Introduction to Parallel Computing: Models, Concepts, and Algorithms (Part II)

SC3260/5260 High-Performance Computing

Hongyang Sun
(hongyang.sun@vanderbilt.edu)

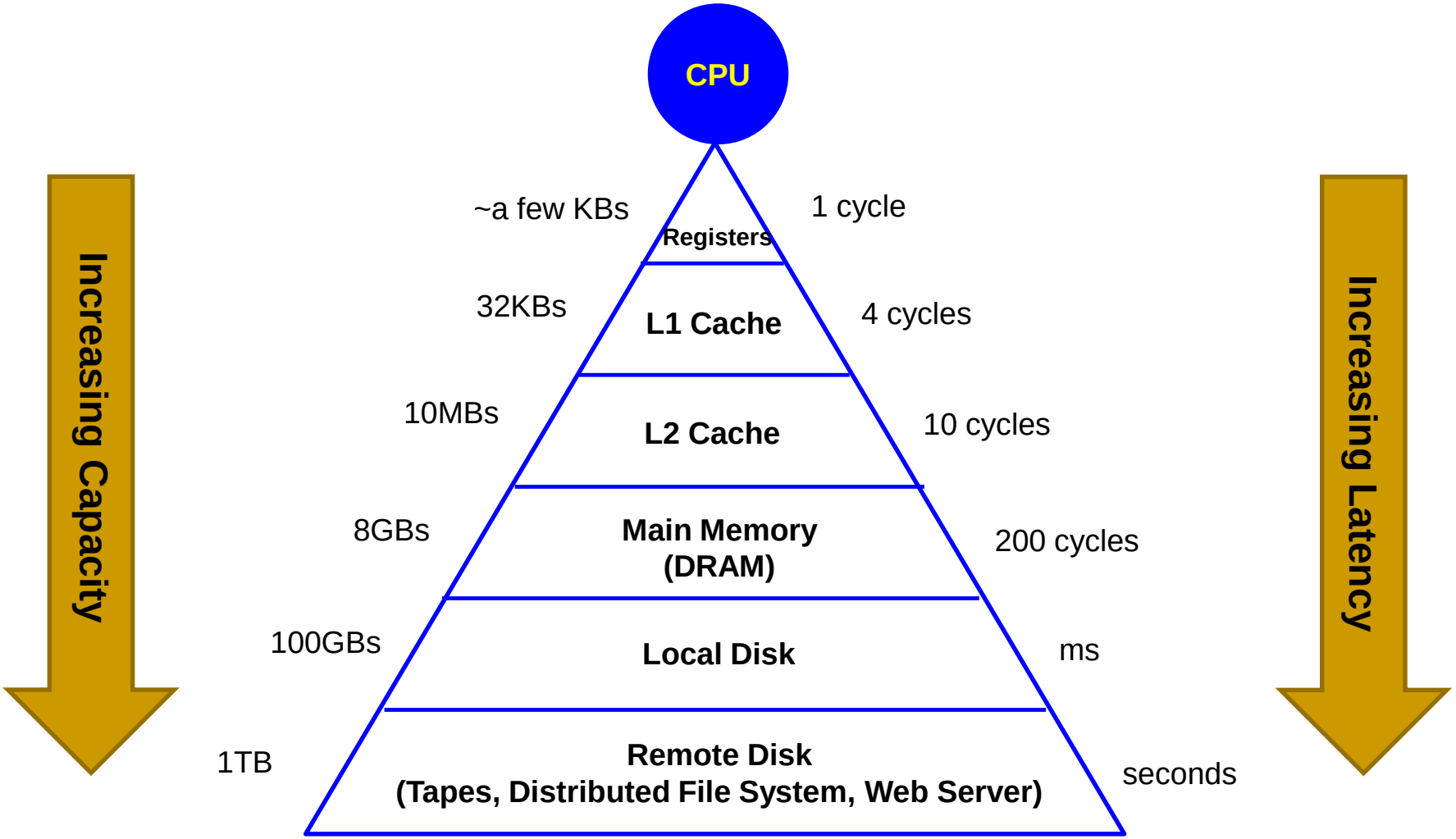
Vanderbilt University

Spring 2020

Improving Software Performance

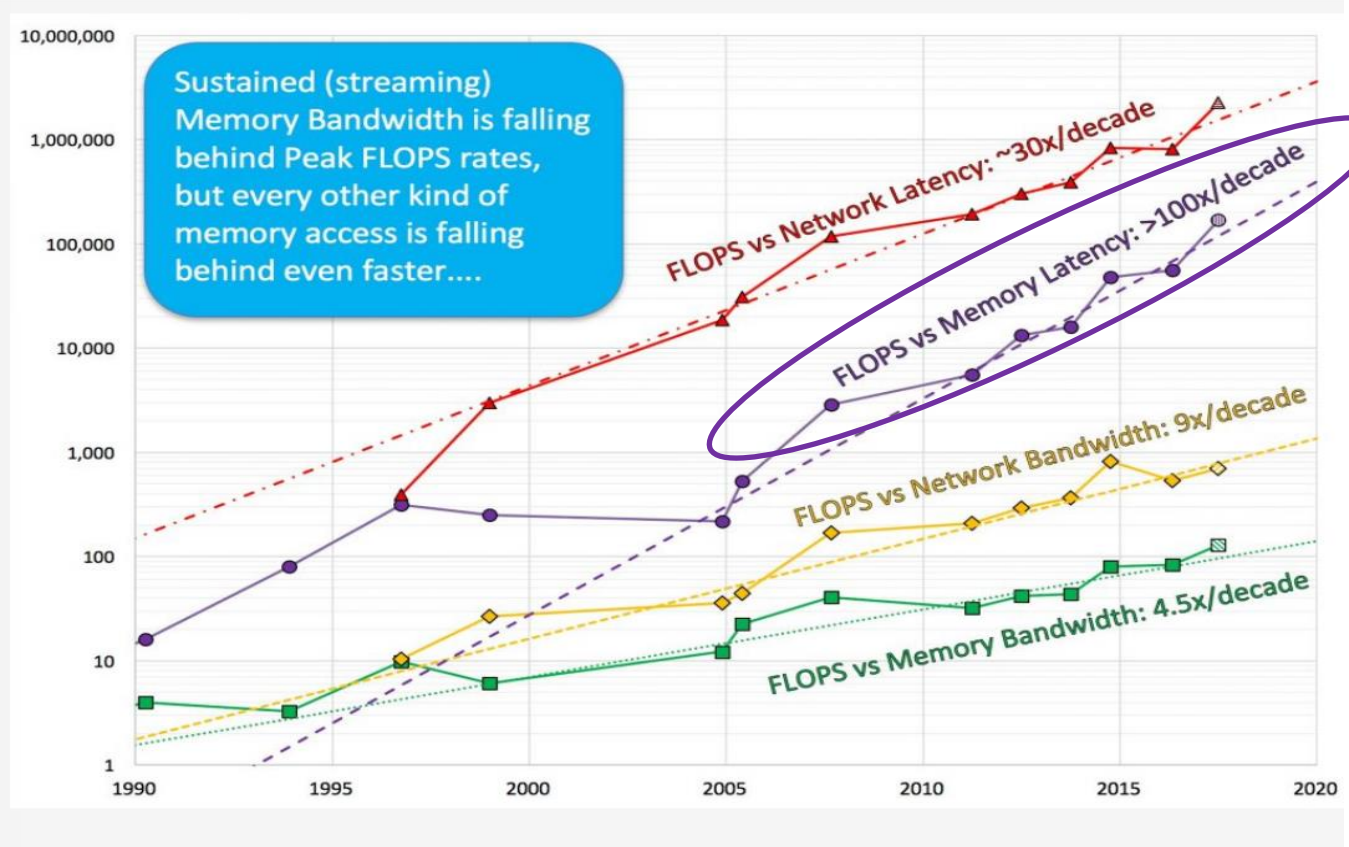
- **Serial Optimization**
  - Programmer should optimize serial code before attempting parallelization.
- **Implicit Parallelism**
  - Hardware and compiler technology implicitly convert part of serial programs into parallel and execute them on multiple processing units of a processor (e.g., pipeline, superscalar, instruction-level parallelism)
  - No effort from programmer, but limited parallelism & performance gains
- **Explore Memory System**
  - Architectural features (e.g., cache, threads) to improve data-movement cost
  - Exploitation of locality (spatial and temporal) by hardware and programmer
- **Explicit Parallelism**
  - Programmer's responsibility to identify explicit parallelism and write parallel code with help of parallel programming language and libraries to be run on multicore or distributed processors.
  - No simple "recipes", but some design guidelines could help.

Memory Hierarchy



Memory System Performance

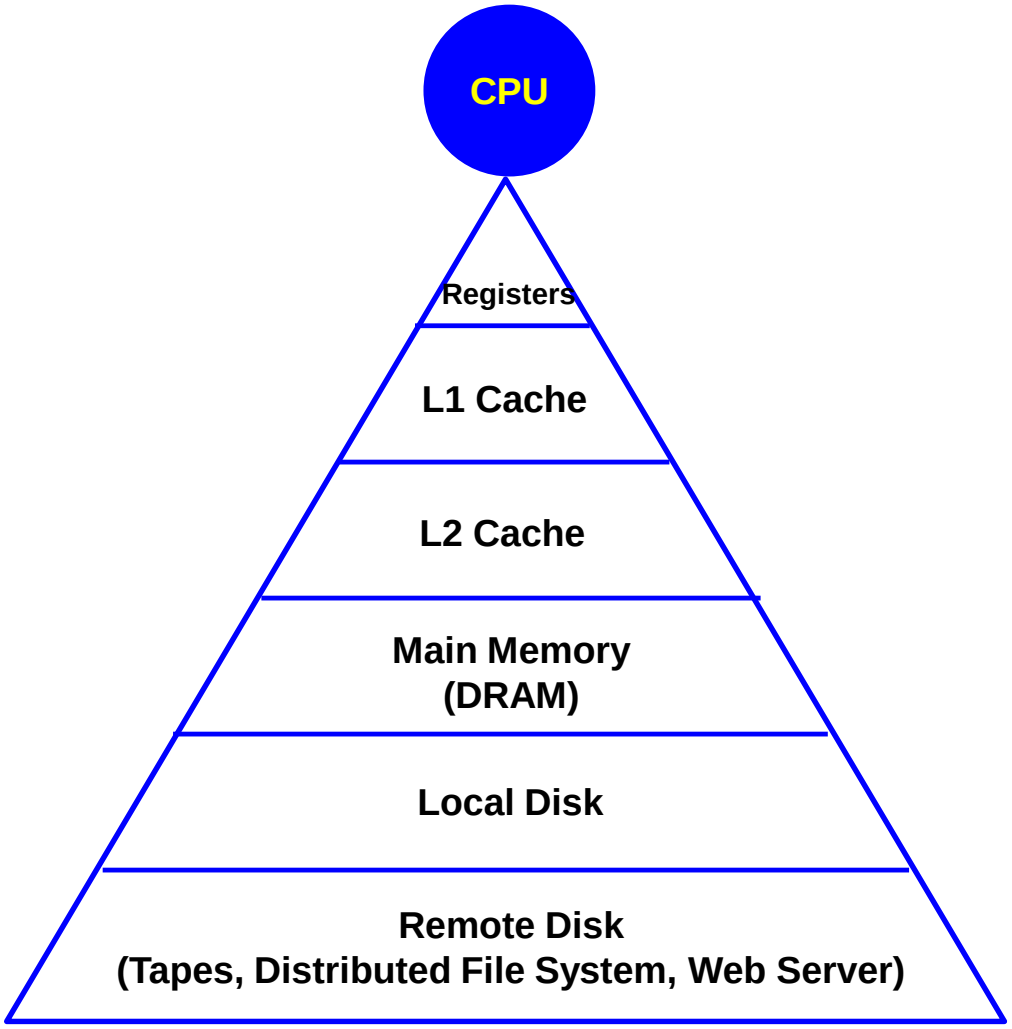
Widening Gap Between Processor and Memory Performance



