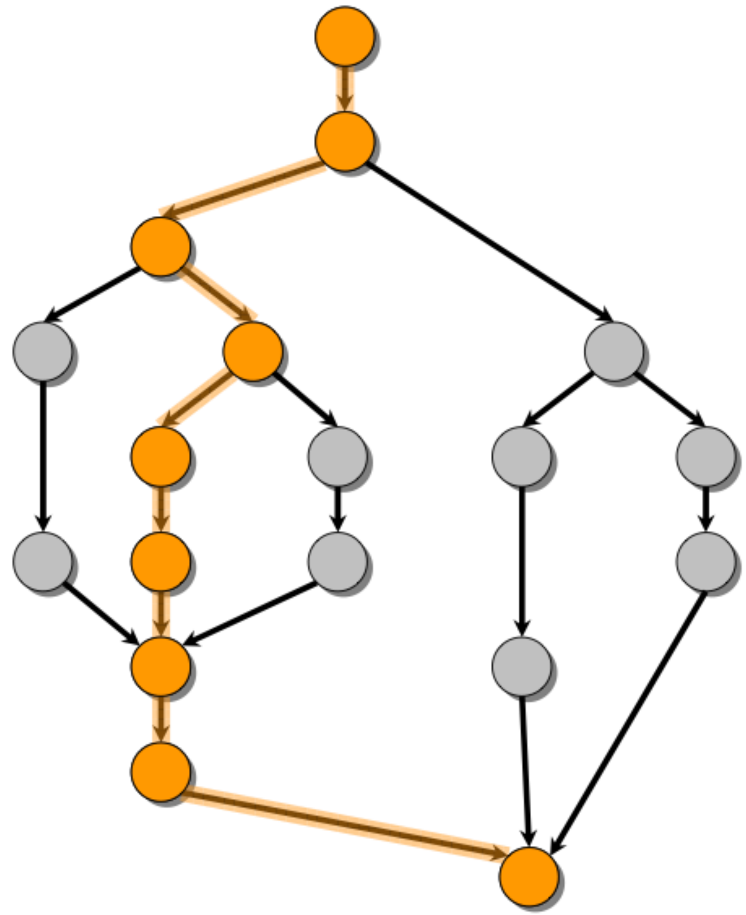
Slide Src: Charles E. Leiserson

Parallelism

- Average degree of concurrency:
  - Because we have the lower bound T_p T_∞, the maximum possible speedup given T_1 and T_∞ is

T_1/T_∞ = ***parallelism***

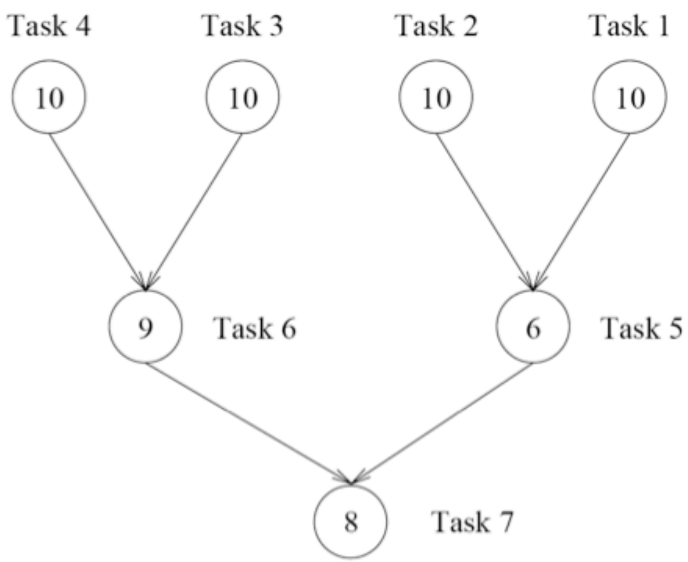
= the average degree of ***concurrenc***



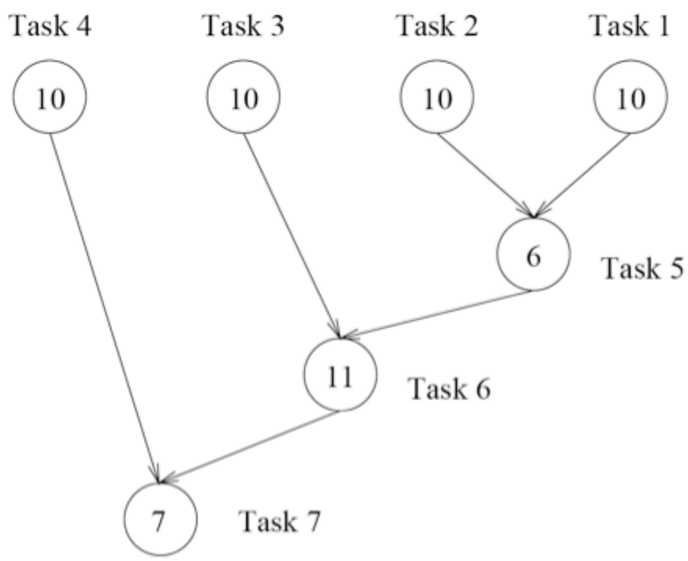
Tall graph or fat graph?, What does a tall, skinny graph imply?

