

---

# Parallel Programming

## An Introduction

Xu Liu

Derived from Prof. John Mellor-Crummey's COMP 422 from Rice University

---

---

**Applications need  
performance  
(speed)**

# The Need for Speed: Complex Problems

---

- **Science**
  - understanding matter from elementary particles to cosmology
  - storm forecasting and climate prediction
  - understanding biochemical processes of living organisms
- **Engineering**
  - combustion and engine design
  - computational fluid dynamics and airplane design
  - earthquake and structural modeling
  - pollution modeling and remediation planning
  - molecular nanotechnology
- **Business**
  - computational finance - high frequency trading
  - information retrieval
  - data mining
- **Defense**
  - nuclear weapons stewardship
  - cryptology

# Earthquake Simulation

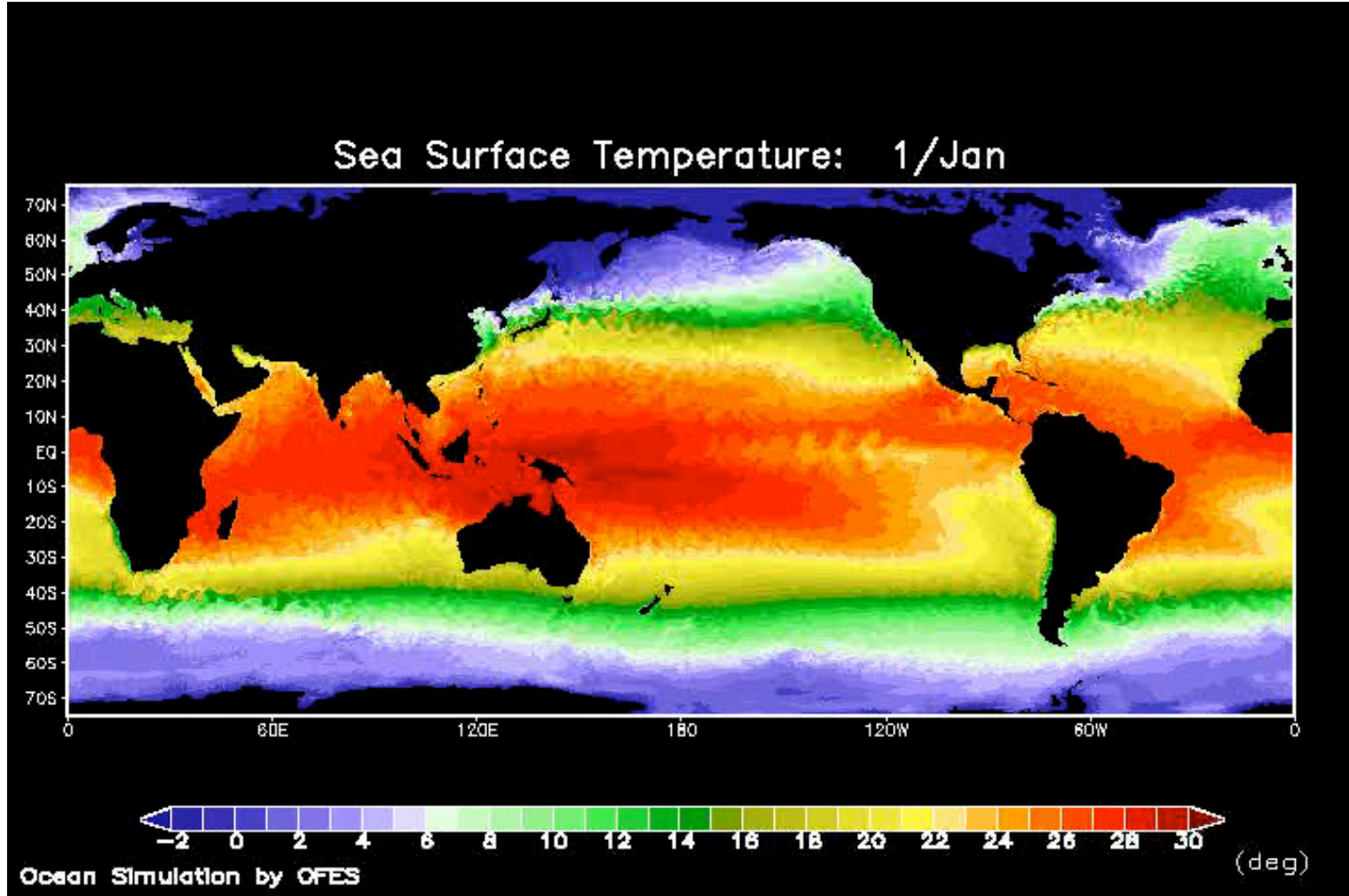


**Earthquake Research Institute, University of Tokyo**

**Tonankai-Tokai Earthquake Scenario**

**Photo Credit: The Earth Simulator Art Gallery, CD-ROM, March 2004**

# Ocean Circulation Simulation



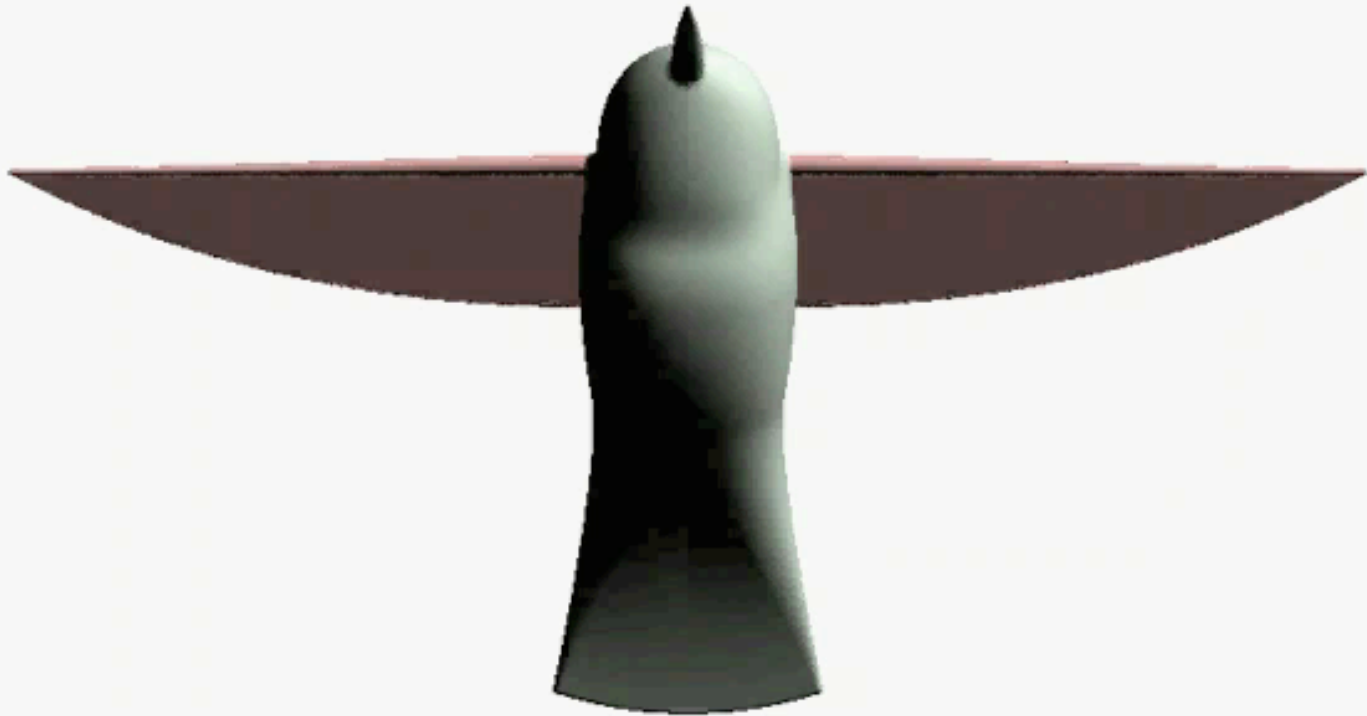
**Ocean Global Circulation Model for the Earth Simulator**