Source: Trends in the relative performance of floating point operations and data access for HPC over the past 25 years, from John McCalpin's SC16 Invited Talk.

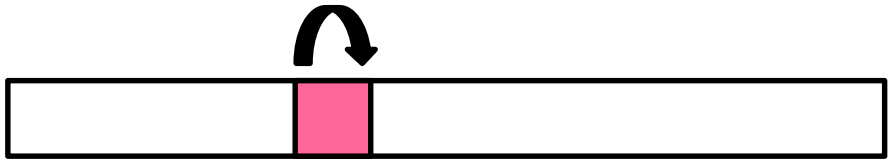
How Cache Works?

- CPU references (i.e., reads or writes) data from the main memory.
  - If referenced data is in cache (cache hit), data is served directly by cache (~1 cycle, fast).
  - If referenced data is not in cache (cache miss), fetch data from memory and also store it in the cache (~200 cycles, slow).

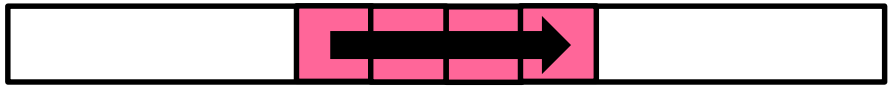


Why Cache Helps to Reduce Latency?

- Data tend to exhibit locality of reference.
  - Temporal Locality: recently referenced data tend to be referenced again in near future.



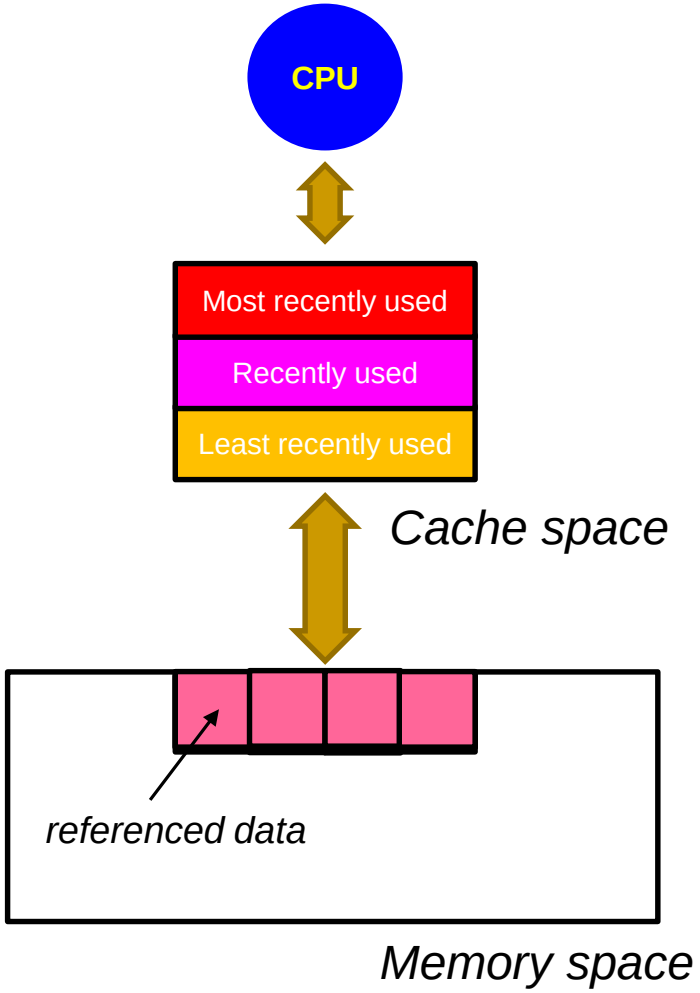
  - Spatial Locality: consecutive data items in memory tend to be referenced together.



How Cache Explores Locality?

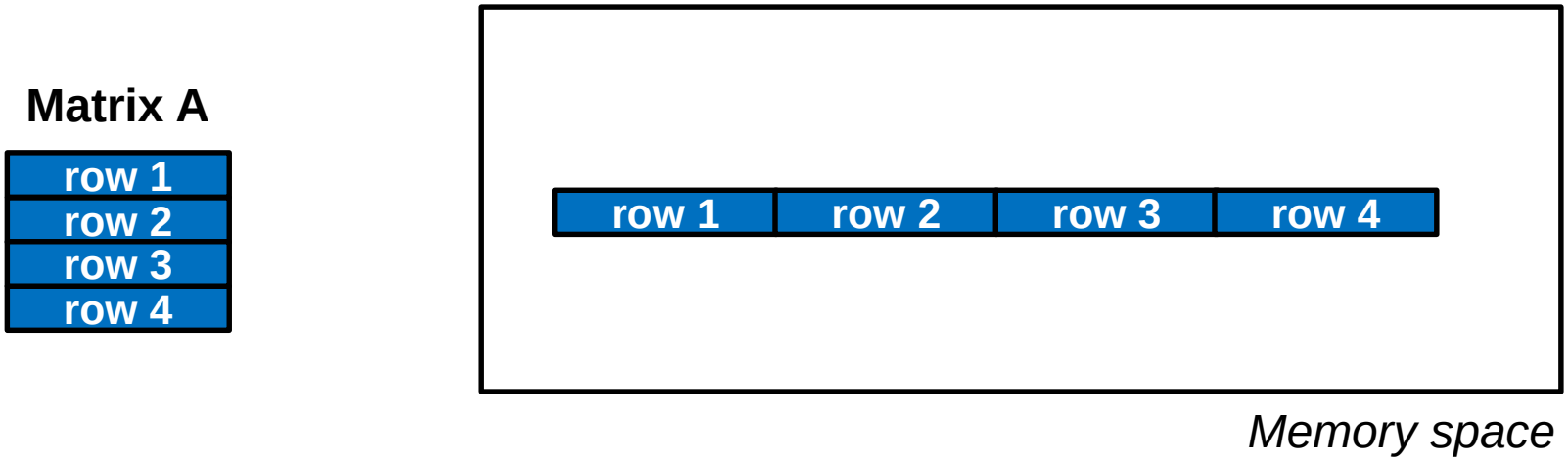
- Explore temporal locality:
  Store more recently used data (e.g., “least recently used (LRU)” cache replacement policy).

- Explore spatial locality:
  Bring a block of consecutive data from memory to cache for each reference.



How Programmer Explores Locality?

- Design algorithms and write codes that are cache-friendly.
- Reuse data as much as possible with regular data access patterns.
- Be aware of data layout in memory!
  - In C/C++, matrices are laid out in row-major order.

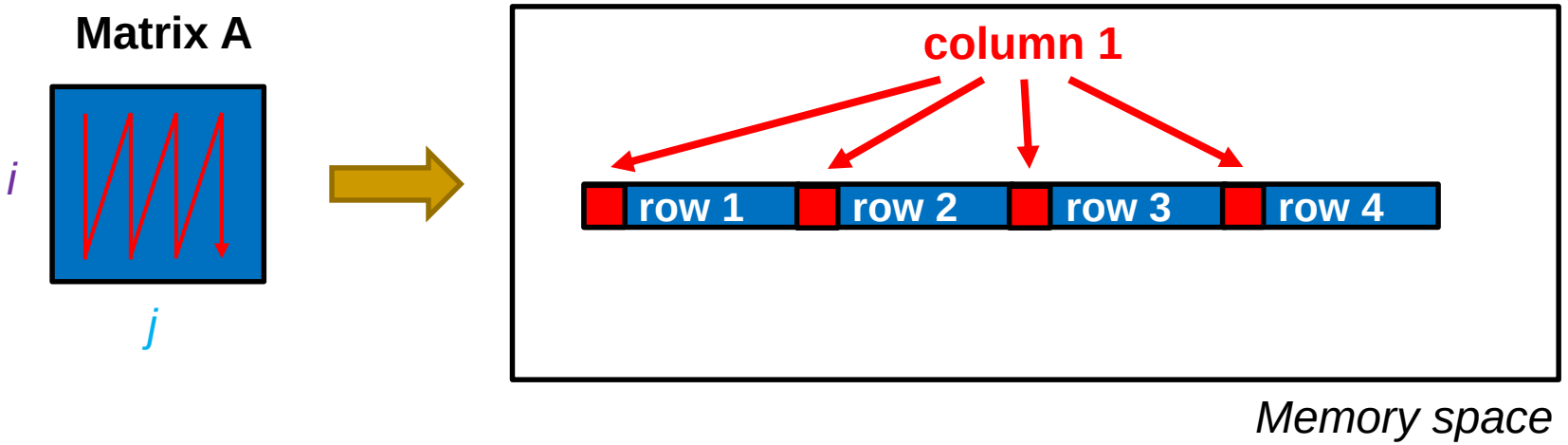


How Programmer Explores Locality?