Examples of Critical Path Length

- Consider the task dependency graphs of the two graph decompositions
- Execution time of each task indicated inside the node



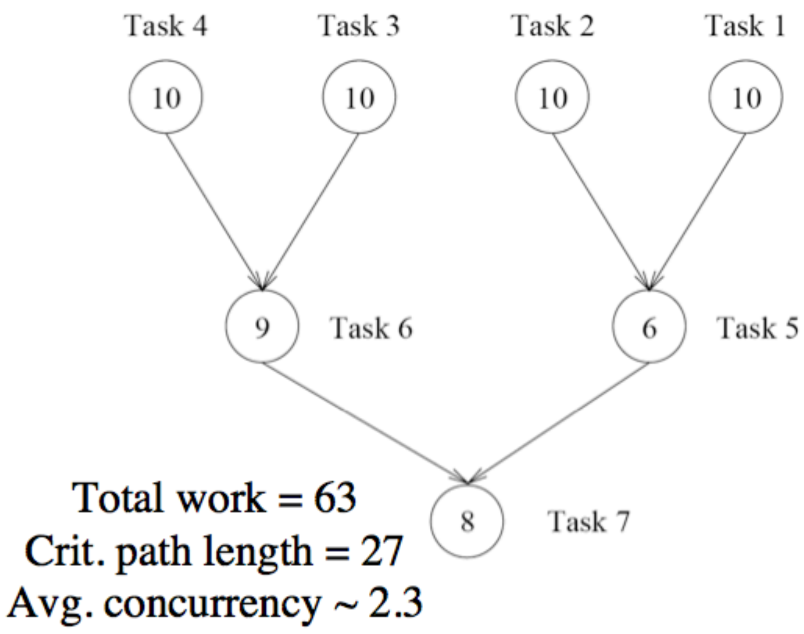
(a)



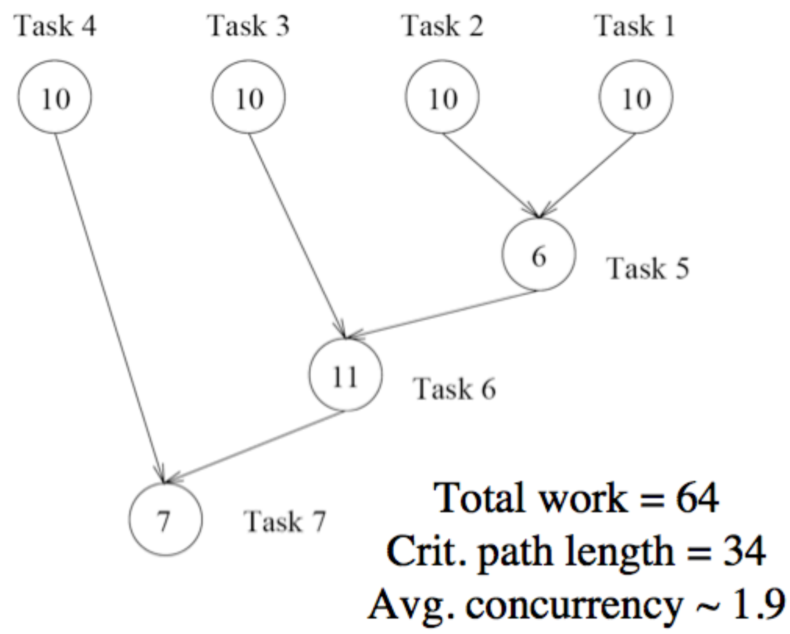
(b

Examples of Critical Path Length

- Consider the task dependency graphs of the two graph decompositions
- Execution time of each task indicated inside the node

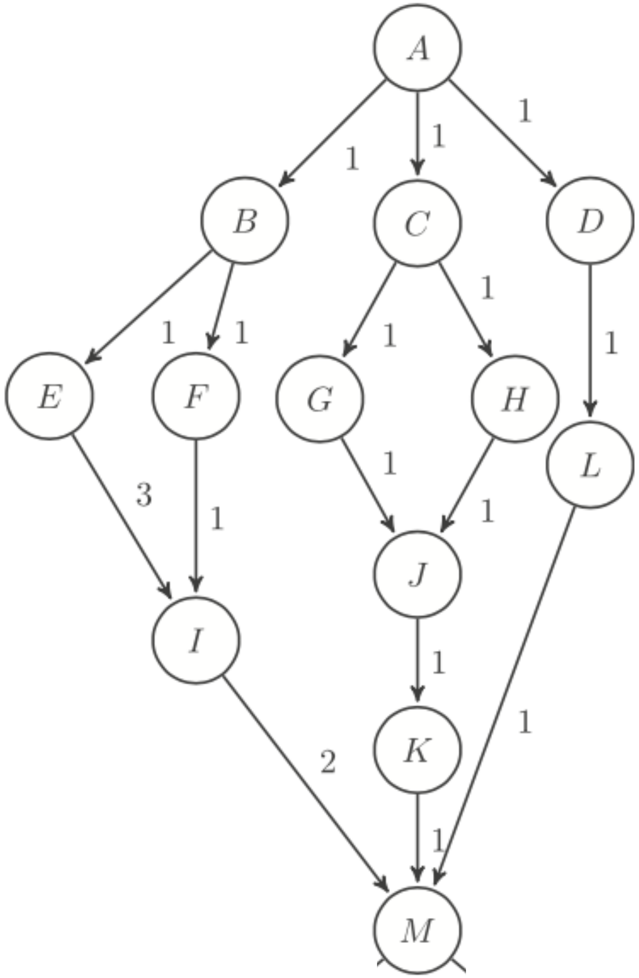


(a)



(b)