**Seasonal Variation of Ocean Temperature**

**Photo Credit: The Earth Simulator Art Gallery, CD-ROM, March 2004**

# Air Velocity (Front)

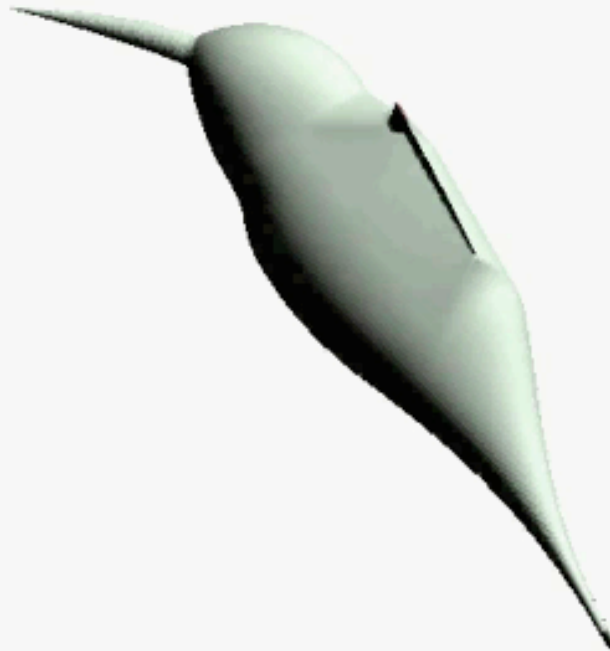
---



Andrew A. Johnson  
Army HPC Research Center

# Air Velocity (Side)

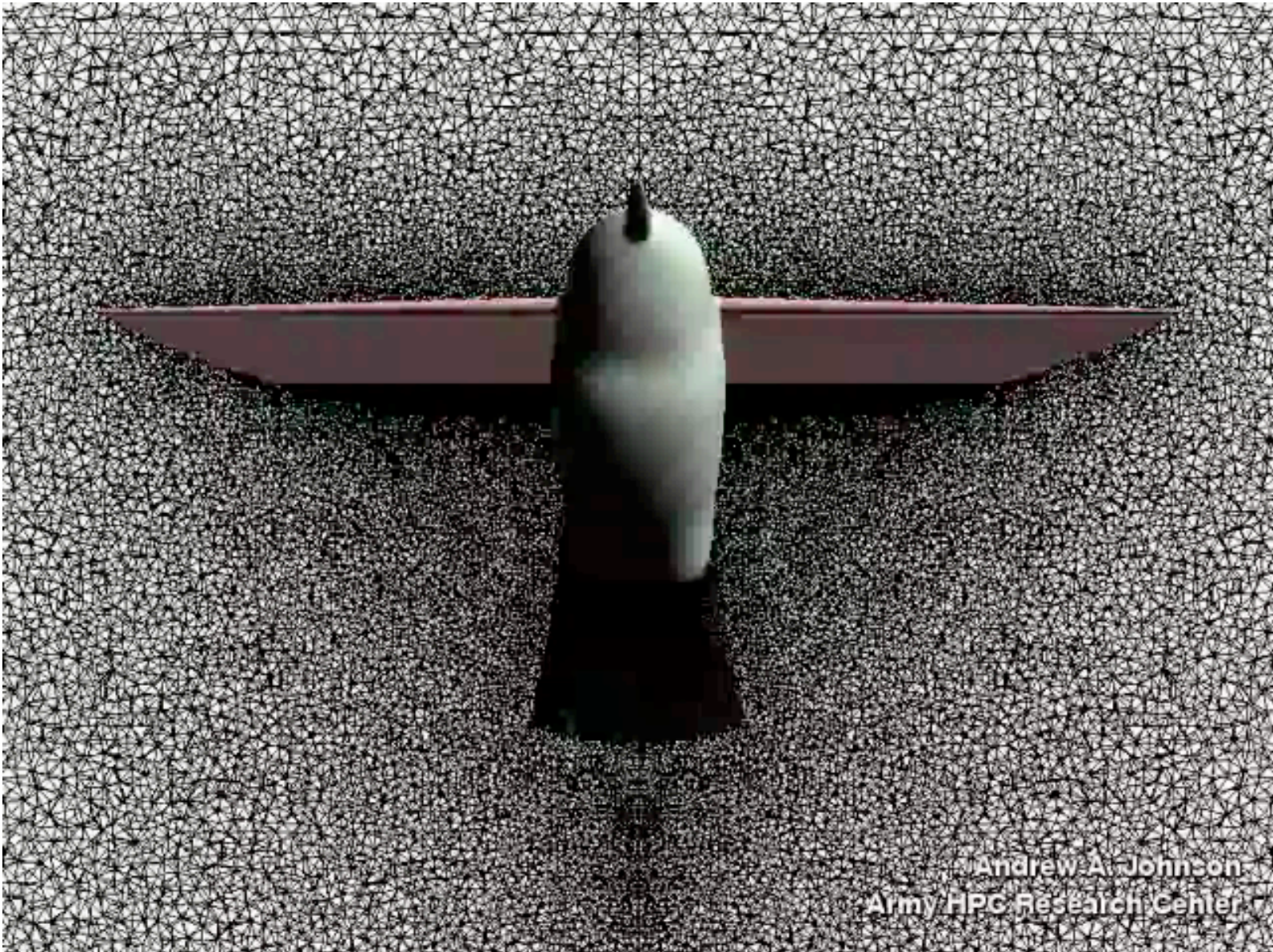
---



Andrew A. Johnson  
Army HPC Research Center

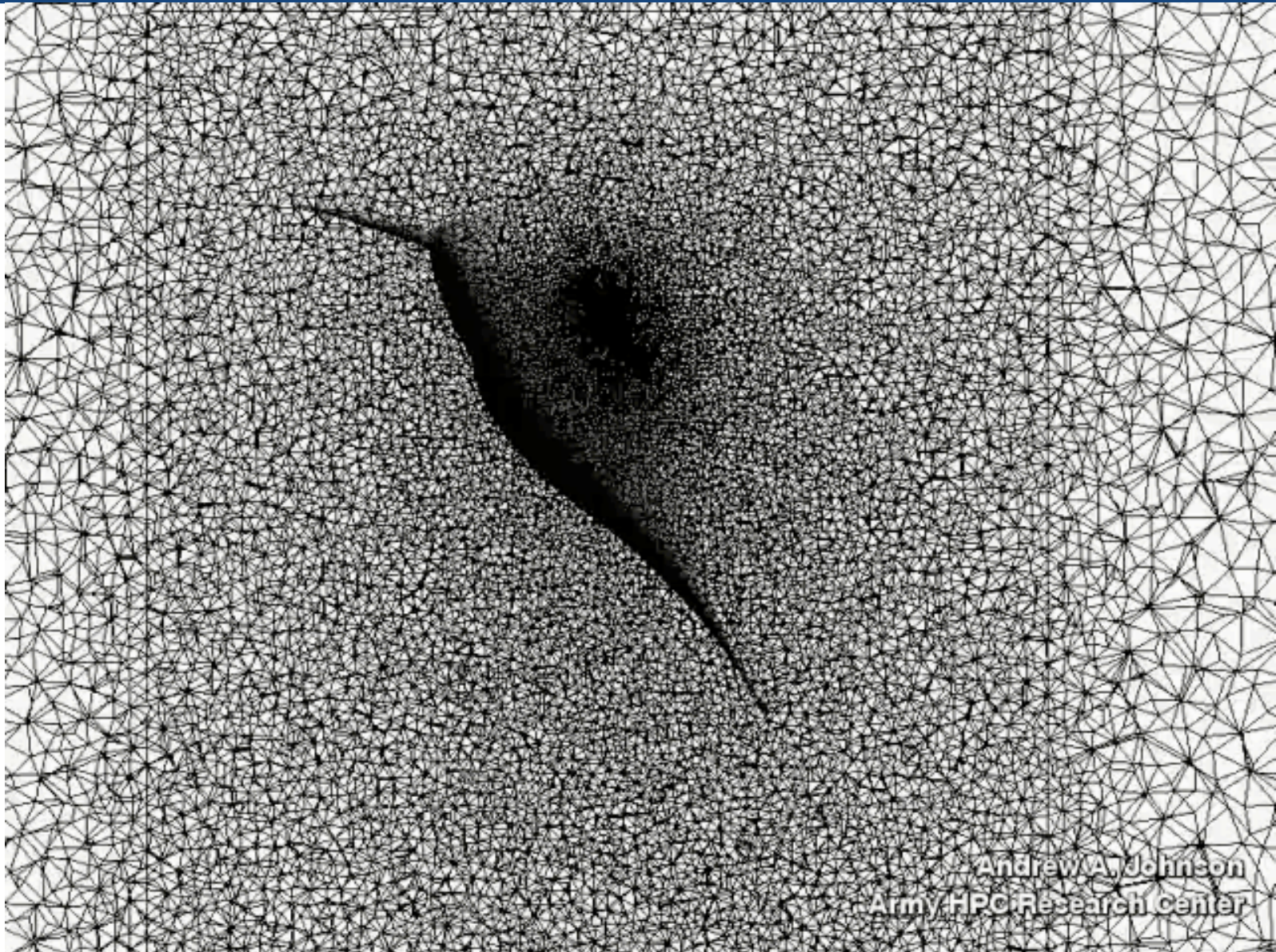