- Explore spatial locality
  - column-by-column access has poor spatial locality ☹

Example: *Incrementing each element of a large matrix A*

```
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        A[i][j] += 1;
```

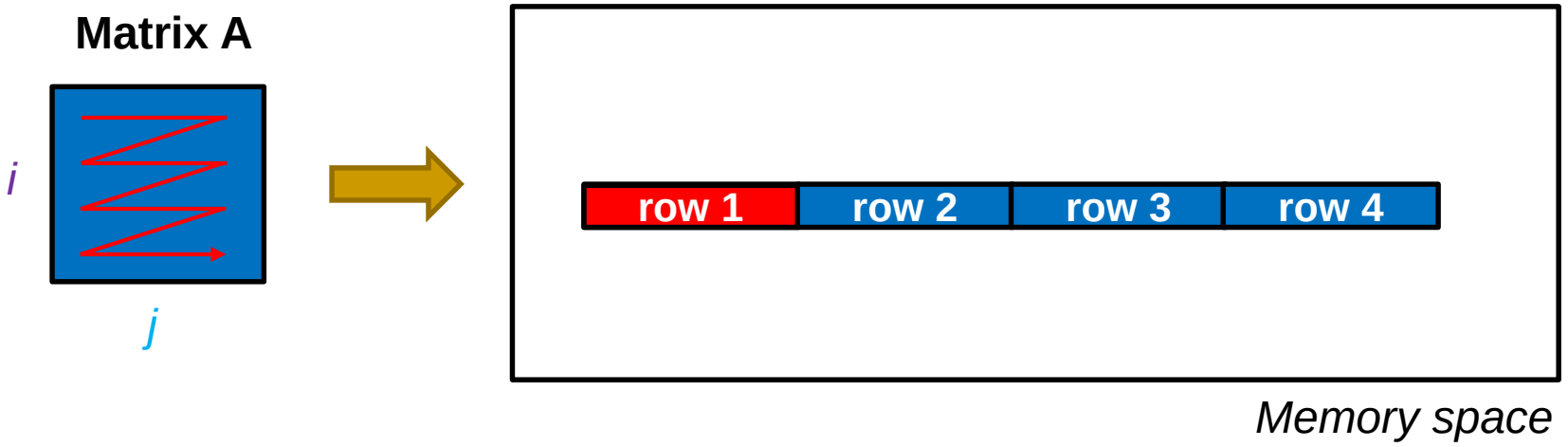


How Programmer Explores Locality?

- Explore spatial locality
  - row-by-row access has good spatial locality ☺

Example: *Incrementing each element of a large matrix A*

```
for (i=0; i<N; j++)
    for (j=0; j<N; j++)
        A[i][j] += 1;
```



How Programmer Explores Locality?

- Explores temporal locality

Poor temporal locality ☹
Each new access of A[i][j] is too far apart in time from the previous access, element will be replaced in cache ➔ cache miss.

Example: ***Accessing each element of a large matrix A multiple times***

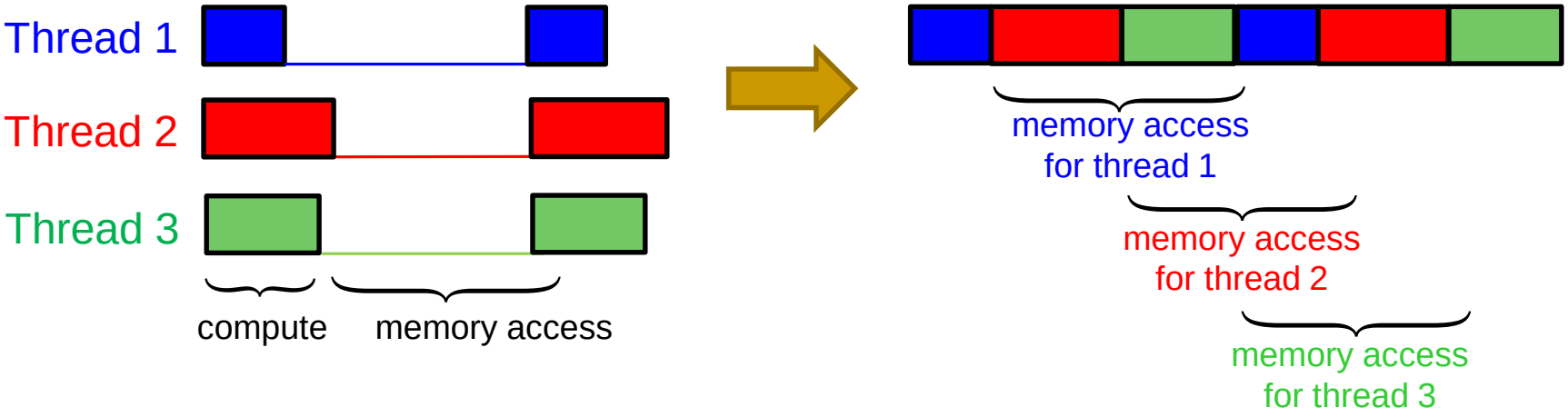
```
for (loop=0; loop<10; loop++)
   for (i=0; i<N; j++)
      for (j=0; j<N; j++)
         access (A[i][j]);
```

Good temporal locality ☺
All accesses of A[i][j] are performed together before accessing next element.

```
for (i=0; i<N; j++)
   for (j=0; j<N; j++)
      for (loop=0; loop<10; loop++)
         access (A[i][j]);
```

Other Approach to Hide Mem. Latency

- Overlapping computation with memory access
  - **Multithreading**: creating more threads than number of available processors/cores. While one thread is waiting for memory, another thread could keep core busy. (explored by GPU, but need high mem. bandwidth)



  - **Prefetching**: advance load needed data from memory, relying on compiler to resolve dependency.

Design of Parallel Programs

Parallel Program Design

Four-step guideline to the design of parallel programs:

1) **Partitioning**: decompose the problem into many small tasks. Focus on identifying parallelization opportunities and ignore number of processors.
2) **Communication**: Identify communication patterns required to coordinate task executions.
3) **Agglomeration**: Combine small tasks to improve parallel processing efficiency and reduce communication cost.
4) **Mapping**: Assign tasks to processors to balance load among processors, maximize resource utilization, and optimize performance.

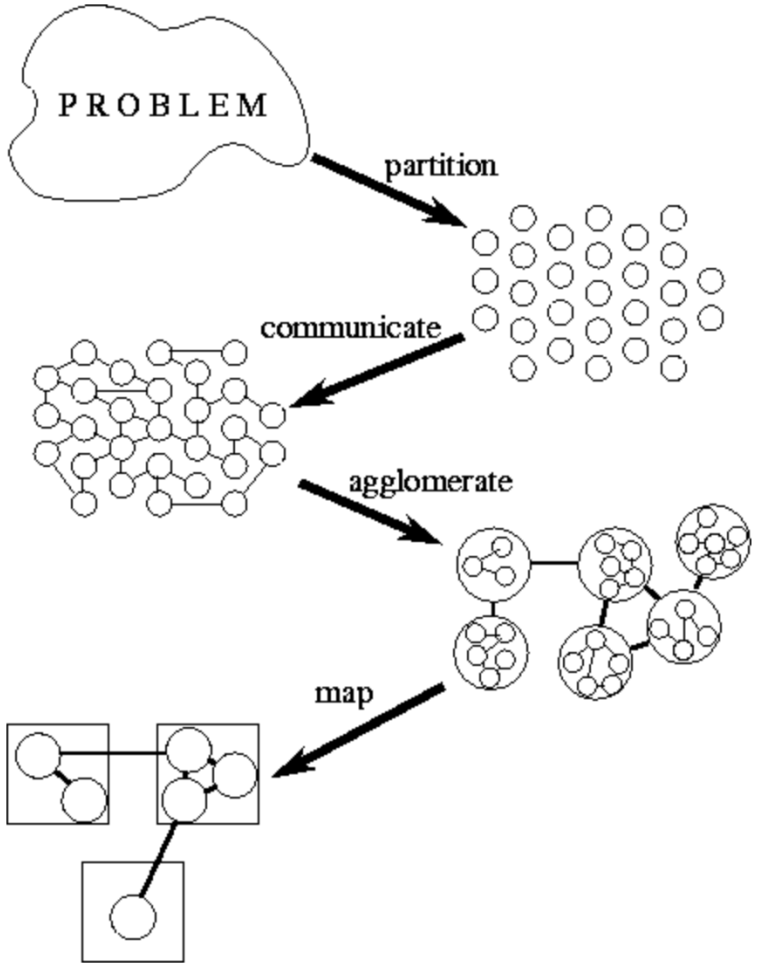
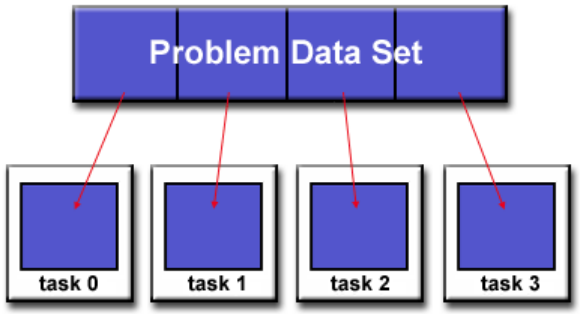


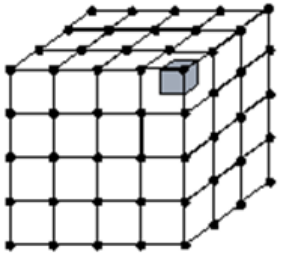
Image source: "Designing and Building Parallel Programs (by Foster)"

(1) Partitioning

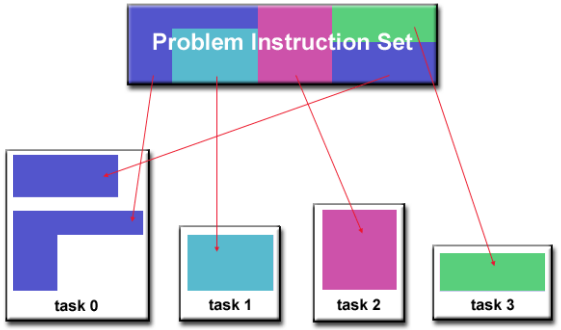
- **Domain Decomposition** (*Data-Parallel Model*)