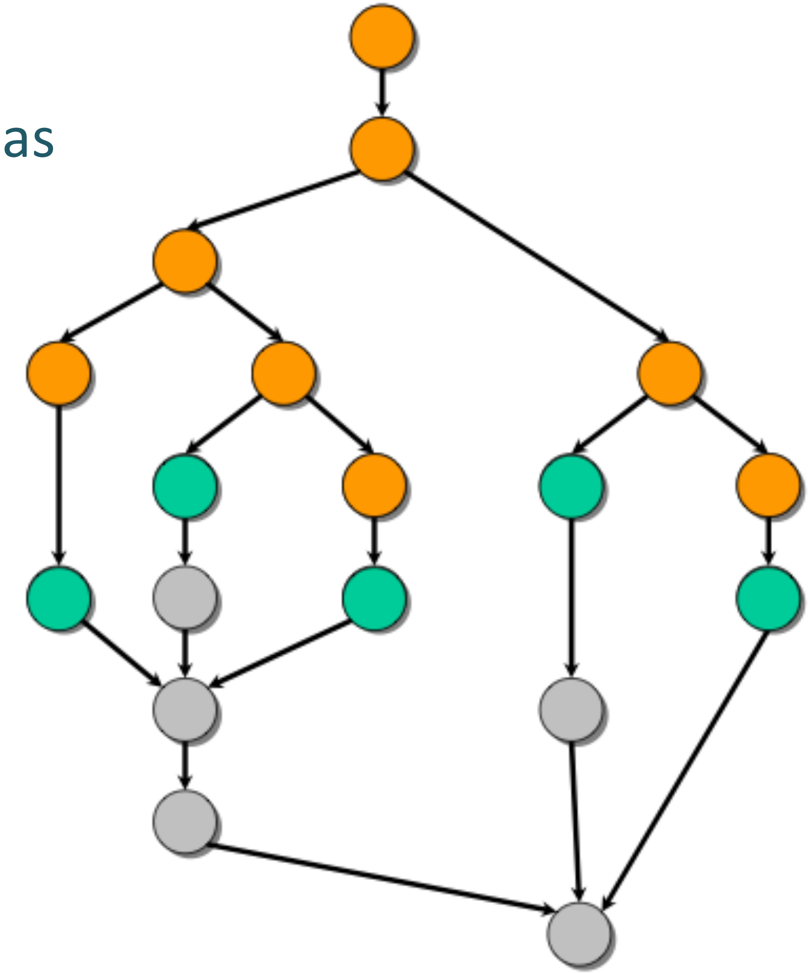
Critical Path Example



- Assume each node has the same unit of work (1 unit)
- Path 1: A, C, G, J, K, M
- Path 2: A, B, E, I, M
- Path 2 incurs more communication time thus it is the critical path (longer to execute)

Scheduling Tasks

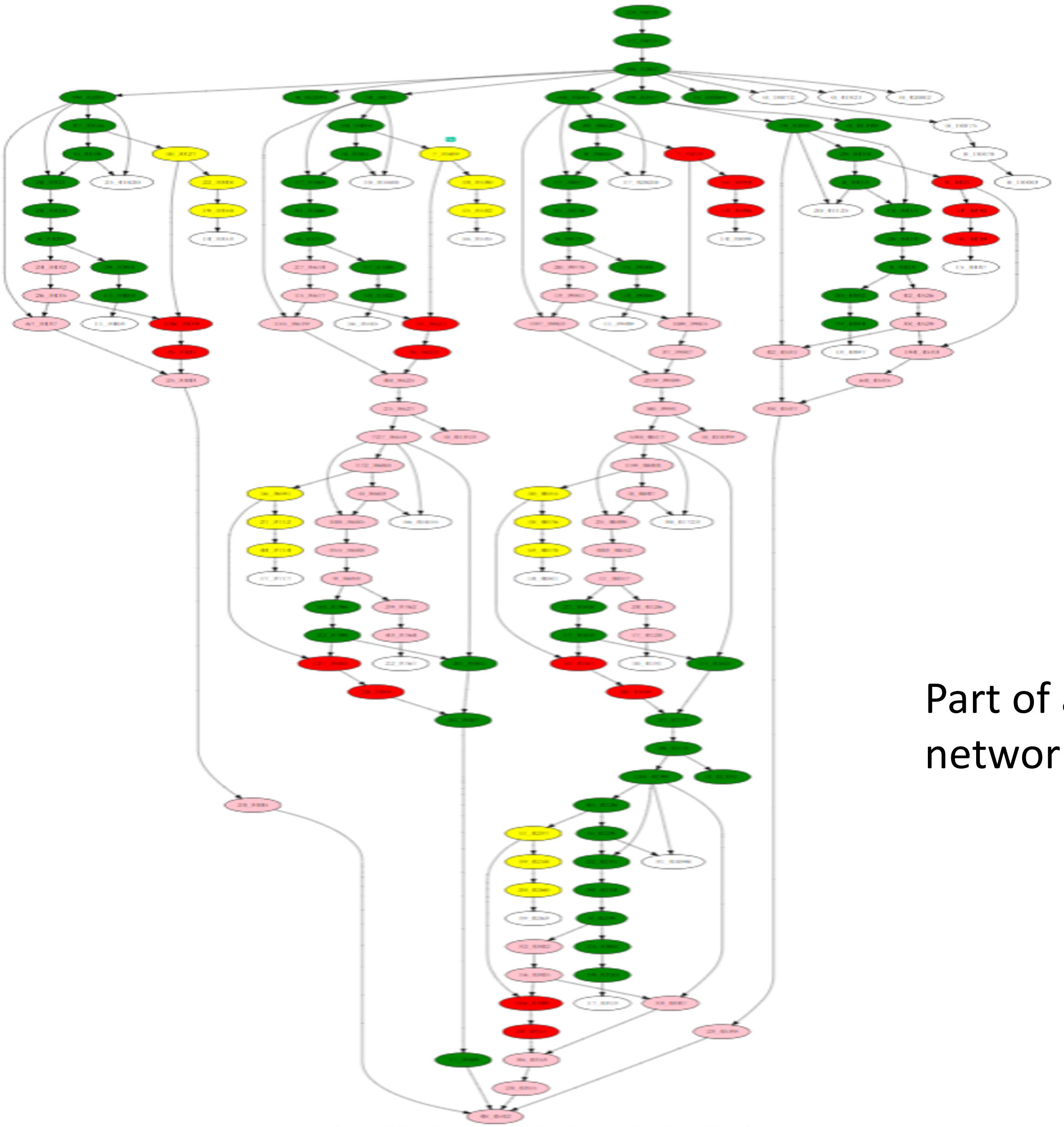
- A runtime usually responsible for scheduling ready tasks
- **Scheduling Idea:** Increase **parallelism** as much as possible on every step
- A task is **ready** if all its predecessors have executed
- **Complete Tasks**
- **Incomplete Tasks**
- **Ready Tasks**
- How would you schedule the ready tasks?



Deep Learning Example

- TensorFlow and many other DL frameworks represent the neural network operations as a DAG (directed acyclic graph).
  - Nodes represent computation
  - Edges represent the communication
- TensorFlow schedules a task when its dependencies are met.