# Mesh Adaptation (front)





# Mesh Adaptation (side)



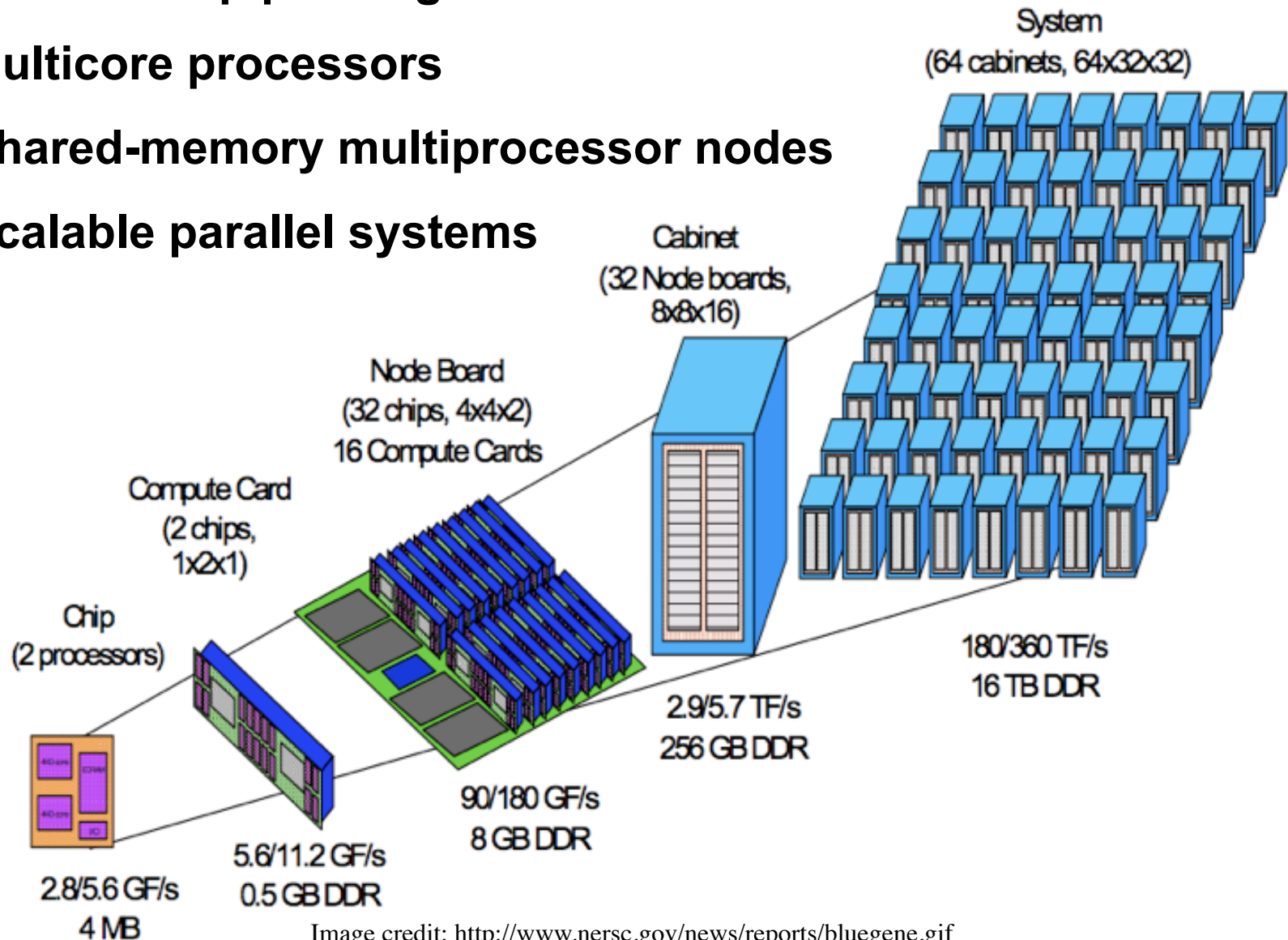
---

# **Parallel Hardware in the Large**



# Hierarchical Parallelism in Supercomputers

- Cores with pipelining and short vectors
- Multicore processors
- Shared-memory multiprocessor nodes
- Scalable parallel systems



# Blue Gene/Q Packaging Hierarchy

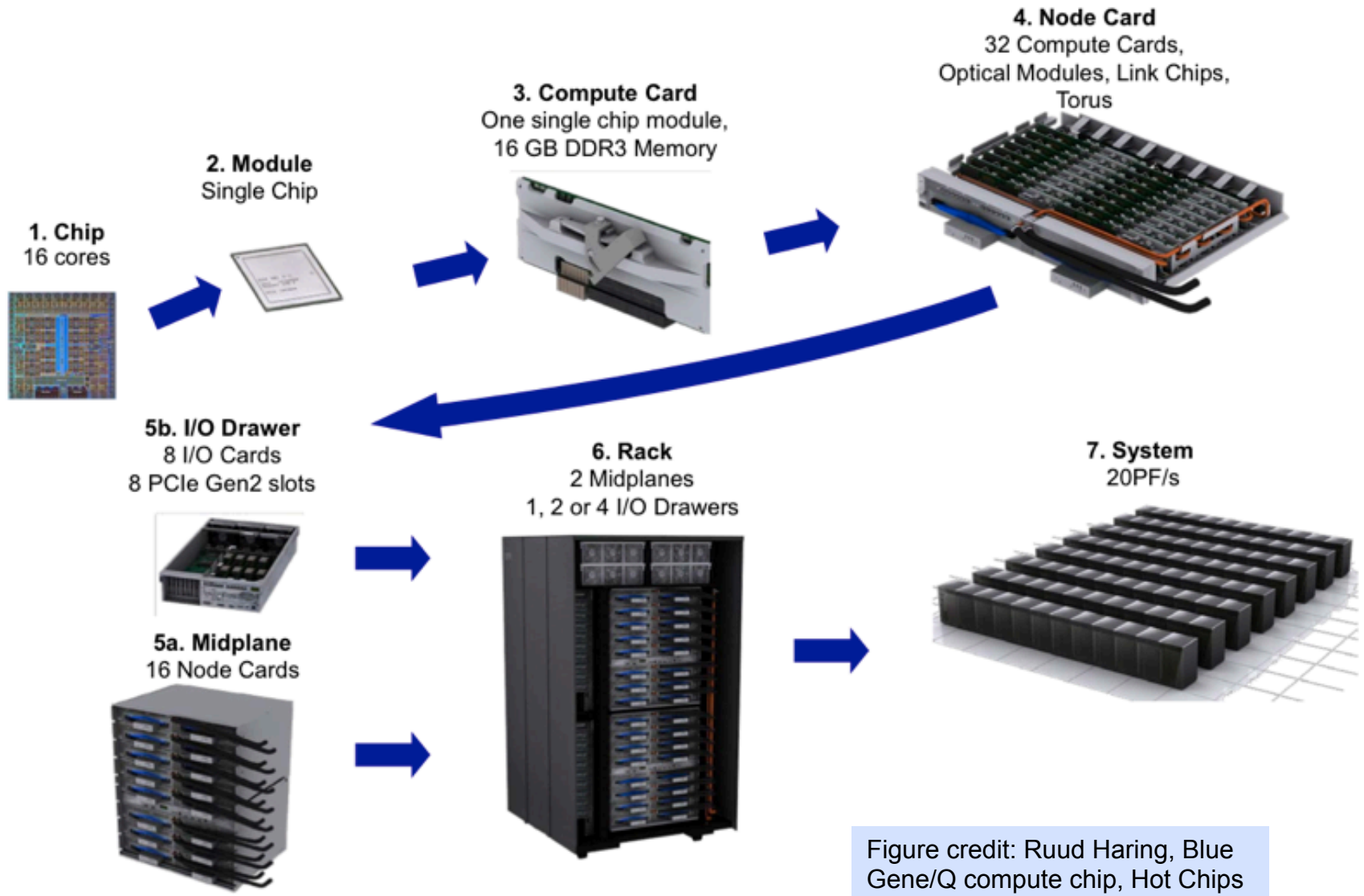


Figure credit: Ruud Haring, Blue Gene/Q compute chip, Hot Chips 23, August, 2011.