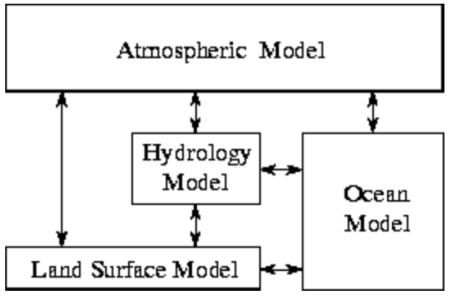
Tasks (shaded) for a problem involving 3D grid



- **Functional Decomposition** (*Task-Parallel Model*)



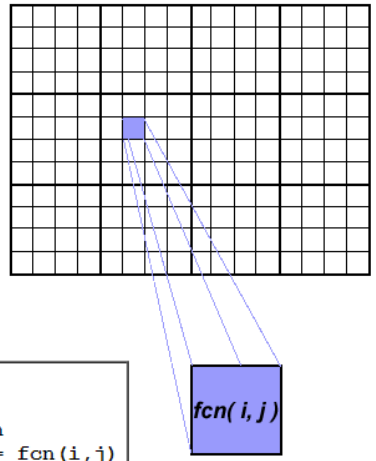
Tasks in a climate model involving different sub-models



(2) Communication

- **No Communication** (*Embarrassingly parallel*)

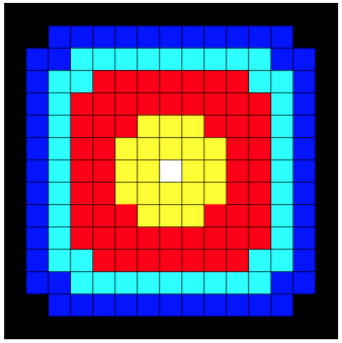
Independent processing of each array element a(i, j) by a function fcn(i, j)

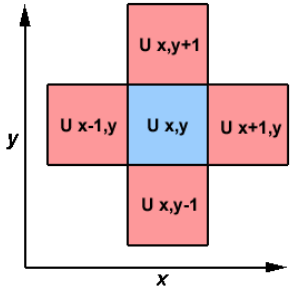



```
do j = 1,n
  do i = 1,n
    a(i,j) = fcn(i,j)
  end do
end do
```

- **Local Communication (Point-to-Point)** (*Require data from neighboring tasks*)

Finite difference method (FDM) in 2D heat equation



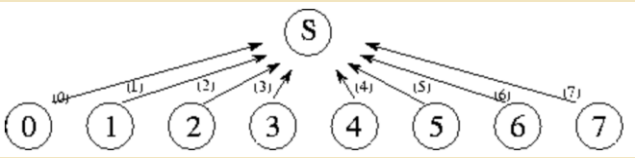


$$U_{x,y}^{(t+1)} = U_{x,y}^{(t)} + c \cdot (U_{x-1,y}^{(t)} + U_{x+1,y}^{(t)} + U_{x,y-1}^{(t)} + U_{x,y+1}^{(t)} - 4U_{x,y}^{(t)})$$

(2) Communication

- **Global Communication (Collective)**
  (*Performed together by many tasks*)

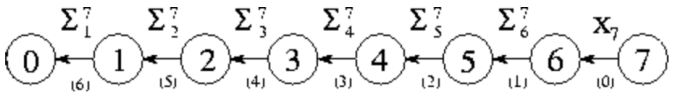
Example: compute ***reduction*** $S = \sum_{i=0}^{N-1} x_i$ *, where each processor holds one* x_i



Sequential implementation: $O(N)$

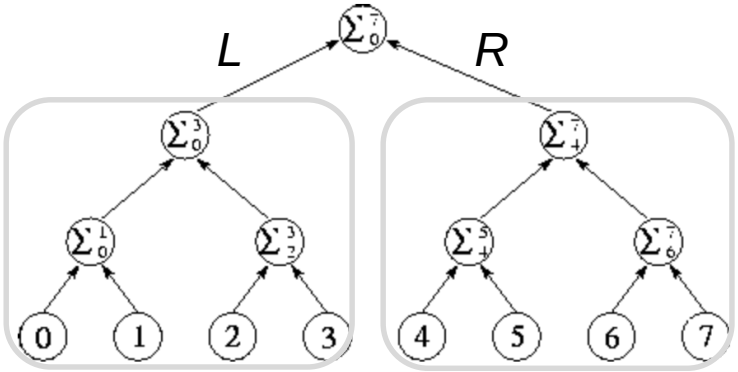
for $i = 0$ to $N - 1$

$S \leftarrow S + x_i$



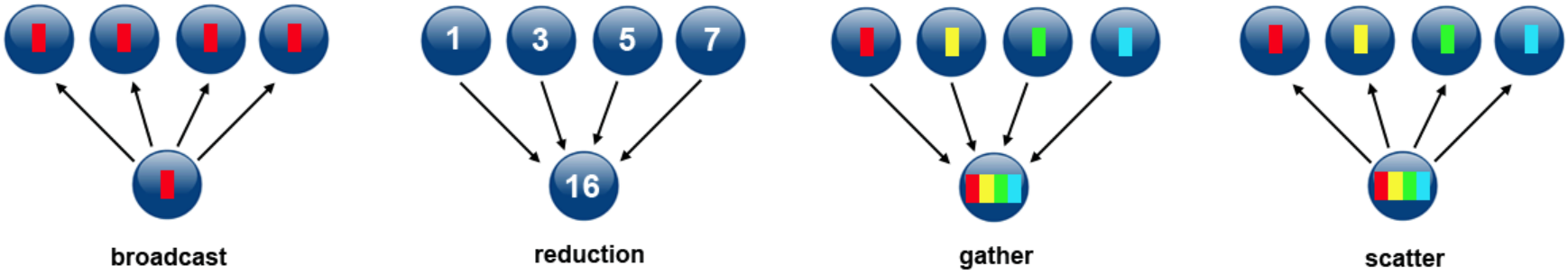
Divide-and-conquer implementation: $O(\log N)$

- Partition problem into two subproblems L and R
- Solve subproblem L recursively
- Solve subproblem R recursively
- Combine solutions of L and R



(2) Communication

- **Global Communication (Collective)**



Reducing Communication Overhead

- **Latency**: time delay in sending one message.
- **Bandwidth**: amount of data that can be send per unit of time.

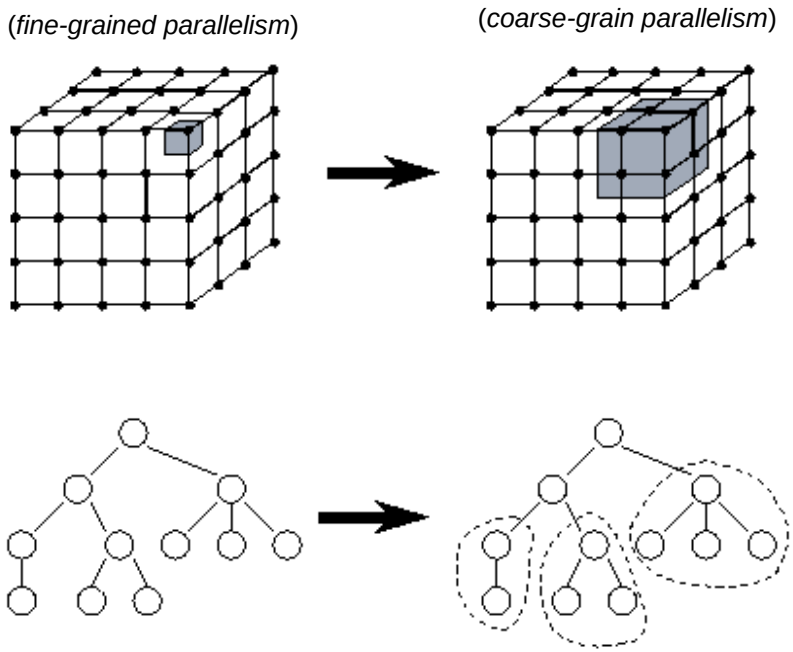
  ➔ *Preferable to send fewer large messages than lots of small messages.*

- **Synchronous (blocking)**: other work must wait until the communications have completed.
- **Asynchronous (non-blocking)**: other work can be done while communications take place.

  ➔ *Preferable to interleave computation with communication using asynchronous approach.*

(3) Agglomeration

- This step revisits partitioning and communication decisions by forming tasks of appropriate size.
  - Smaller tasks run less efficiently and require more communication, but they enable better load balancing.
  - Larger tasks execute more efficiently on some architectures with less communication, but they are harder to schedule.

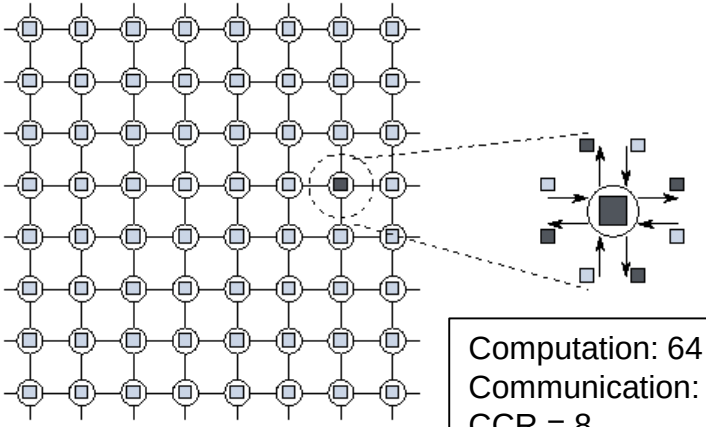


After this step, # tasks should remain > # processors for flexibility of scheduling

(3) Agglomeration

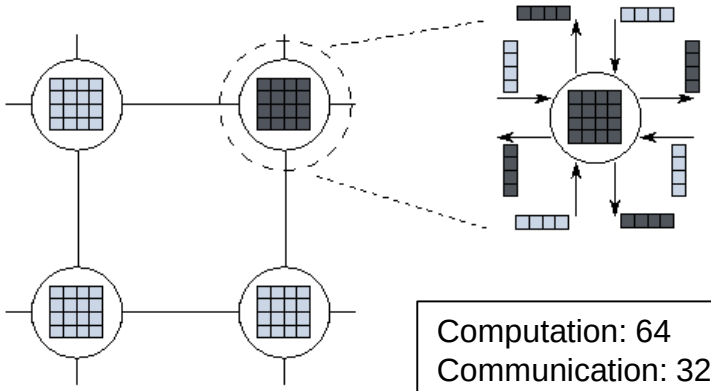
- **Adjusting Granularity**: decreasing communication-to-computation ratio (CCR).