Examples- TensorFlow

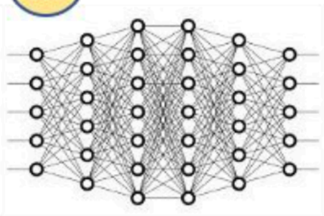


Part of a neural network graph

Deep Learning Example

Deep Learning Model

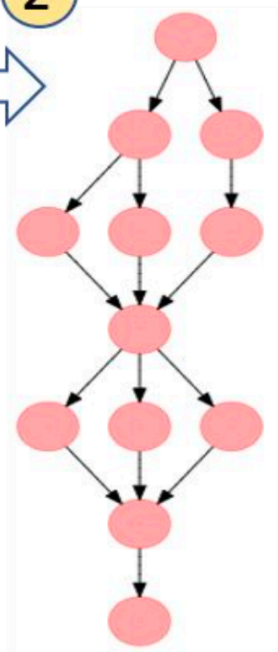
1

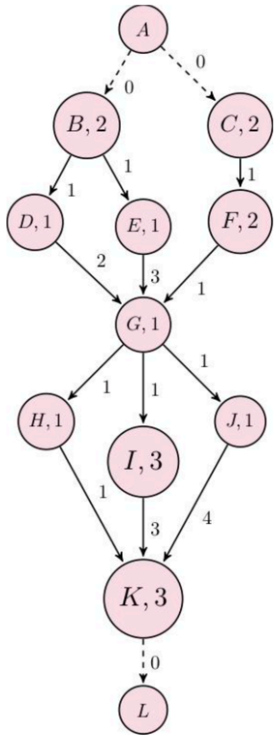


Python, C/C++, Java

Computational Graph

2





- *G(V,E)*: Task graph
- *V*
  - *n* ∈ *V*: Task.
  - *w(n)*: weight of n, computation time
- *E*
  - *e* ∈ *E*: Dependency.
  - *c(e):* cost of e, communication time
  - Defines the execution order

Task Graph Creation

- Task graphs can be generated *statically* or *dynamically*

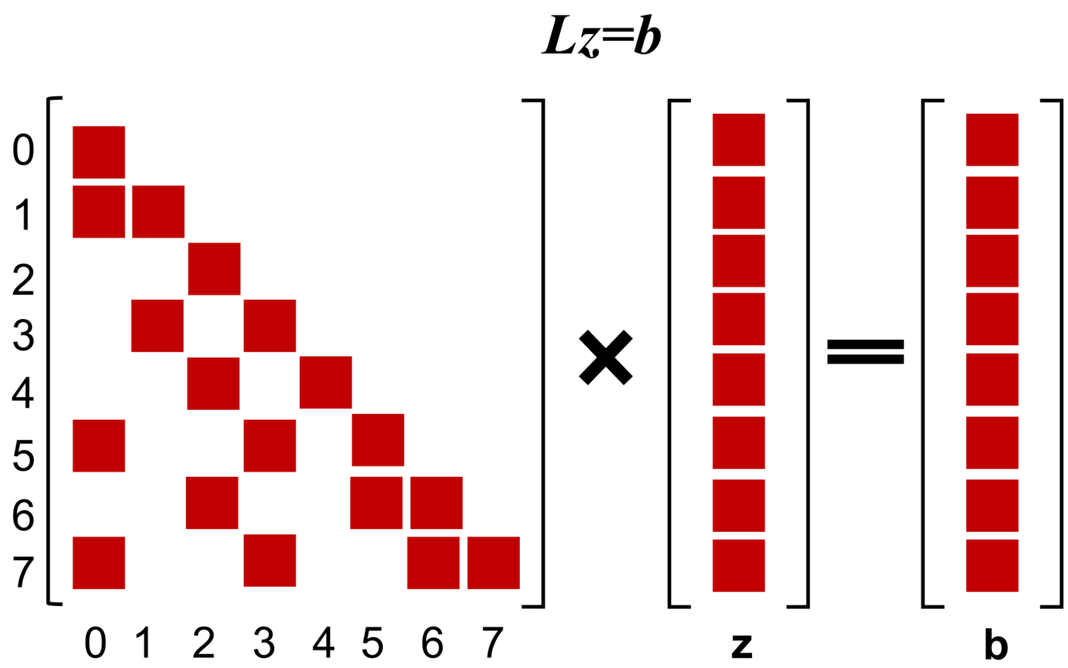
- **Static graphs:** do not change its structure throughout the execution, easier to reason about its schedule and perform optimizations
- **Dynamic graphs:** change its structure over the execution, need to reconstruct the graph, perform optimizations as the task graph unfolds

Real DNN Graphs

Model	Acronym	#Layers	HSD	SL	#Parameters	#Graph Nodes	Dataset
Recurrent Neural Network for Word-Level Language [51]	Word-RNN	10	2048	28	0.44 billion	11744	Tiny Shakespeare [23]
	Word-RNN-2	8	4096	25	1.28 billion	10578	
		#Layers	CHSD	ED			
Character-Aware Neural Language Models [26]	Char-CRN	8	2048	15	0.23 billion	22748	Penn Treebank (PTB) [33]
	Char-CRN-2	32	2048	15	1.09 billion	86663	
		#Conv. Layers [65]	#RU	WF			
Wide Residual Network [64]	WRN	610	101	14	1.91 billion	187742	CIFAR100 [28]
	WRN-2	304	50	28	3.77 billion	79742	
		#Layers	HSD	MD			
Transformer [54]	TRN	24	5120	2048	1.97 billion	80550	IWSLT 2016 German–English corpus [6]
	TRN-2	48	8192	2048	5.1 billion	160518	
		#Hidden Layers	FS				
Eidetic 3D LSTM[58]	E3D	320	5		0.95 billion	55756	Moving MNIST digits [50]
	E3D-2	512	5		2.4 billion	55756	

Number of operations reaches hundreds of thousands, may scale up to millions.