# Scale of the Largest HPC Systems (Nov 2013)

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Super Computer Center in Guangzhou China	<b>Tianhe-2 (MilkyWay-2)</b> - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	<b>Titan</b> - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	<b>Sequoia</b> - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	<b>K computer</b> , SPARC64 Villifx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	<b>Mira</b> - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
6	Swiss National Supercomputing Centre (CSCS) Switzerland	<b>Piz Daint</b> - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
7	Texas Advanced Computing Center/Univ. of Texas United States	<b>Stampede</b> - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P Dell	462,462	5,168.1	8,520.1	4,510
8	Forschungszentrum Juelich (FZJ) Germany	<b>JUQUEEN</b> - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	458,752	5,008.9	5,872.0	2,301
9	DOE/NNSA/LLNL United States	<b>Vulcan</b> - BlueGene/Q, Power BQC 16C 1.600GHz, Custom Interconnect IBM	393,216	4,293.3	5,033.2	1,972
10	Leibniz Rechenzentrum Germany	<b>SuperMUC</b> - iDataPlex DX360M4, Xeon E5-2680 8C 2.70GHz, Infiniband FDR	147,456	2,897.0	3,185.1	3,423

> 1.5M  
cores

all  
> 100K  
cores

hybrid  
CPU+GPU



# Top Petascale Systems

**(PetaFLOPS =  $10^{15}$  Floating-point Operations Per Second)**

- **China: NUDT Tianhe-1a**
  - hybrid architecture
    - 14,336 6-core Intel Westmere processors
    - 7,168 NVIDIA Tesla M2050M GPU
  - proprietary interconnect
  - peak performance ~4.7 petaflop
- **ORNL Jaguar system**
  - 6-core 2.6GHz AMD Opteron processors
  - over 224K processor cores
  - toroidal interconnect topology: Cray Seastar2+
  - peak performance ~2.3 petaflop
  - upgraded 2009



# Challenges of Parallelism in the Large

---

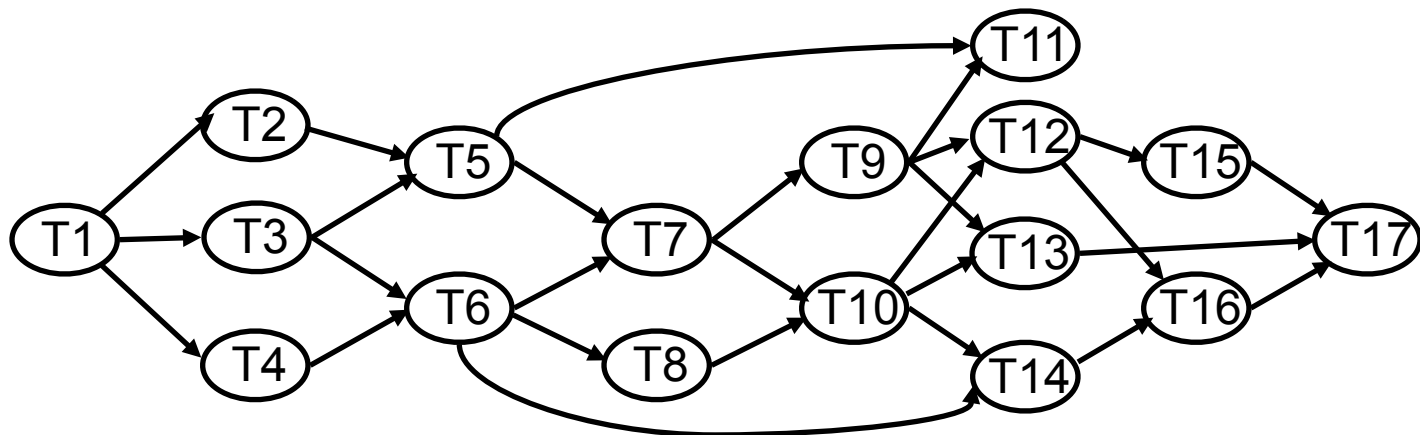
- **Parallel science applications are often very sophisticated**
  - e.g. adaptive algorithms may require dynamic load balancing
- **Multilevel parallelism is difficult to manage**
- **Extreme scale exacerbates inefficiencies**
  - algorithmic scalability losses
  - serialization and load imbalance
  - communication or I/O bottlenecks
  - insufficient or inefficient parallelization
- **Hard to achieve top performance even on individual nodes**
  - contention for shared memory bandwidth
  - memory hierarchy utilization on multicore processors

---

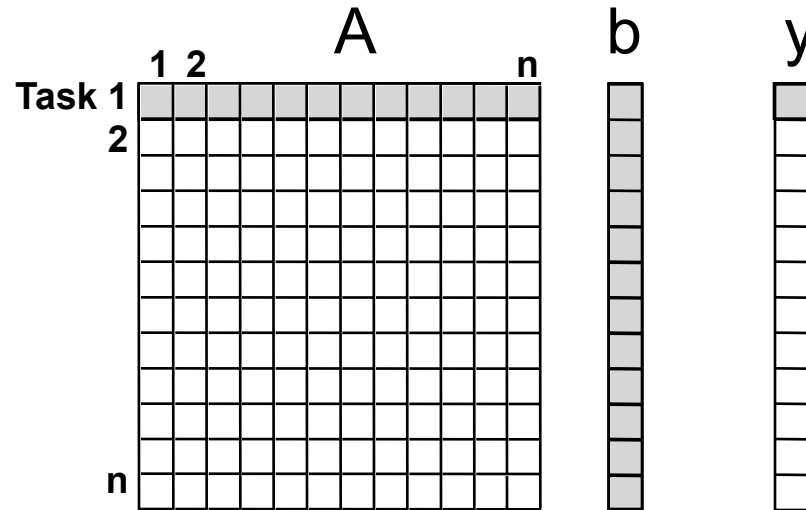
# **Parallel Programming Concept**

# 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 *task dependency DAG*
  - node = task
  - edge = control dependence



# Example: Dense Matrix-Vector Multiplication

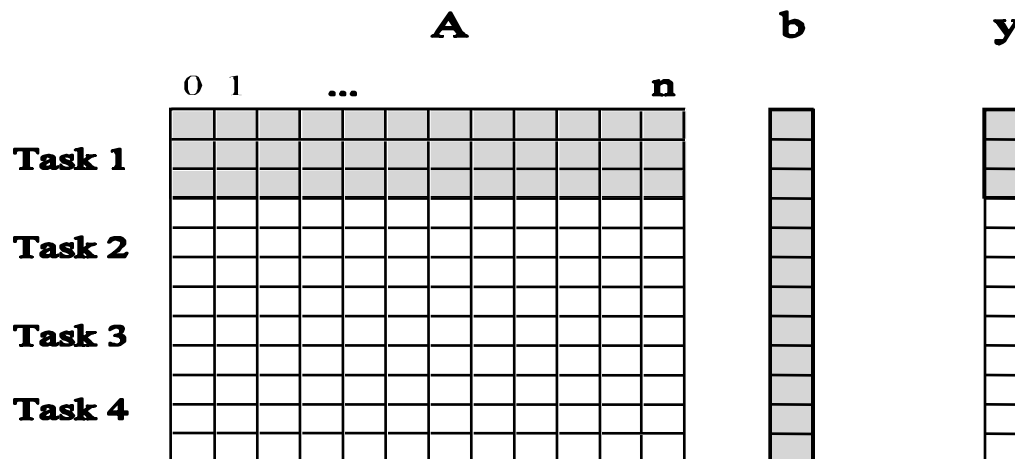


- Computing each element of output vector  $y$  is independent
- Easy to decompose dense matrix-vector product into tasks
  - one per element in  $y$
- Observations
  - task size is uniform
  - no control dependences between tasks
  - tasks share  $b$



# Granularity of Task Decompositions

- **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  $y$

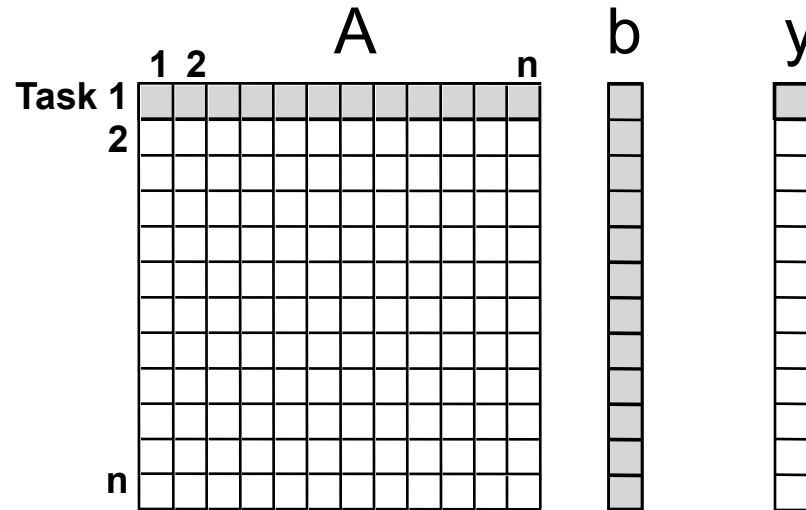


# Degree of Concurrency

---

- **Definition:** number of tasks that can execute in parallel
- **May change during program execution**
- **Metrics**
  - *maximum degree of concurrency*
    - *largest # concurrent tasks at any point in the execution*
  - *average degree of concurrency*
    - *average number of tasks that can be processed in parallel*
- **Degree of concurrency vs. task granularity**
  - *inverse relationship*