Fine-grained partitioning of an 8x8 mesh



Computation: 64
Communication: 64*8
CCR = 8

Coarse-grained partitioning of an 8x8 mesh



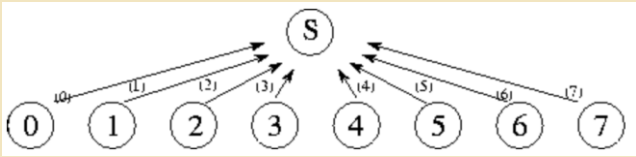
Computation: 64
Communication: 32*4
CCR = 2

Surface-to-volume effect: *Preferable to use higher-dimensional decompositions (i.e., agglomerating tasks in all dimensions) than reducing the dimension of the decomposition.*

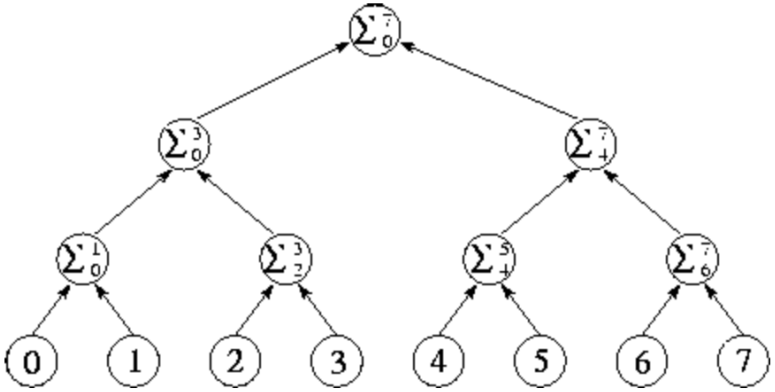
(3) Agglomeration

- **Replicating Computation**: explore tradeoff between computation and communication.

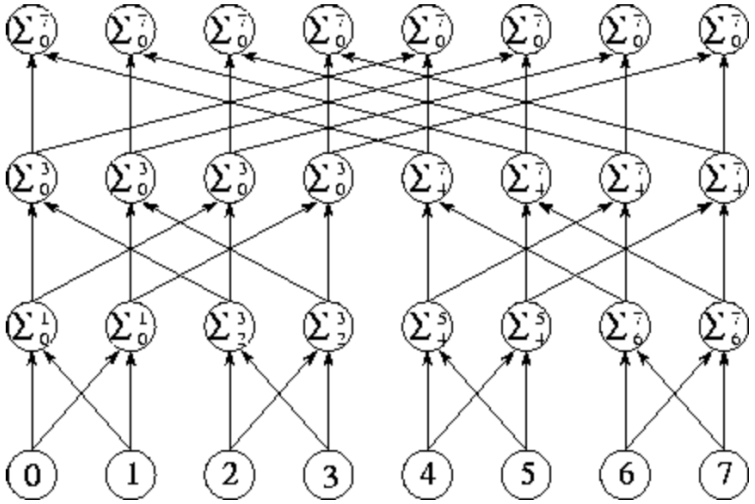
Example: compute ***reduction*** $S = \sum_{i=0}^{N-1} x_i$ *, and* ***broadcast*** *the result to all processors*



Reduction + Broadcast:
Computation: N
Communication: $2 \log N$

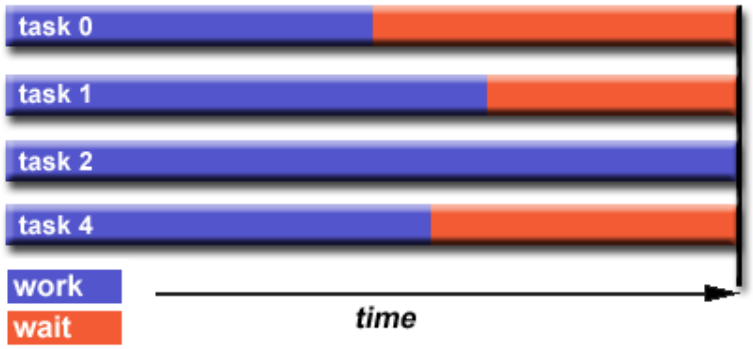


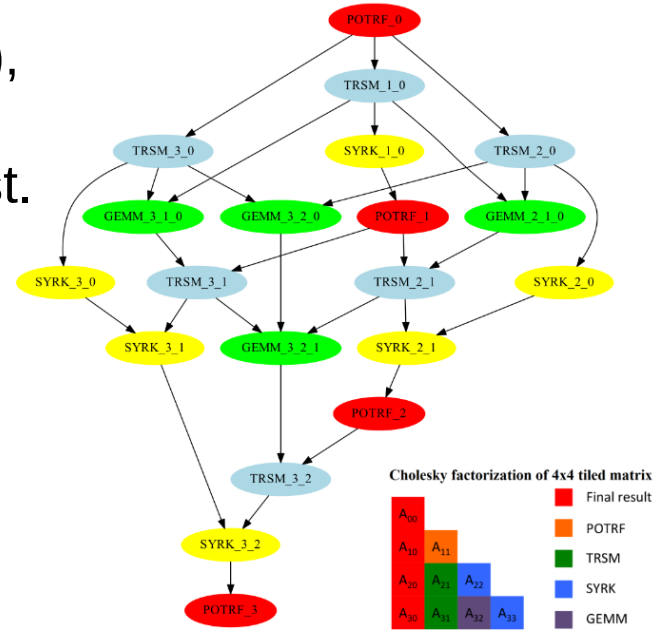
Simultaneous Reduction (Butterfly):
Computation: $N \log N$
Communication: $\log N$



(4) Mapping

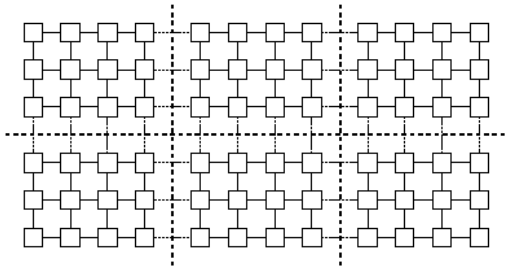
- This step assigns tasks to processors
  - Balance the workload of different processors.
  - Increase utilization of the system (or reduce the idle time).
  - Minimize execution time of applications.
- Generally a difficult problem (NP-complete), even for tasks without *dependencies (communications)*, but many heuristics exist.

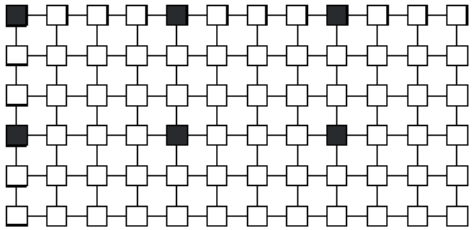




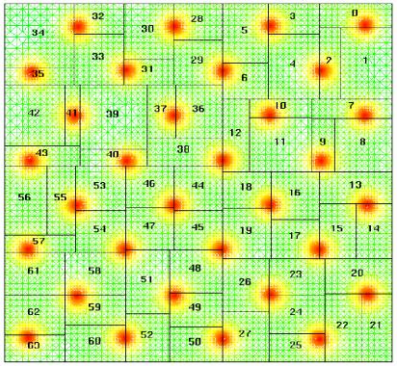
(4) Mapping

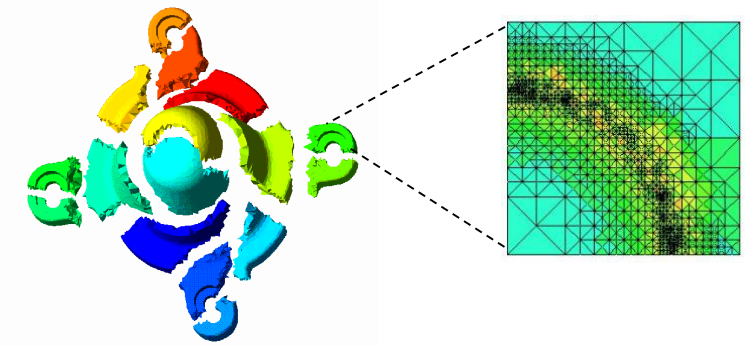
- **Load Balancing (Domain Decomposition) – Static**
  - Blocked or cyclic partition for regular data (e.g., dense matrix, mesh, grid)





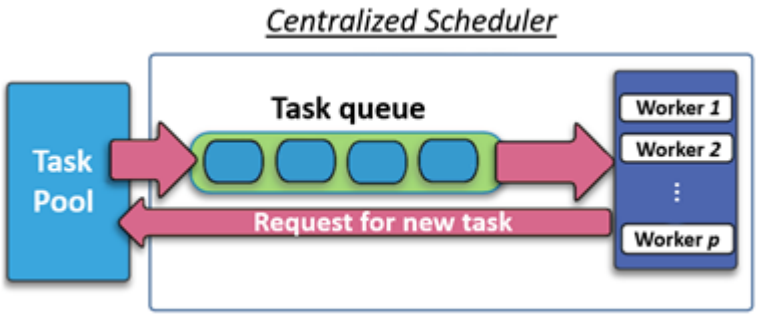
  - Recursive coordinate/graph bisection for irregular data (e.g., sparse graph/matrix)

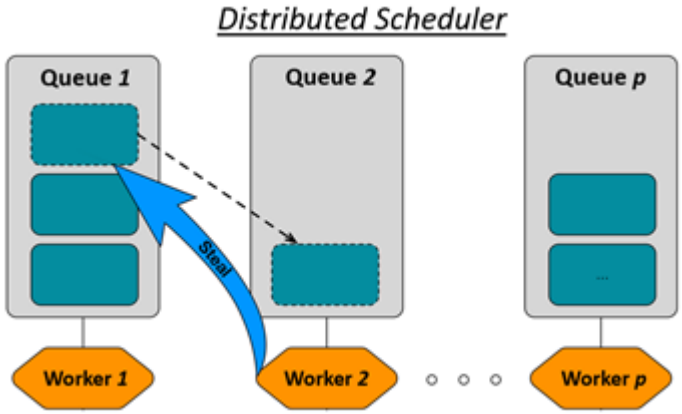




(4) Mapping

- **Task Scheduling (Functional Decomposition) – Dynamic**
  - Centralized approach (work sharing)
    - Central task queue & pool of tasks.
    - A processor requests a new task from central queue if it finishes its current task.
    - Generally better performance but high overhead.
  - Distributed approach (work stealing)
    - Each processor has own task queue.
    - A processor randomly “steals” a task from another task queue if no work.
    - Usually more scalable with less overhead.





