Introduction to Parallel Programming

- Goal:
  - Method for developing efficient parallel algorithms that have little communication overhead, load imbalance and search overhead

- Learning goal:
  - You should be able to apply this method to simple cases

Introduction

- Language notation: message passing
- Distributed-memory machine
  - All machines are equally fast
  - E.g., identical workstations on a network

- 5 parallel algorithms of increasing complexity:
  - Matrix multiplication
  - Successive overrelaxation
  - All-pairs shortest paths
  - Linear equations
  - Traveling Salesman problem

Message Passing

- SEND (destination, message)
  - blocking: wait until message has arrived (like a fax)
  - non blocking: continue immediately (like a mailbox)

- RECEIVE (source, message)
- RECEIVE-FROM-ANY (message)
  - blocking: wait until message is available
  - non blocking: test if message is available

Syntax

- Use pseudo-code with C-like syntax
- Use indentation instead of { ..} to indicate block structure
- Arrays can have user-defined index ranges
- Default: start at 1
  - int A[10:100] runs from 10 to 100
  - int A[N] runs from 1 to N
- Use array slices (sub-arrays)
  - A[i..j] = elements A[i] to A[j]
  - A[i, *] = elements A[i, 1] to A[i, N] i.e. row i of matrix A
  - A[*, k] = elements A[1, k] to A[N, k] i.e. column k of A

Parallel Matrix Multiplication

- Given two N x N matrices A and B
- Compute C = A x B
- $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + .. + A_{iN}B_{Nj}$

$$\underset{\mathbf{A}}{\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}} \times \underset{\mathbf{B}}{\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}} = \underset{\mathbf{C}}{\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}}$$

Sequential Matrix Multiplication

```
for (i = 1; i <= N; i++)
  for (j = 1; j <= N; j++)
      C [i,j] = 0;
      for (k = 1; k <= N; k++)
          C[i,j] += A[i,k] * B[k,j];
```