Examples- SpTRSV



Ax=b where A is ***A=LU***

then ***Ax=b:***

(LU)x=b

z=Ux

then ***Ax=b:***

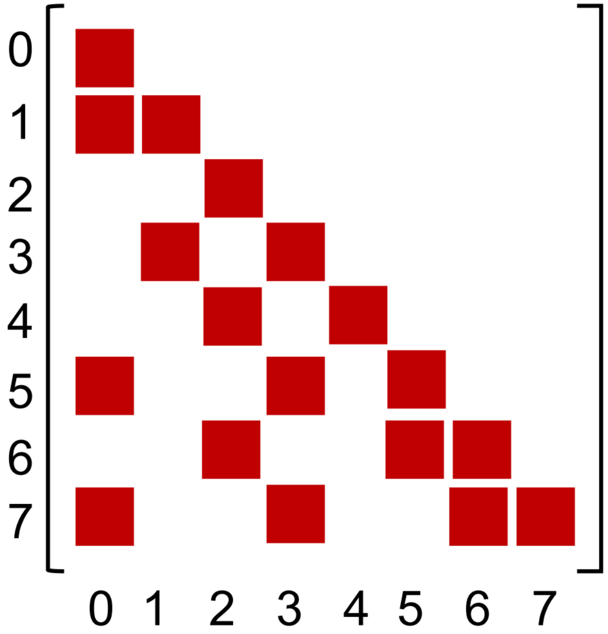
Lz=b

→ forward substitution

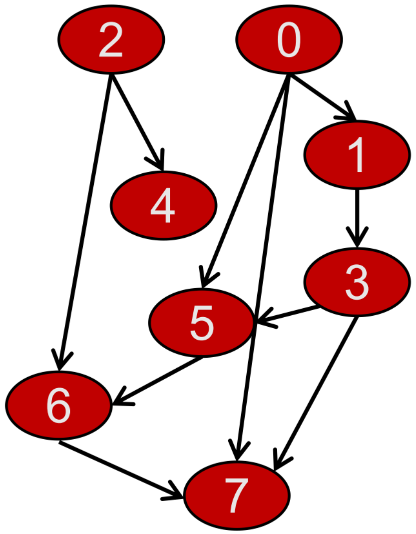
→ backward substitution (***z=Ux***)

SpTRSV- Sparse Tridiagonal Solve

Examples- SpTRSV



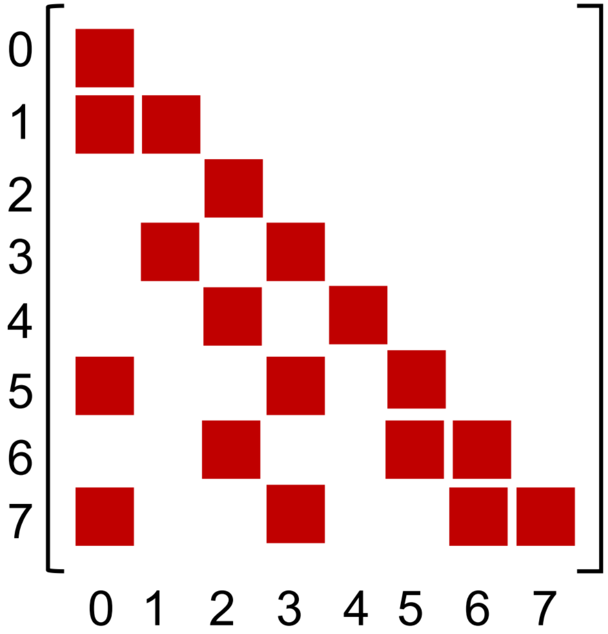
Lower Triangular matrix L



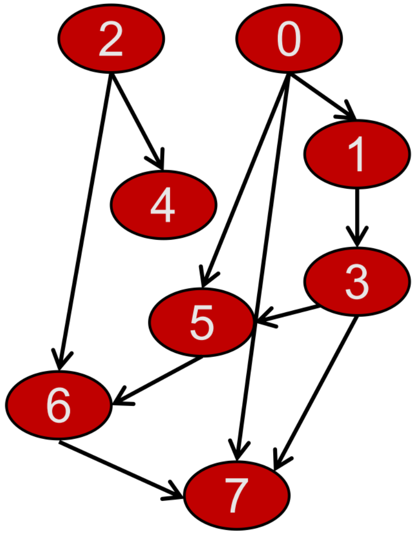
DAG of L

- Represented as a task graph
- For example, task 5 can execute only task 0 and 3 are finished.

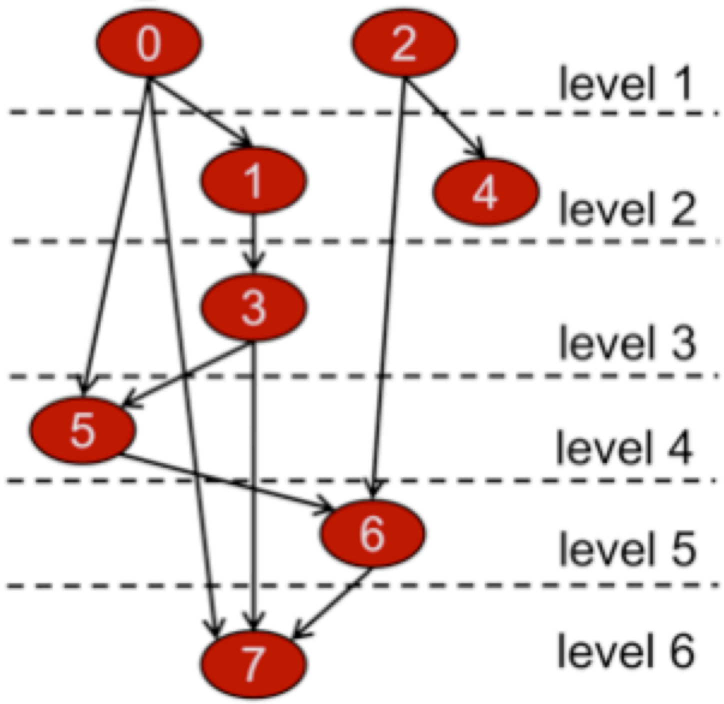
Examples- SpTRSV



Lower Triangular matrix L



DAG of L

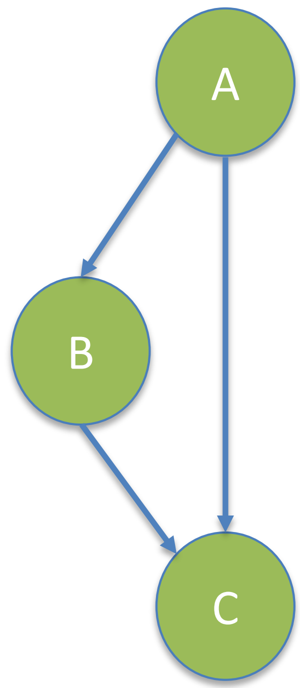


DAG of L partitioned into level-sets

- Level-set method
  - Each level executes in parallel (node 0 and 2 are independent)
  - At the end of a level, synchronize – before advancing into the next level