Summary

1) **Partitioning**:
   - Domain decomposition (data-parallel)
   - Functional decomposition (task-parallel)
2) **Communication**:
   - No communication (embarrassingly parallel)
   - Local communication (point-to-point)
   - Global communication (collective)
3) **Agglomeration**:
   - Adjusting task granularity (reducing CCR)
   - Replicating computation (tradeoff between computation and communication)
4) **Mapping**:
   - Static load balancing: block/cyclic partition (for regular data), recursive bisection (for irregular data)
   - Dynamic task scheduling: centralized work sharing, distributed work stealing

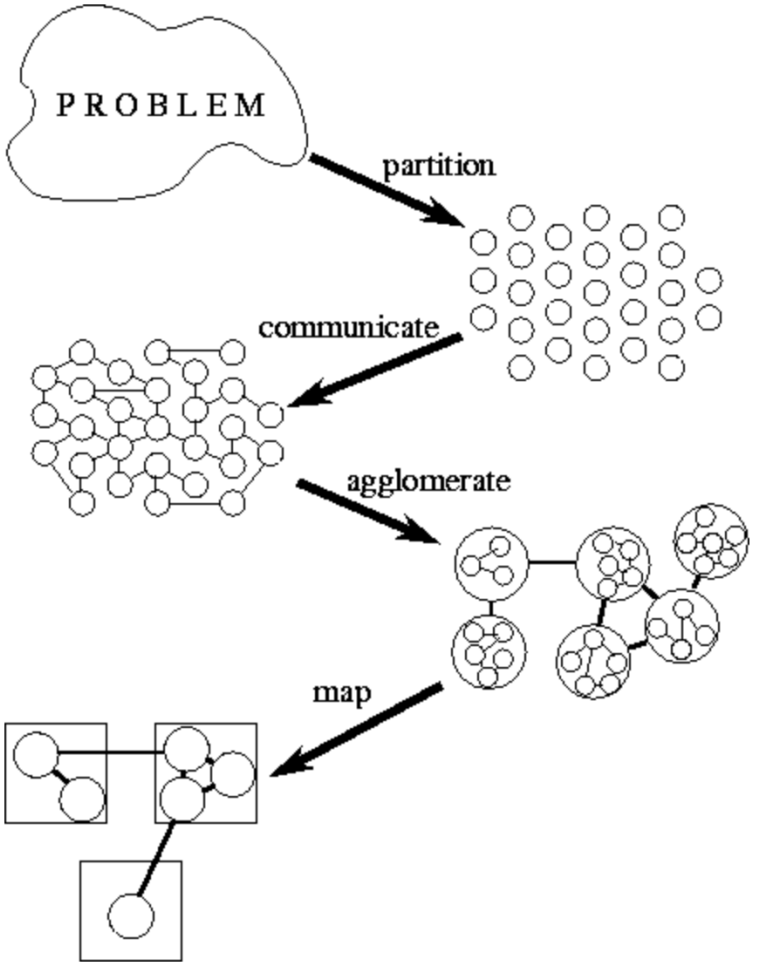
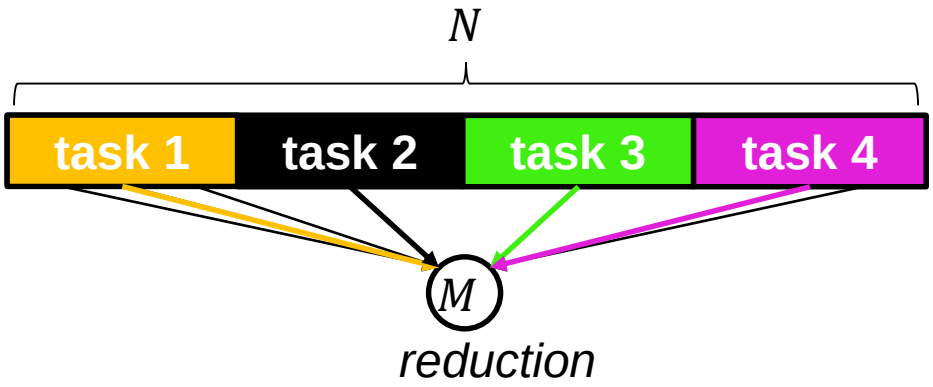
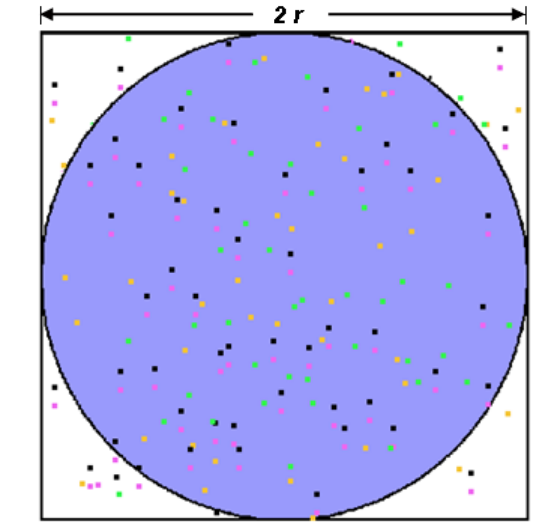


Image source: "Designing and Building Parallel Programs (by Foster)"

Case Study (1): Approximating π

1) **Partitioning**: create one task per point (generation and testing), array of N tasks.
2) **Communication**: no communication among tasks, one global reduction to compute M.
3) **Agglomeration**: combine multiple tasks and pre-compute local M.
4) **Mapping**: statically map tasks to processors (if homogeneous), or dynamically schedule tasks to processors (if heterogeneous).



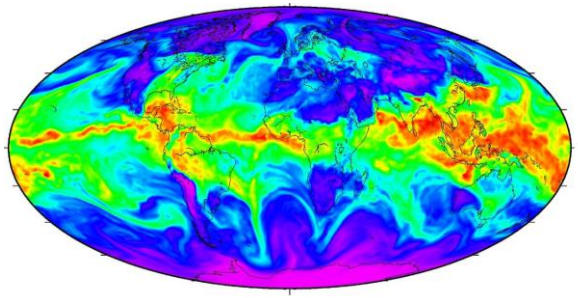


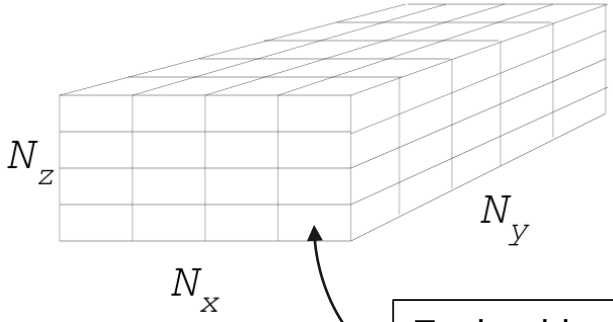
Monte-Carlo method:
Randomly generate N points in within square, and if M of them fall inside circle:

$$\pi \approx 4 \times \frac{M}{N}$$

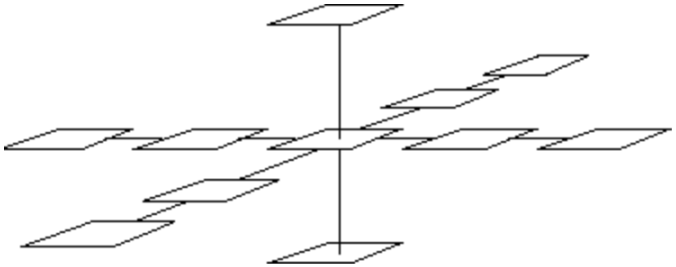
Case Study (2): Atmosphere Model

- Simulates atmospheric processes (e.g., wind, clouds, precipitation) to study evolution of tornadoes, to forecast weather, etc.
- Solves a set of partial differential equations (PDEs) in a continuous space, approximated by finite elements in a 3D grid.





Each grid point maintains a vector of values, representing pressure, temperature, wind velocity, humidity, etc.



Finite difference method: at each step, the state of a grid point is updated using a 9-point stencil in horizontal dimension and a 3-point stencil vertically.

Case Study (2): Atmosphere Model

- Computations involved:
  - **State computation**: for each grid point, simulate state using 9-point stencil horizontally, and 3-point stencil vertically.
  - **Global operation**: periodically compute the total mass of atmosphere to verify the correctness of simulation:

$$\text{Total mass} = \sum_{i=0}^{N_x-1} \sum_{j=0}^{N_y-1} \sum_{k=0}^{N_z-1} M_{i,j,k}$$

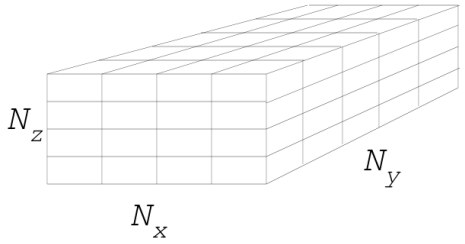
  - **Physics computation**: many additional physics simulations (e.g., radiation, clear sky) using numerical methods with dependencies across vertical columns only, e.g., total clear sky (TCS) at grid point (i, j, k) defined as:

$$\text{TCS}_{i,j,k} = \Pi_{\ell=1\cdots k}(1 - \text{cld}_\ell) \times \text{TCS}_{i,j,1} = (1 - \text{cld}_k)\text{TCS}_{i,j,k-1}$$

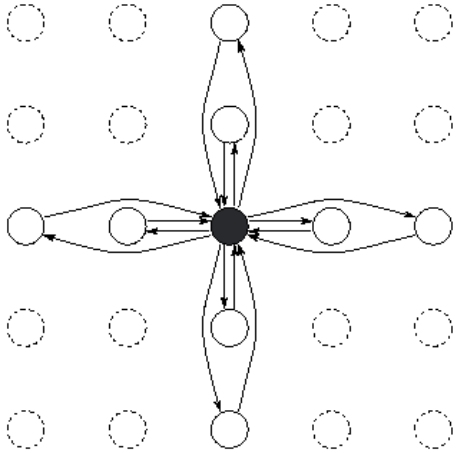
cloud fraction at level ℓ

Case Study (2): Atmosphere Model

1) **Partitioning**:
   - Domain decomposition with $N_x \times N_y \times N_z$ tasks, one per each grid point.