# Example: Dense Matrix-Vector Multiplication



- 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  $b$

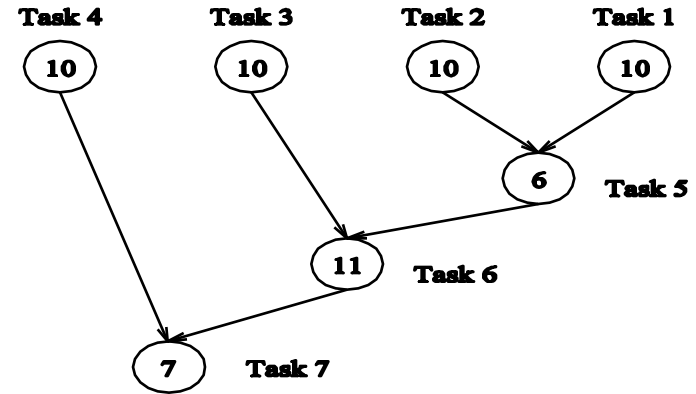
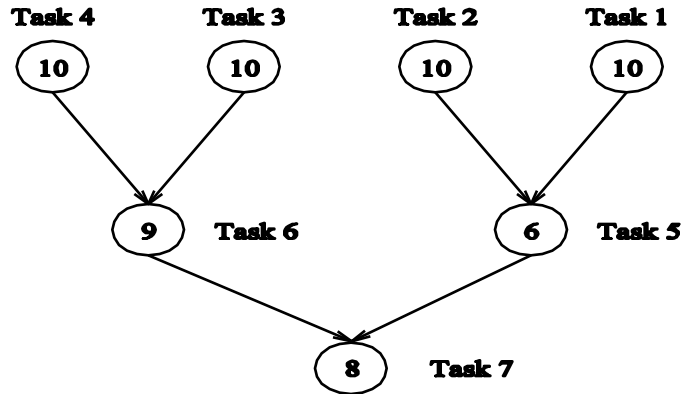
**Question: Is  $n$  the maximum number of tasks possible?**

# Critical Path

---

- **Edge in task dependency graph represents task serialization**
- **Critical path = longest weighted path through graph**
- **Critical path length = lower bound on parallel execution time**

# Critical Path Length



## Questions:

What are the tasks on the critical path for each dependency graph?

What is the shortest parallel execution time for each decomposition?

How many processors are needed to achieve the minimum time?

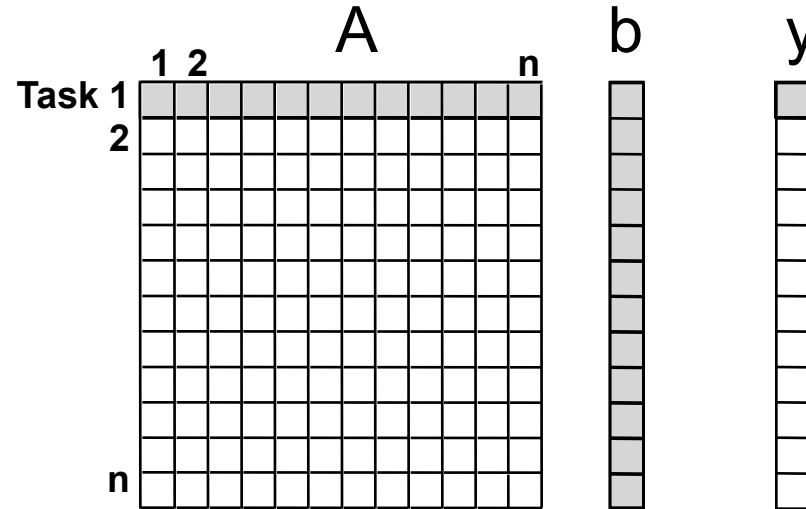
What is the maximum degree of concurrency?

What is the average parallelism?



# Critical Path Length

Example: dependency graph for dense-matrix vector product



## Questions:

What does a task dependency graph look like for DMVP?

What is the shortest parallel execution time for the graph?

How many processors are needed to achieve the minimum time?

What is the maximum degree of concurrency?

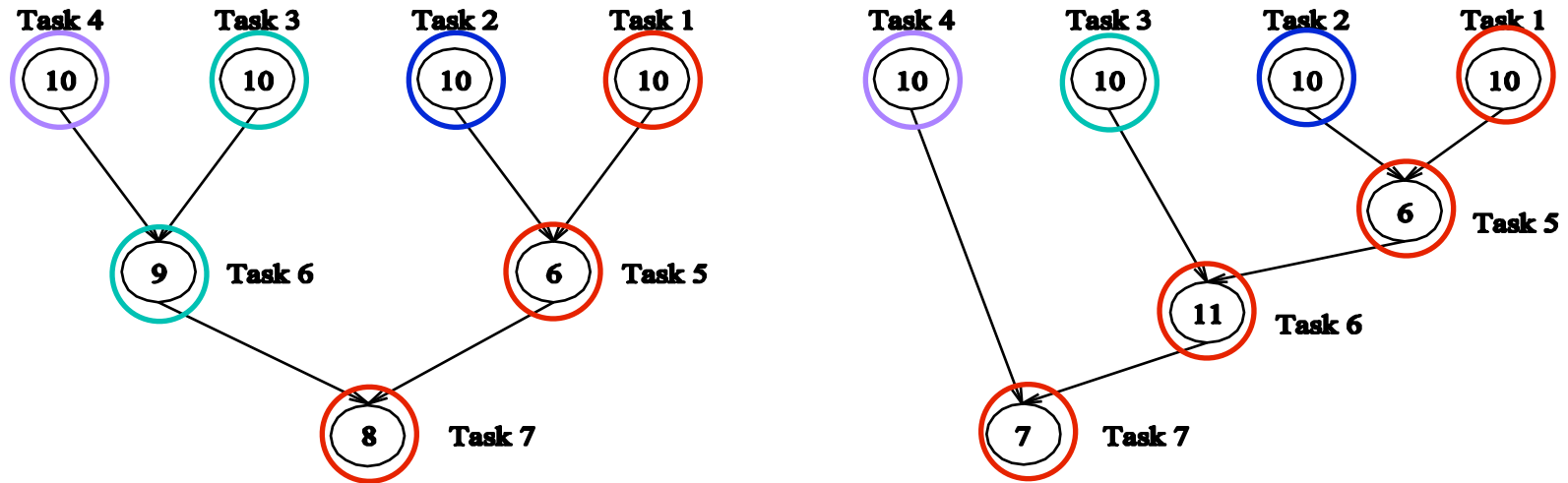
What is the average parallelism?

# Limits on Parallel Performance

---

- What bounds parallel execution time?
  - minimum task granularity
    - *e.g. dense matrix-vector multiplication  $\leq n^2$  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*
- Measures of parallel performance
  - speedup =  $T_1/T_p$
  - parallel efficiency =  $T_1/(pT_p)$

# Processes and Mapping Example



- Consider the dependency graphs in levels
  - no nodes in a level depend upon one another
  - compute levels using topological sort
- Assign all tasks within a level to different processes

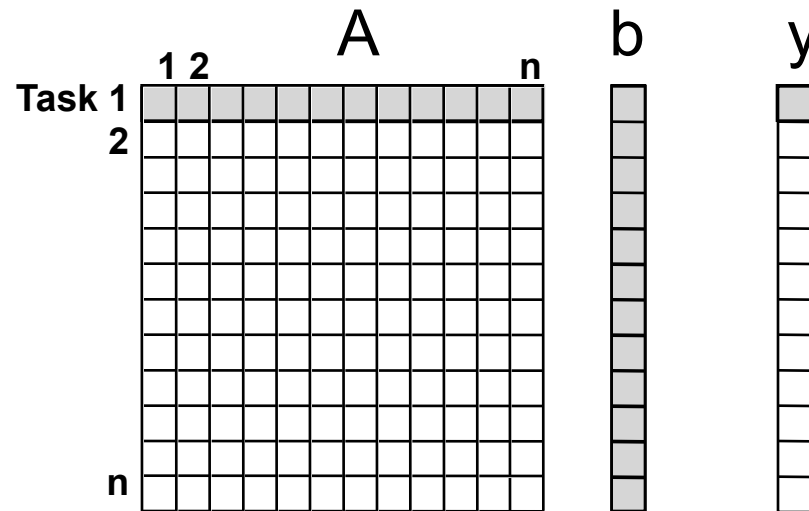
---

# **Task Decomposition**

# Decomposition Based on Output Data

- 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**





# Output Data Decomposition: Example

---

- **Matrix multiplication:  $C = A \times B$**
- **Computation of  $C$  can be partitioned into four tasks**

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 1:  $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

Task 2:  $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

Task 3:  $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

Task 4:  $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

Other task decompositions possible

# Exploratory Decomposition


---

- **Exploration (search) of a state space of solutions**
  - problem decomposition reflects shape of execution
- **Examples**
  - theorem proving
  - game playing


# Exploratory Decomposition Example

## Solving a 15 puzzle


- Sequence of three moves from state (a) to final state (d)

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

(a)

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

(b)

1	2	3	4
5	6	7	8
9	10	11	
13	14	15	12

(c)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(d)