Sparcifying DAG

- We can potentially remove redundant dependencies
- Dependencies are transitive
  - If B depends on A, C depends on B and A
  - Remove the dependency edge between C and A
  - This is to help the scheduler
  - If data needs to be transfer from A to C, we should still do it



The depend Clause in OpenMP

- The *task dependence* is fulfilled when the predecessor task has completed
  - `in` dependency-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` clause.
  - `out` and `inout` dependency-type: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` clause.
- The list items in a depend clause may include array sections.

Task Dependences - RAW

```
#include <stdio.h>
int main()
{
    int x= 1;
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task shared(x) depend(out: x)
          x = 2;
        #pragma omp task shared(x) depend(in: x)
          printf("x = %d\n", x);
    }
    return 0;
}
```

- The program will always print "x = 2", because the **depend** clauses enforce the ordering of the tasks. If the **depend** clauses had been omitted, then the tasks could execute in any order and the program would have a race condition.

Task Dependences - WAR

```
#include <stdio.h>
int main()
{
    int x= 1;
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task shared(x) depend(in: x)
          printf("x = %d\n", x);

        #pragma omp task shared(x) depend(out: x)
          x = 2;
    }
    return 0;
}
```

- The program will always print "x = 1", because the **depend** clauses enforce the ordering of the tasks. If the **depend** clauses had been omitted, then the tasks could execute in any order and the program would have a race condition.

Task Dependences - WAW

```
#include <stdio.h>
int main()
{
    int x= 1;
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task shared(x) depend(out: x)
          x = 1;
        #pragma omp task shared(x) depend(out: x)
          x = 2;
         #pragma omp taskwait
        printf("x = %d\n", x);
    }
    return 0;
}
```

- The program will always print "x = 2", because the **depend** clauses enforce the ordering of the tasks. If the **depend** clauses had been omitted, then the tasks could execute in any order and the program would have a race condition.

The depend Clause

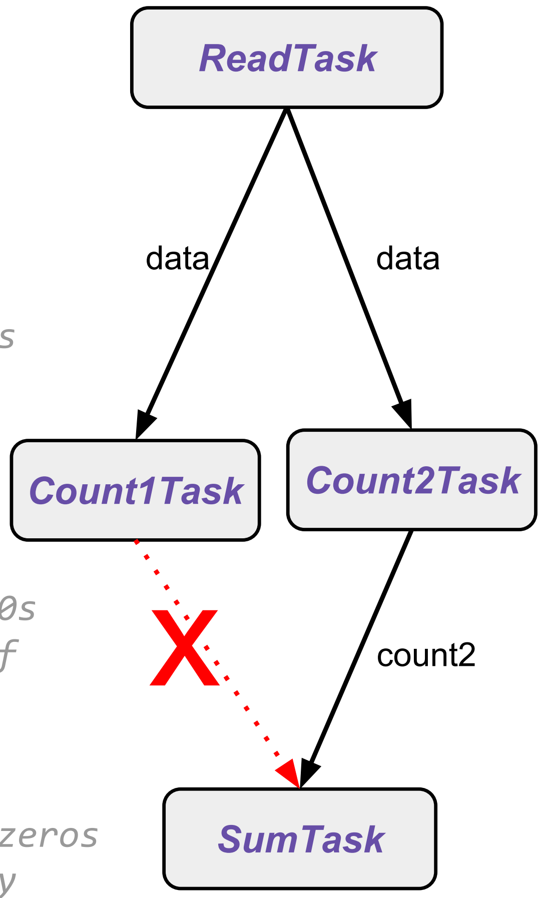
```
#pragma omp parallel
#pragma omp single
{
   int x = 1;
   for (int i = 0; i < T; ++i) {
   #pragma omp task shared(x, ...) depend(out: x)// T1
      preprocess_some_data(...);

   #pragma omp task shared(x, ...) depend(in: x) // T2
      do_something_with_data(...);

   #pragma omp task shared(x, ...) depend(in: x) // T3
      do_something_independent_with_data(...);
   }
} // end omp single, omp parallel
```

T1 has to complete before T2 and T3 can start
T2 and T3 can be executed in parallel.

Missing Dependency

```
#pragma omp task depend(out:data)
{ /**    ReadTask   */
   data = new int[N];             // set memory
   ReadData(“input.txt”, data);   // read data
}
#pragma omp task depend(in:data)
{ /**    Count1Task */
   count1 = 0;
   for(int i = 0; i < N/2; i++)   // count non 0s
     if(data[i]) count1++;        // in 1st half
}
#pragma omp task depend(in:data, out:count2)
{ /**    Count2Task */
   count2 = 0;
   for(int i = N/2+1; i < N; i++) // count non 0s
     if(data[i]) count2++;         // in 2nd half
}
#pragma omp task depend(in:count2)
{ /**    SumTask    */
   numNonZeros = count1 + count2; // total non-zeros
   delete data;                   // free memory
}
```

(a)