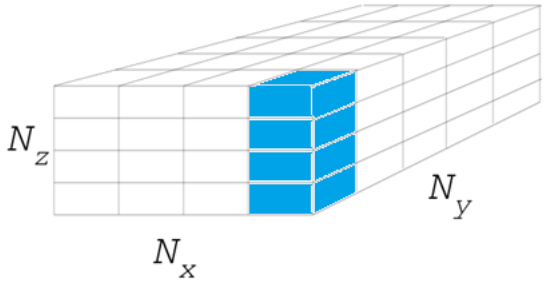
2) **Communication**:
   - **Local communication**: at each time step, a grid point exchanges 16 messages with 8 horizontal neighbors (for state computation) and many more vertically (for both state and physics computations).
   - **Global communication**: less frequently, a global reduction is done to check total mass.



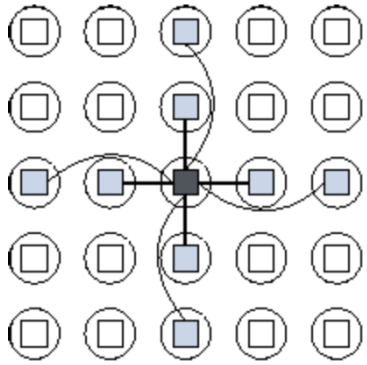
Case Study (2): Atmosphere Model

3) **Agglomeration**:

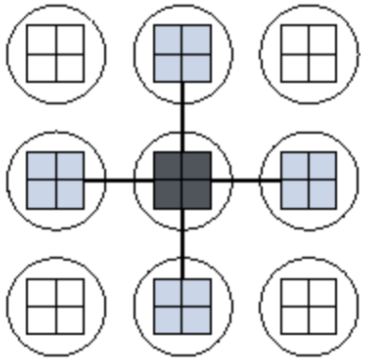
- Combine all tasks *vertically* in each column to drastically reduce number of communications.



- Combine neighboring tasks (e.g., 2x2) *horizontally* to further reduce communication; number of tasks after agglomeration is $(N_x \times N_y)/4$.



16 messages per one computation
➔ CCR = 16

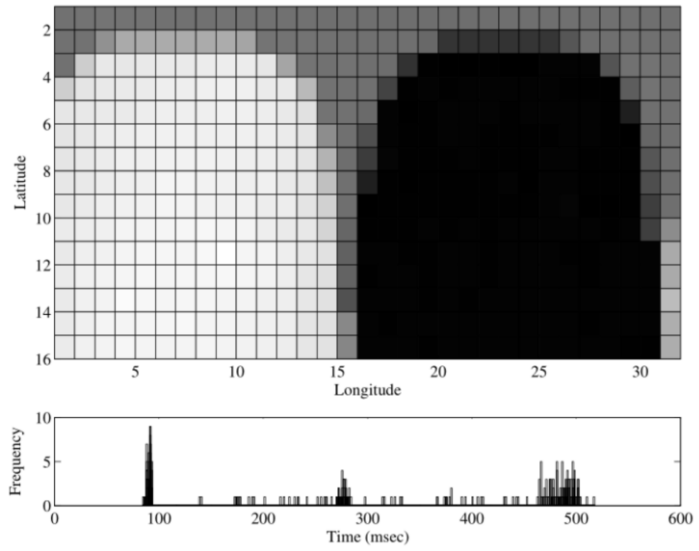


48 messages per 4 computations
➔ CCR = 12
(messages can be combined to further reduce latency)

Case Study (2): Atmosphere Model

4) **Mapping**:
   - Static with domain decomposition; loads vary significantly for different grid columns (e.g., day/night radiation, clouds based on humidity).
   - Tradeoff between load balance and communication overhead.

Blocked mapping has unbalanced load



Cyclic mapping improves load balance

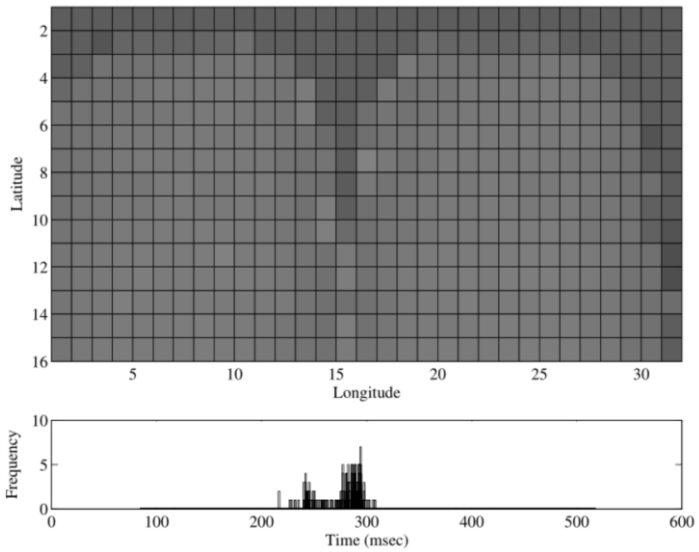
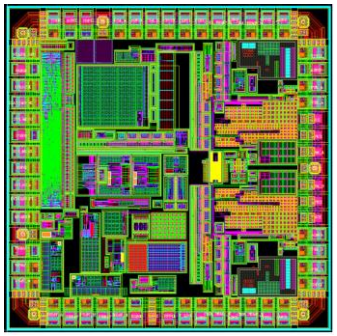
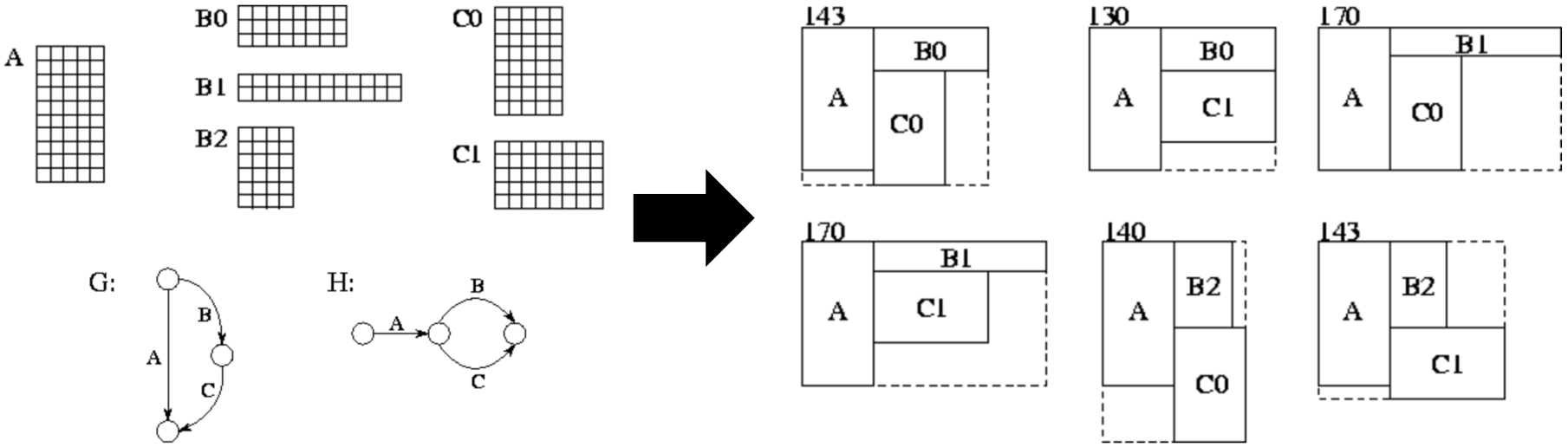


Image source: "Designing and Building Parallel Programs (by Foster)"

Case Study (3): Floorplan Optimization

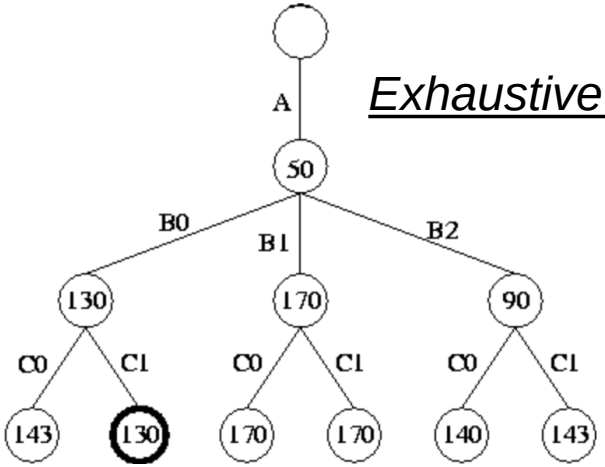
- Optimizes area/power in layout of VLSI.
  - **Input 1**: M cells, and cell c_i has $I(c_i)$ implementations.
  - **Input 2**: Polar graphs *G* and *H* specify vertical and horizontal adjacency relationship of cells.
  - **Output**: a layout of cells that minimizes total area of bounding rectangle.



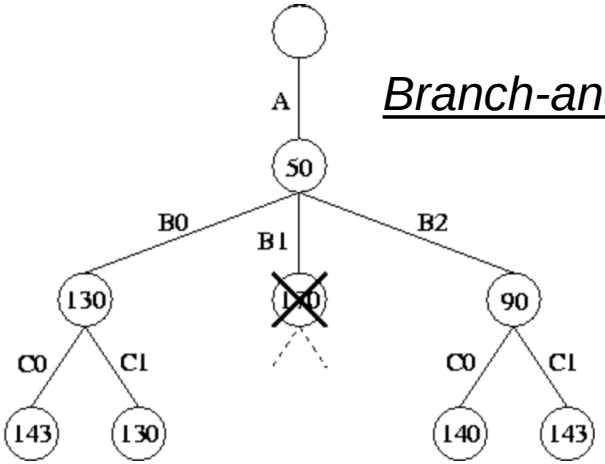


Case Study (3): Floorplan Optimization

- Exploring all possible configurations using a *search tree*
  - **Exhaustive search**: visit all nodes of a search tree, has total cost $\prod_{i=1\ldots M} I(c_i)$ e.g., 20 cells, each with 6 implementations
    ➔ search space = $6^{20} \approx 4 \times 10^{15}$.
  - **Branch-and-bound**: keep track of best solution so far (with area $A_{\min}$), and prune a branch if its area is already greater than $A_{\min}$
    ➔ significantly reduce the search space.