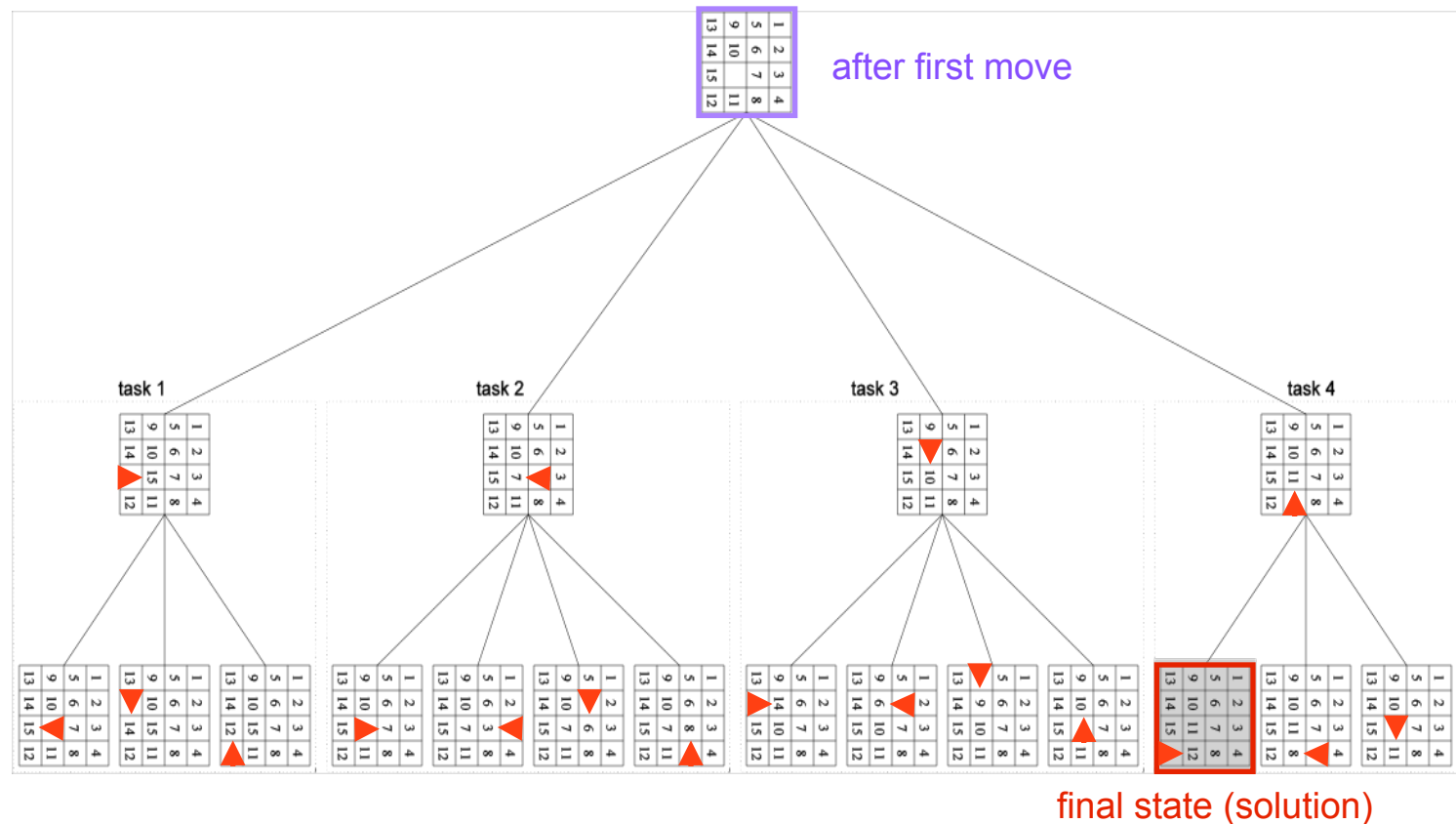
- From an arbitrary state, must search for a solution

# Exploratory Decomposition: Example

## Solving a 15 puzzle

### Search

- generate successor states of the current state
- explore each as an independent task



---

# Task Mapping

# Mapping Techniques

---

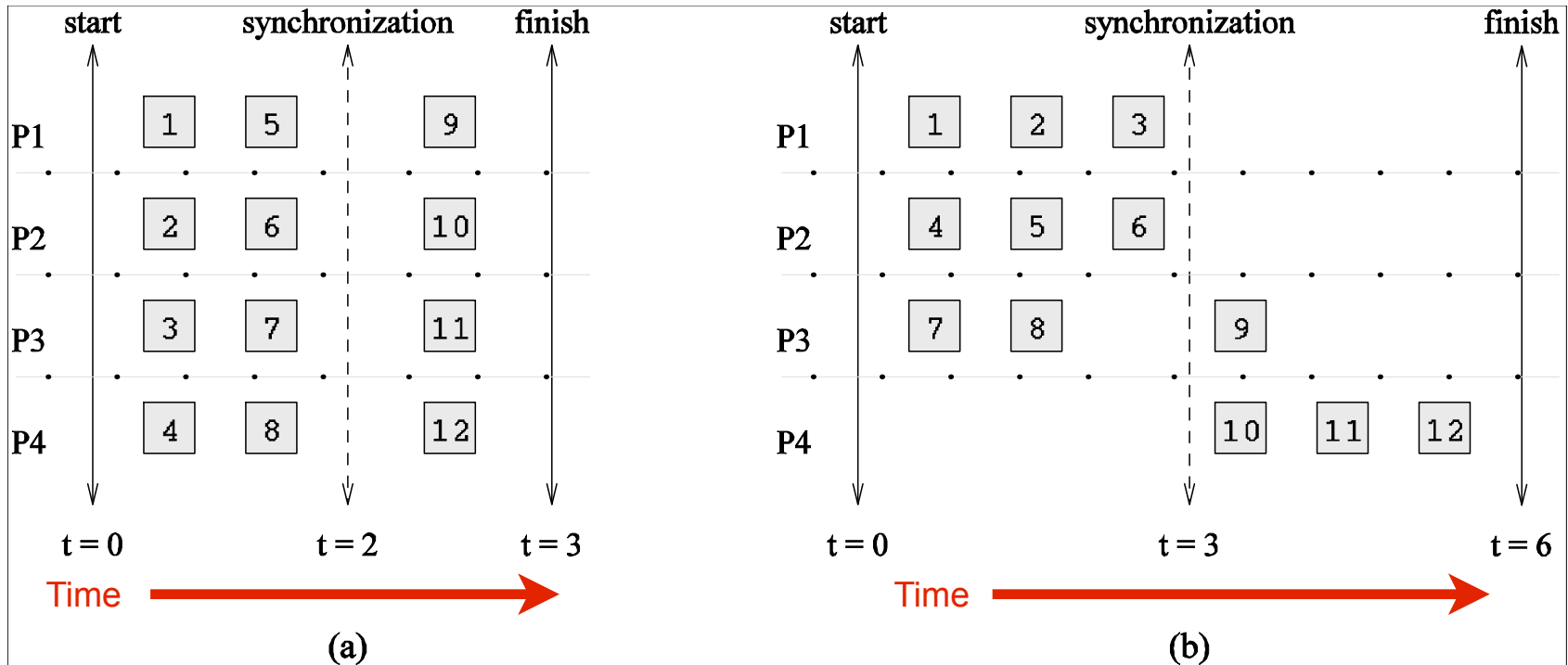
**Map concurrent tasks to processes for execution**

- **Overheads of mappings**
  - serialization (idling)
  - communication
- **Select mapping to minimize overheads**
- **Conflicting objectives: minimizing one increases the other**
  - assigning all work to one processor
    - minimizes communication
    - significant idling
  - minimizing serialization introduces communication



# Mapping Techniques for Minimum Idling

- Must simultaneously minimize idling and load balance
- Balancing load alone does not minimize idling



# Mapping Techniques for Minimum Idling

---

## Static vs. dynamic mappings

- **Static mapping**
  - *a-priori* mapping of tasks to processes
  - requirements
    - a good estimate of task size
- **Dynamic mapping**
  - map tasks to processes at runtime
  - why?
    - tasks are generated at runtime, or
    - their sizes are unknown