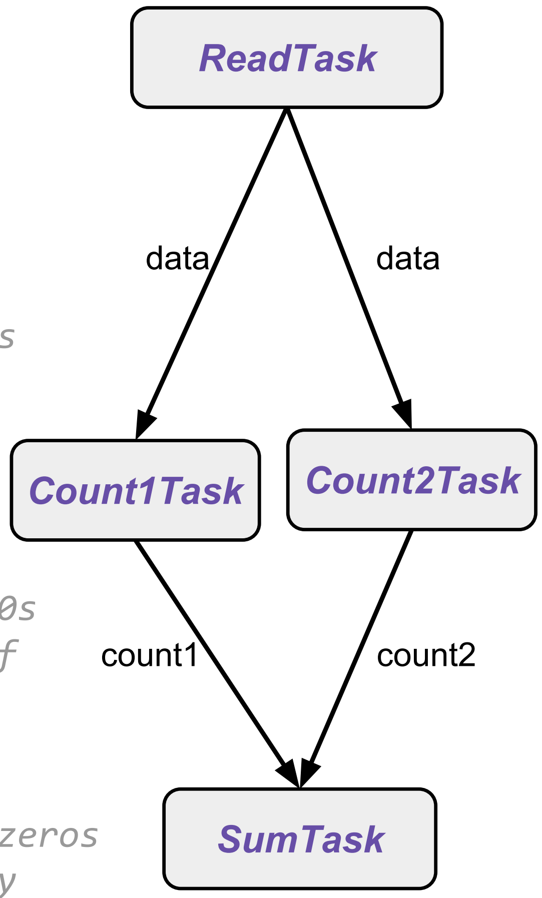


(b)

Task Dependency

```
#pragma omp task depend(out:data)
{ /**    ReadTask   */
   data = new int[N];              // set memory
   ReadData(“input.txt”, data);    // read data
}
#pragma omp task depend(in:data, out:count1)
{ /**    Count1Task */
   count1 = 0;
   for(int i = 0; i < N/2; i++)    // count non 0s
     if(data[i]) count1++;         // in 1st half
}
#pragma omp task depend(in:data, out:count2)
{ /**    Count2Task */
   count2 = 0;
   for(int i = N/2+1; i < N; i++) // count non 0s
     if(data[i]) count2++;         // in 2nd half
}
#pragma omp task depend(in:count1, in:count2)
{ /**    SumTask    */
   numNonZeros = count1 + count2; // total non-zeros
   delete data;                    // free memory
}
```

(a)



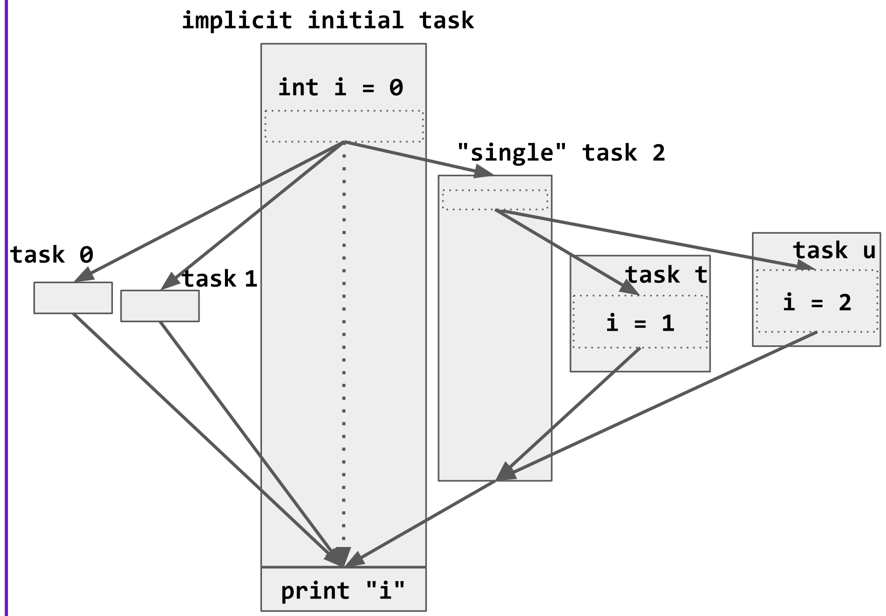
(b)

Non-Determinism

(a) Example code

```
int main() {
int i = 0;
#pragma omp parallel num_threads(2)
#pragma omp single
{
  #pragma omp task shared(i) // task t
  #pragma omp critical(lock_i)
  { i = 1; }

  #pragma omp task shared(i) // task u
  #pragma omp critical(lock_i)
  { i = 2; }
}
printf ("i=%d\n",i);
return 0;
}
```

(b) Task dependency



Even though updates on shared variable *i* is protected with a lock, this code results in non-deterministic execution, causing a race

OpenMP Tasks

- Do not try to use data parallel constructs to mimic task parallelism
  - Can create error-prone programs

```
#pragma omp parallel
{
   if(tid == 0)
        //do this
   else
        //do that
   #pragma omp barrier
}
```

If you have a work sharing construct inside else, then tid=0 will not reach that construct! Result is incorrect

```
else {
#pragma omp parallel for
 for … {
   …
 }
}
```

Your code will NOT work with single thread! Should work with any number of threads