Exhaustive search



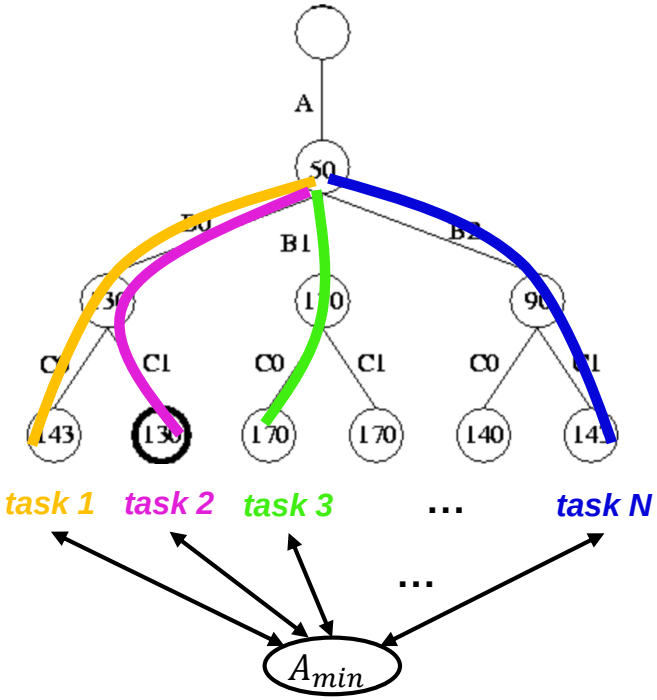
Branch-and-bound

Case Study (3): Floorplan Optimization

1) **Partitioning**:
   - functional decomposition; each task explores one possibility in search space. Number of tasks:

$$N = \Pi_{i=1\ldots M} I(c_i)$$

2) **Communication**: each task
   - checks $A_{\min}$ while going down the path.
   - updates $A_{\min}$ (if necessary) when complete.



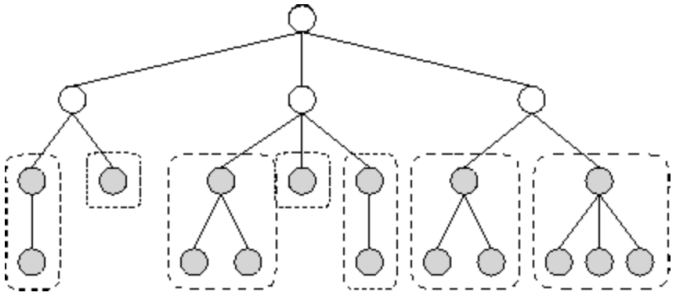
All tasks access centralized $A_{\min}$, leading to very high communication overhead;
➔ *Trade-off with pruning opportunities: only check A_{min} in a subset of tasks.*

Case Study (3): Floorplan Optimization

3) **Agglomeration**:
   - Combine all tasks in a subtree rooted at depth *D.* Number of tasks reduced to:

     $$N' = \Pi_{i=1\ldots D} I(c_i)$$

   - Maintain $A_{\min}$ independently in each subtrees, which synchronize periodically.
   - Use *depth-first search* for each agglomerated task (improve the effectiveness of pruning).



4) **Mapping**: functional decomposition (scheduling tasks to procs), since tasks do not have same size (due to pruning).
   - **Centralized scheduling**: a processor, when idle, requests a new task from the pool.
   - **Distributed work-stealing**: initially allocate root node to a single processor, which explores search tree in depth-first manner. A processor, when idle, steals work from others in top-most node in search path.

Case Study (4): Computational Chemistry

- A fundamental method in quantum chemistry is to compute ***Fock matrix F*** (size $N \times N$ representing electronic structure of atoms/molecules):

$$F_{ij} = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} D_{kl} \left(I_{ijkl} - \frac{1}{2} I_{ikjl} \right)$$

  - D: $N \times N$ matrix (read-only)
  - I: Integrals computed using 4 indices and an array A with $O(N)$ elements, representing approximation of repulsive forces between electrons.

- In theory, $2N^4$ integrals needed;
- In practice, $N^4/8$ is sufficient (by exploring redundancy and symmetry)

```
procedure fock_build
begin
  for i = 1 to N
     for j = 1 to i
        for k = 1 to j
           for l = 1 to k
              integral(i ,j ,k ,l )
           endfor
        endfor
     endfor
  endfor
end

procedure integral(i ,j ,k ,l )
begin
  I = compute_integral(i, j, k, l)
  F_ij = F_ij + D_kl I
  F_kl = F_kl + D_ij I
  F_ik = F_ik + D_jl I
  F_jl = F_jl - (1/2) D_ik I
  F_il = F_il - (1/2) D_jk I
  F_jk = F_jk - (1/2) D_il I
end
```

Compute $N^4/8$ Integrals

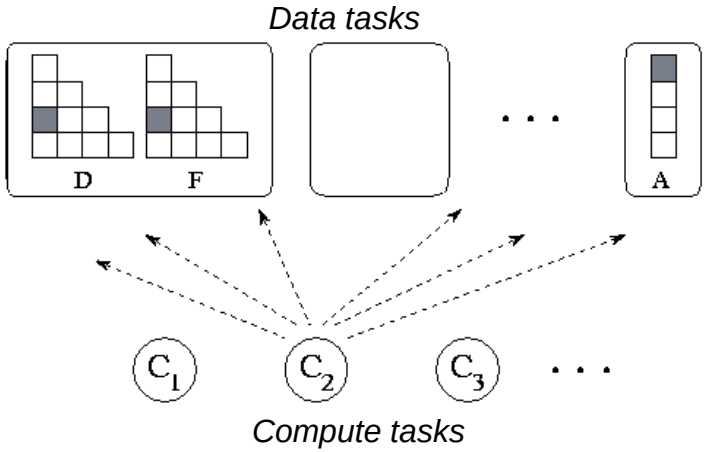
O(1) time using 4 elements in array *A*

Update 6 elements in *F* matrix

Case Study (4): Computational Chemistry

1) **Partitioning**:
   - **Domain decomposition**: each task computes one element F_{ij} of Fock matrix *(fewer tasks $O(N^2)$; cannot explore redundancy in computation).*
   - **Functional decomposition**: each task computes one integral I_{ijkl} *(more tasks $O(N^4)$; less total computation by exploring redundancy).*

2) **Communication**:
   - No obvious communication pattern (each matrix element accessed by many tasks).
   - **Asynchronous communication**: $O(N^2)$ dedicated tasks for responding to request to read and write data (each responsible for one element in D, F, A matrices).
   - Each compute task requests $O(1)$ elements ➔ totally $O(N^4)$ messages.

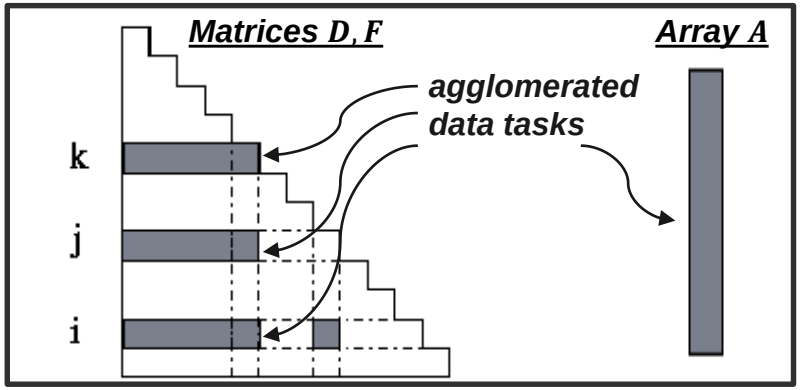


Case Study (4): Computational Chemistry

3) **Agglomeration**:
   - **Compute tasks**: combine integrals in inner loop ➔ $O(N^3)$ tasks.
   - **Data tasks**: combine & replicate elements in D, F, A ➔ $O(N)$ tasks.
   - Each compute task requests data from four data tasks ➔ $O(N^3)$ messages.

```
procedure fock_build
begin
  for i = 1 to N
     for j = 1 to i
        for k = 1 to j
           for l = 1 to k
              integral(i ,j ,k ,l )
           endfor
        endfor
     endfor
  endfor
end
```

agglomerated compute task



4) **Mapping**:
   - **Static**: compute & data tasks with same i index mapped to same processor. ➔*better locality (sharing of the i^{th} row of matrix), but unbalanced load.*
   - **Dynamic**: standard approaches to schedule tasks to idle processors.

Analysis of Algorithms

- $T(n)$: serial execution time of an algorithm for a problem of size n.

- **Asymptotic Analysis**
  - **Big-O notation (Upper bound)**: $T(n) = O(f(n))$ if there exist positive constants c and n_0 such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$.
  - **Big-Ω notation (Lower bound):** $T(n) = \Omega(f(n))$ if there exist positive constants c and n_0 such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$.
  - **Big-Θ notation:** $T(n) = \Theta(f(n))$ if there exist positive constants c_1, c_2 and n_0 such that $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$.

- **Time complexity**
  - Polynomial time: $T(n) = O(n^{O(1)})$ → "easy" or "tractable"
  - Exponential time: $T(n) = O(2^{\text{poly}(n)})$ → "hard" or "intractable"

Parallel Algorithms

- $T(n, p)$: execution time of an algorithm using p processors for a problem of size n.
- Exponential-time serial algorithms will still take exponential time in parallel with polynomial number of processors (in terms of problem size). Otherwise, a serial algorithm could simulate the parallel algorithm and solve the problem in polynomial time.
- In this class, we are mainly interested in parallelizing polynomial-time algorithms for large problems (e.g., matrix multiplication, system of equations, sorting, FFT), but *we will not analyze these algorithms.*
- In practice, parallel computing is also used to speedup exponential-time algorithms or to get approximate solutions faster.

References

- I. Foster. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering, *Addison-Wesley Longman Publishing Co., Inc.*.
- A. Grama, A. Gupta, G. Karypis, V. Kumar. Introduction to Parallel Computing (2nd Edition). *Addison-Wesley Professional.*
- B. Barney. Introduction to Parallel Computing Tutorial, Lawrence Livermore National Laboratory. https://computing.llnl.gov/tutorials/parallel_comp/