### Factors that influence choice of mapping

- size of data associated with a task
- nature of underlying domain

# Schemes for Static Mapping

---

- **Data partitionings**
- **Task graph partitionings**
- **Hybrid strategies**

# Mappings Based on Data Partitioning

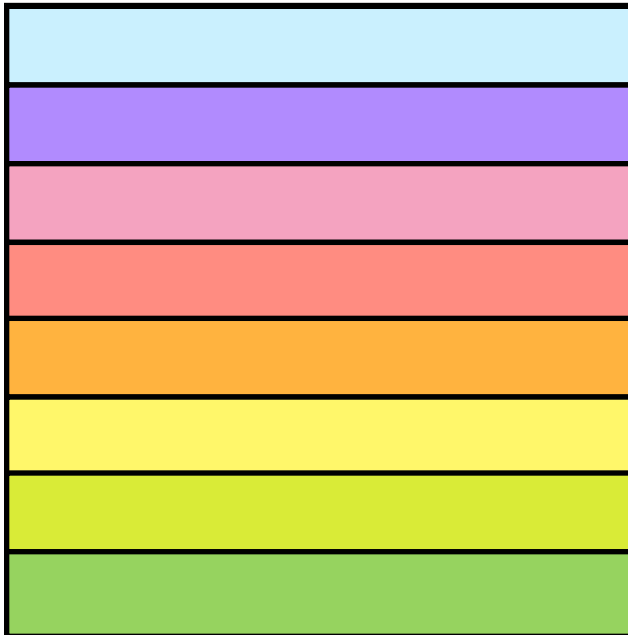
---

Partition computation using a combination of

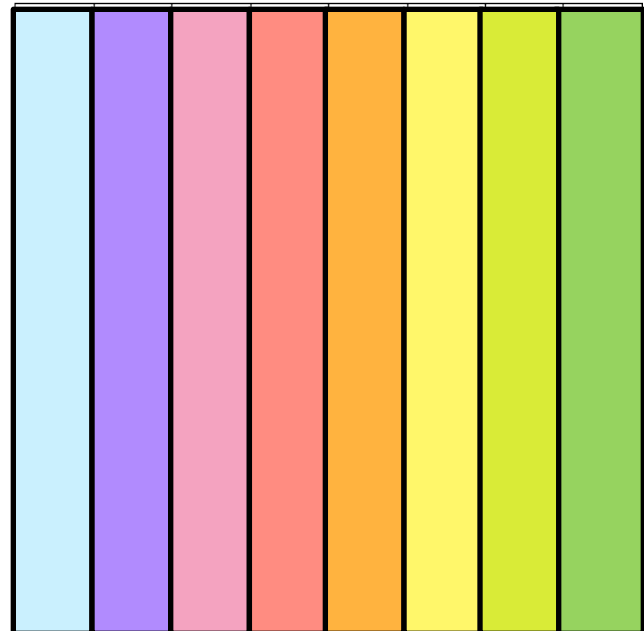
- data partitioning
- owner-computes* rule

**Example: 1-D block distribution for dense matrices**

row-wise distribution



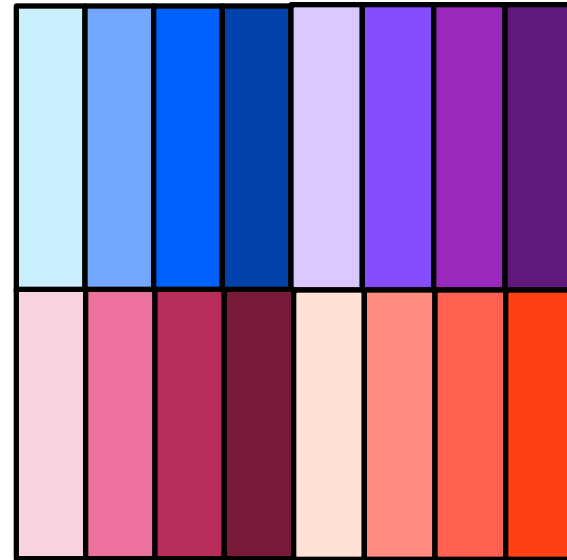
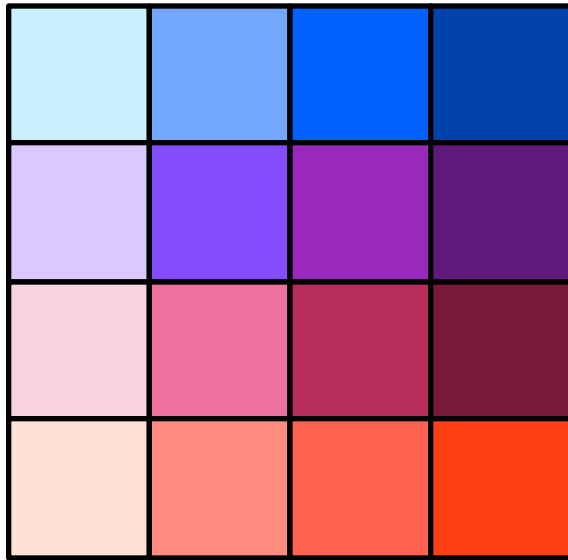
column-wise distribution



# Block Array Distribution Schemes

---

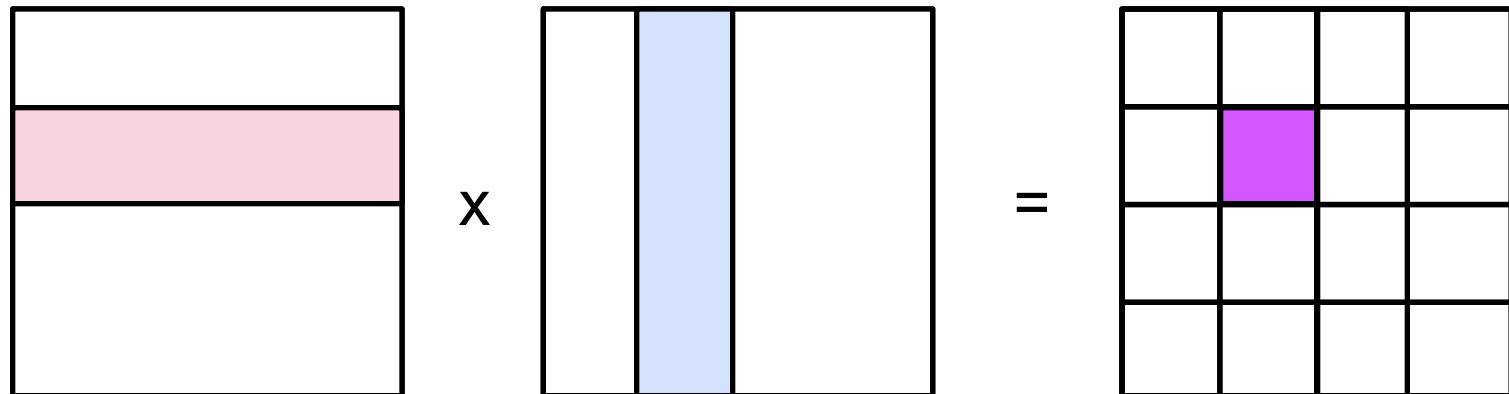
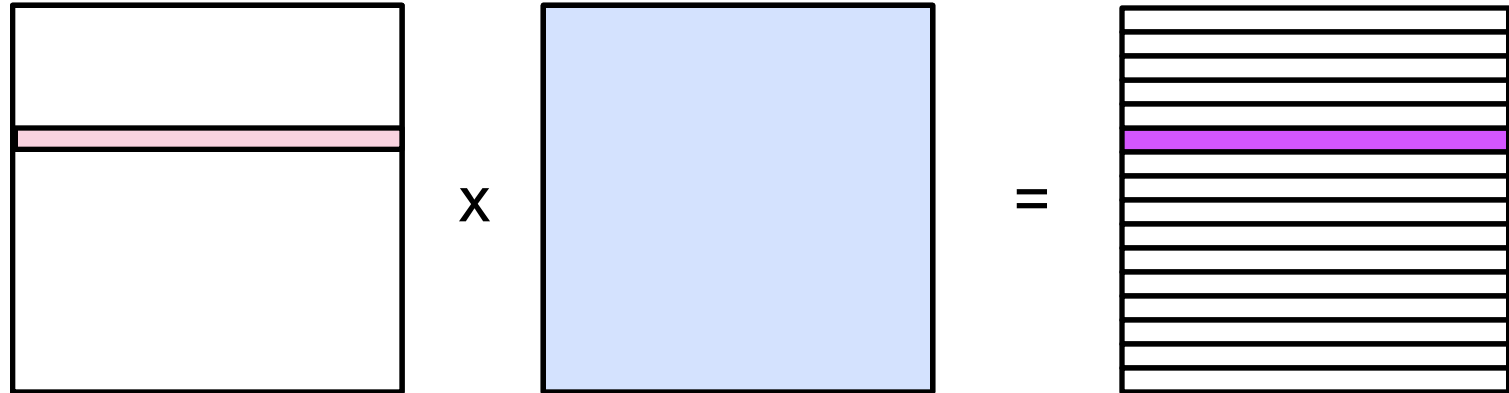
## Multi-dimensional block distributions