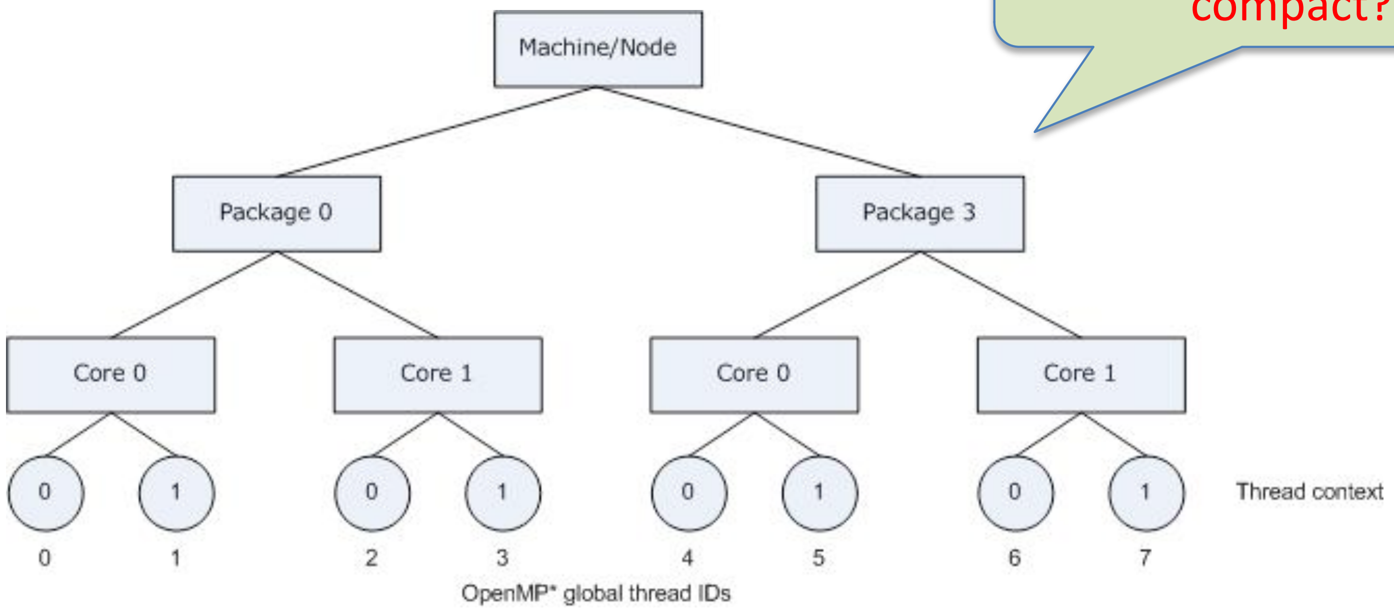
Thread Affinity

- The high-level affinity interface uses an environment variable to determine the machine topology and assigns OpenMP threads to the processors based upon their physical location in the machine.
- This interface is controlled entirely by KMP_AFFINITY on Intel compiler
  - Windows and Linux
- This interface provides compatibility with the gcc GOMP_AFFINITY environment variable
  - Linux Systems only

Compact

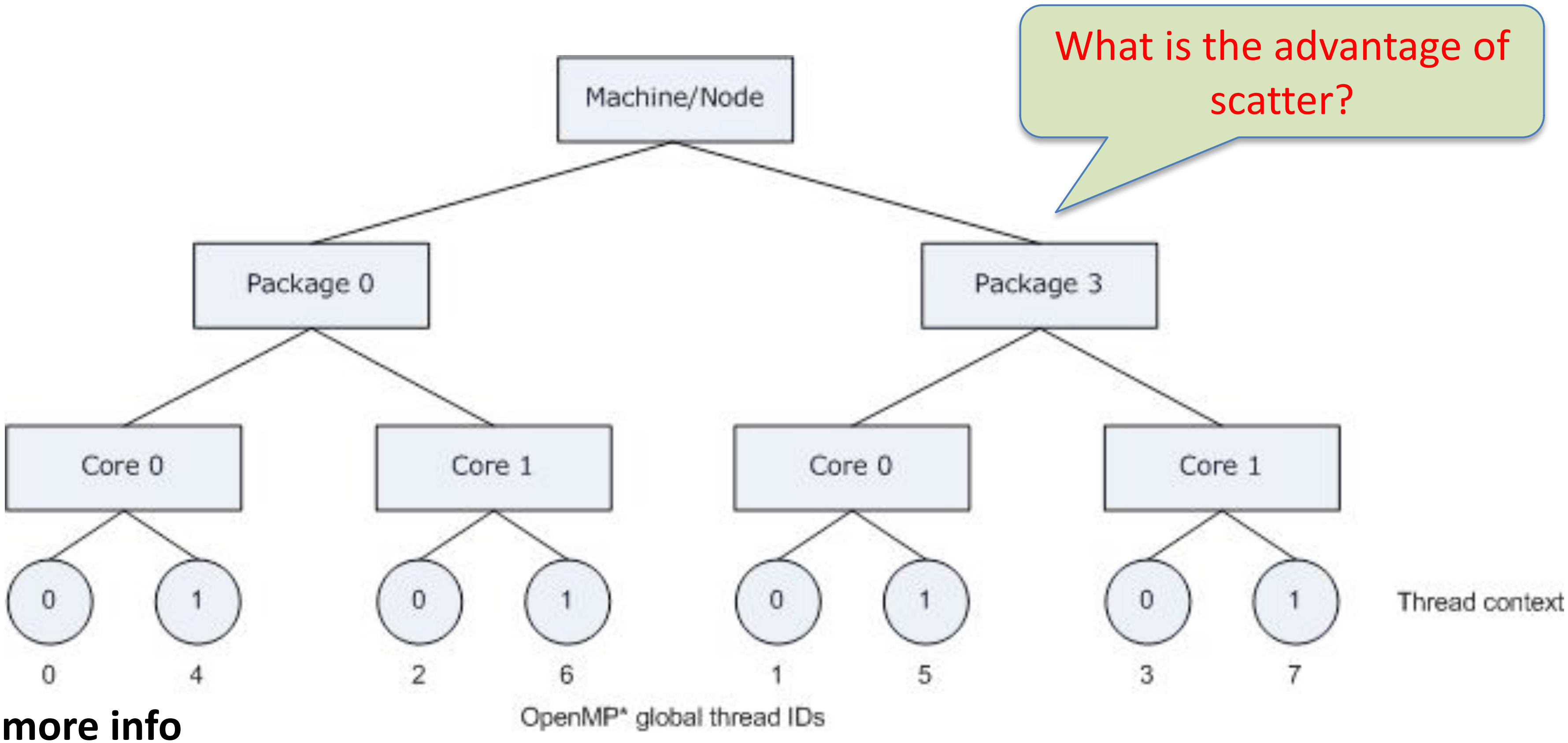
- The following figure illustrates the binding of OpenMP thread to hardware thread contexts when specifying `KMP_AFFINITY=granularity=fine,compact`

What is the advantage of compact?



Scatter

- Evenly distributes threads among cores in a round-robin fashion
- `KMP_AFFINITY=granularity=fine,scatter`



For more info

https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-thread-affinity-interface-linux-and-windows

Acknowledgments

- These slides are inspired and partly adapted from
  - Mary Hall (Univ. of Utah)
  - The course book (Pacheco)
  - Vivek Sarkar (Rice Univ.)
  - Cilk lecture by Charles Leiserson and Bradley Kuszmaul (Lecture 1, Scheduling Theory)