**Multi-dimensional partitioning enables larger # of processes**

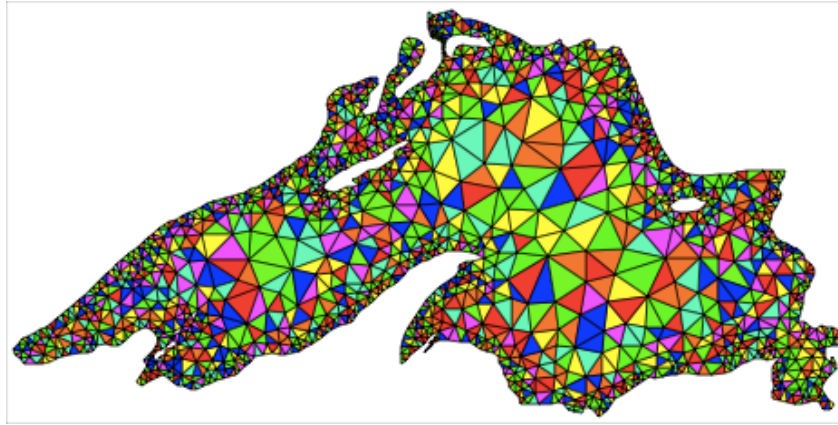
# Data Usage in Dense Matrix Multiplication

Multiplying two dense matrices  $C = A \times B$

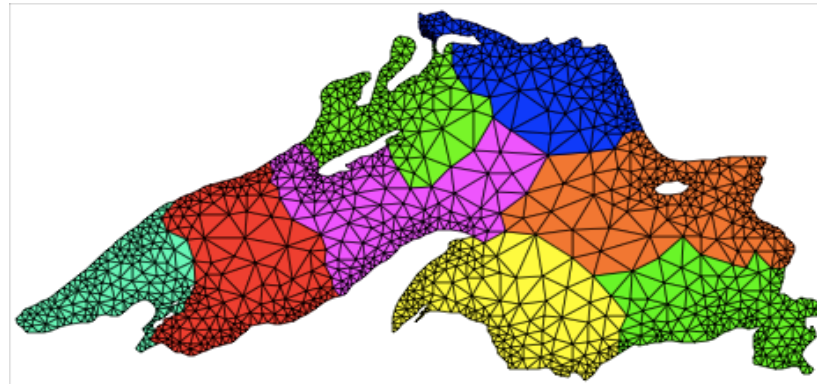


# Partitioning a Graph of Lake Superior

---



**Random Partitioning**

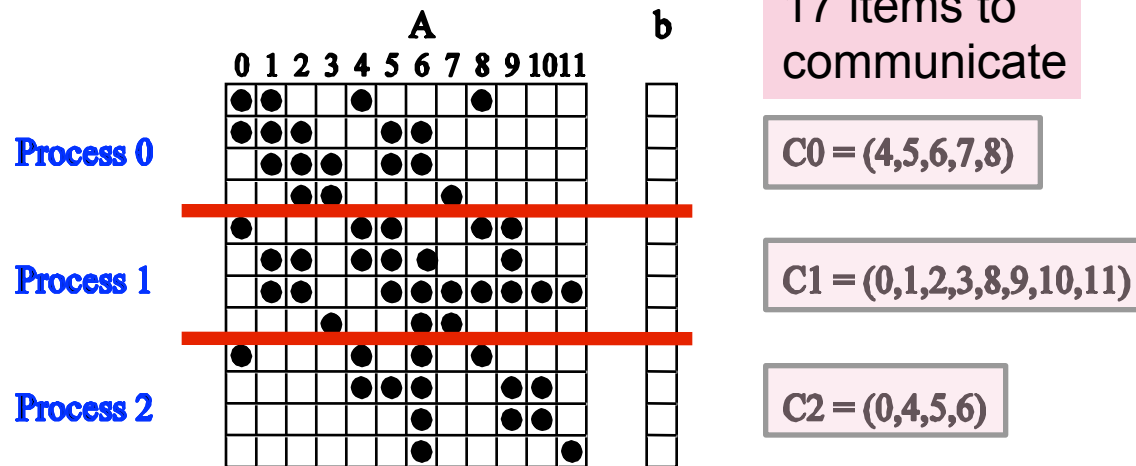


**Partitioning for minimum edge-cut**



# Mapping a Sparse Matrix

## Sparse matrix-vector product

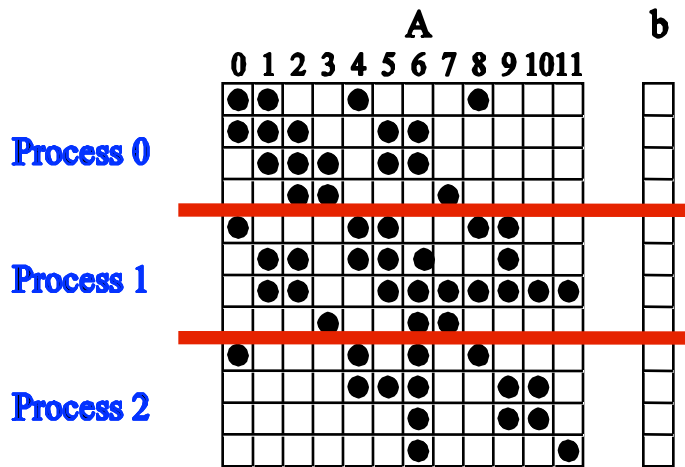


sparse matrix structure

mapping  
partitioning

# Mapping a Sparse Matrix

## Sparse matrix-vector product



sparse matrix structure

mapping  
partitioning

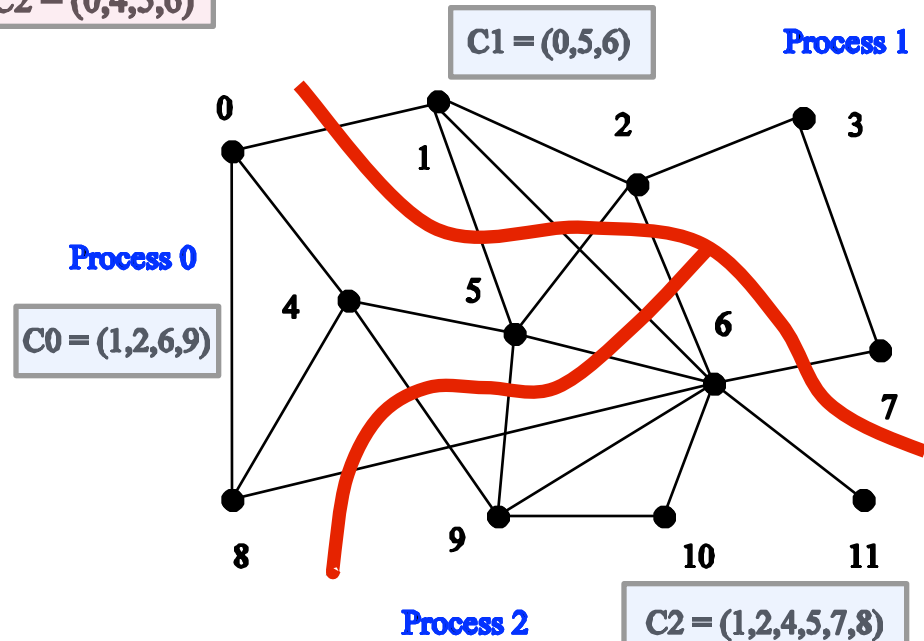
17 items to  
communicate

$C0 = (4,5,6,7,8)$

$C1 = (0,1,2,3,8,9,10,11)$

$C2 = (0,4,5,6)$

13 items to  
communicate



# Schemes for Dynamic Mapping

---

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

# Centralized Dynamic Mapping

---

- **Processes = masters or slaves**
- **General strategy**
  - when a slave runs out of work → request more from master
- **Challenge**
  - master may become bottleneck for large # of processes
- **Approach**
  - chunk scheduling: process picks up several of tasks at once
  - however
    - large chunk sizes may cause significant load imbalances
    - gradually decrease chunk size as the computation progresses

# Distributed Dynamic Mapping

---

- All processes as peers
- Each process can send or receive work from other processes
  - avoids centralized bottleneck
- Four critical design questions
  - how are sending and receiving processes paired together?
  - who initiates work transfer?
  - how much work is transferred?
  - when is a transfer triggered?
- Ideal answers can be application specific
- Cilk uses a distributed dynamic mapping: “work stealing”

# Minimizing Interaction Overheads (1)

---

## “Rules of thumb”

- **Minimize volume of data exchange**
  - partition interaction graph to minimize edge crossings
- **Minimize frequency of communication**
  - try to aggregate messages where possible
- **Minimize contention and hot-spots**
  - use decentralized techniques (avoidance)

# Minimizing Interaction Overheads (2)

---

## Techniques

- **Overlap communication with computation**
  - **use non-blocking communication primitives**
    - overlap communication with your own computation
    - one-sided: prefetch remote data to hide latency
  - **multithread code on a processor**
    - overlap communication with another thread's computation
- **Replicate data or computation to reduce communication**
- **Use group communication instead of point-to-point primitives**
- **Issue multiple communications and overlap their latency**  
(reduces exposed latency)