The order of the operations is over specified

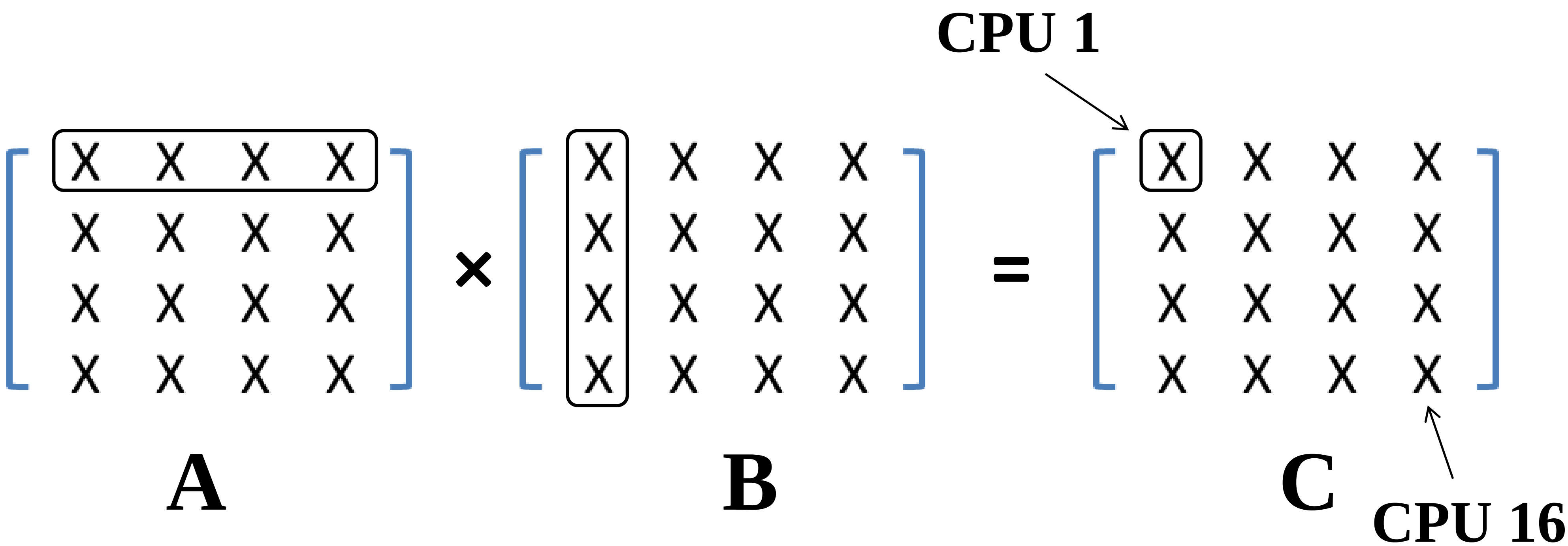
Everything can be computed in parallel

Parallel Algorithm 1

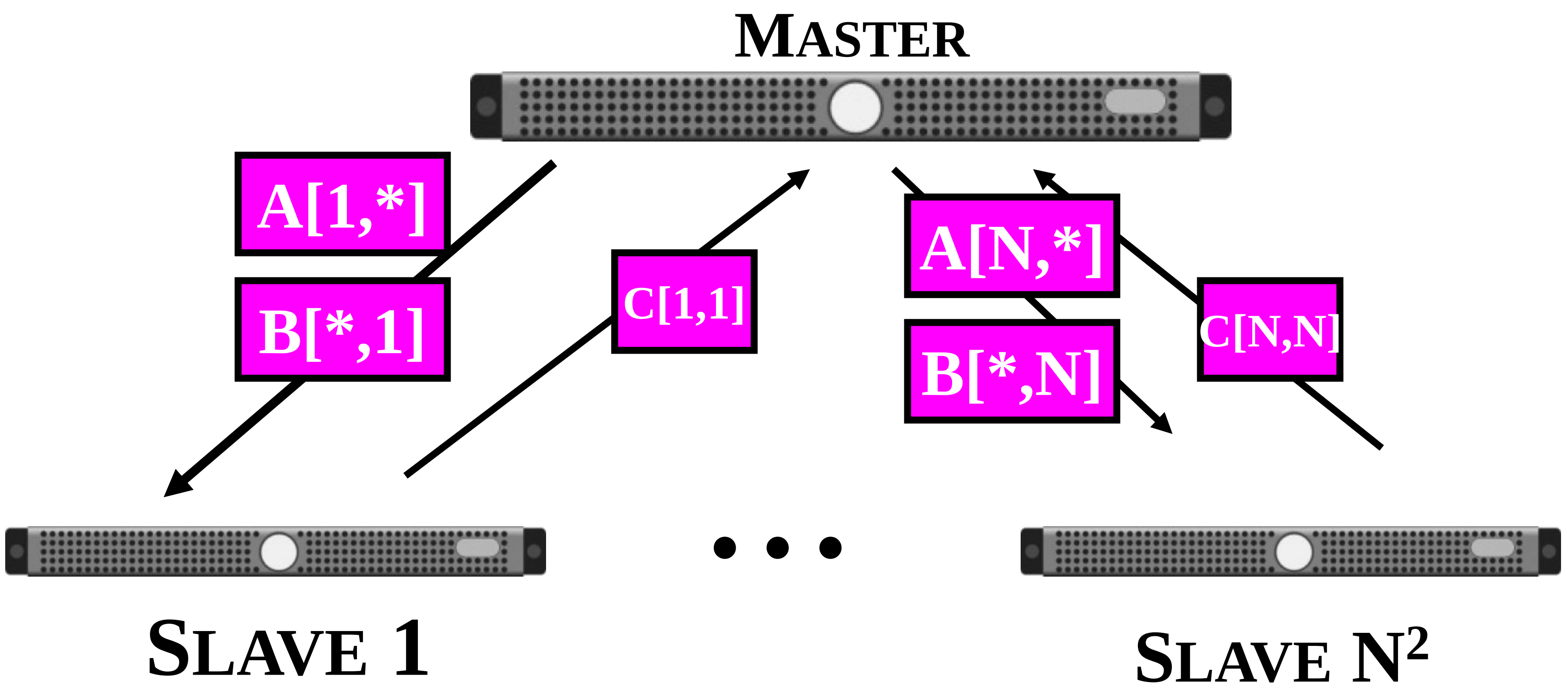
Each processor computes 1 element of C

Requires N^2 processors

Each processor needs 1 row of A and 1 column of B



Structure



Master distributes work and receives results

Slaves (1 .. P) get work and execute it

How to start up master/slave processes depends on Operating System

Parallel Algorithm 1

Master (processor 0):

```
int proc = 1;
for (i = 1; i <= N; i++)
    for (j = 1; j <= N; j++)
        SEND(proc, A[i,*], B[*,j], i, j); proc++;
for (x = 1; x <= N*N; x++)
    RECEIVE_FROM_ANY(&result, &i, &j);
    C[i,j] = result;
```

Slaves (processors 1 .. P):

```
int Aix[N], Bxj[N], Cij;
RECEIVE(0, &Aix, &Bxj, &i, &j);
Cij = 0;
for (k = 1; k <= N; k++) Cij += Aix[k] * Bxj[k];
SEND(0, Cij , i, j);
```

Efficiency (complexity analysis)

- Each processor needs O(N) communication to do O(N) computations
  - Communication: 2*N+1 integers = O(N)
  - Computation per processor: N multiplications/additions = O(N)
- Exact communication/computation costs depend on network and CPU
- Still: this algorithm is inefficient for *any* existing machine
- Need to improve communication/computation ratio

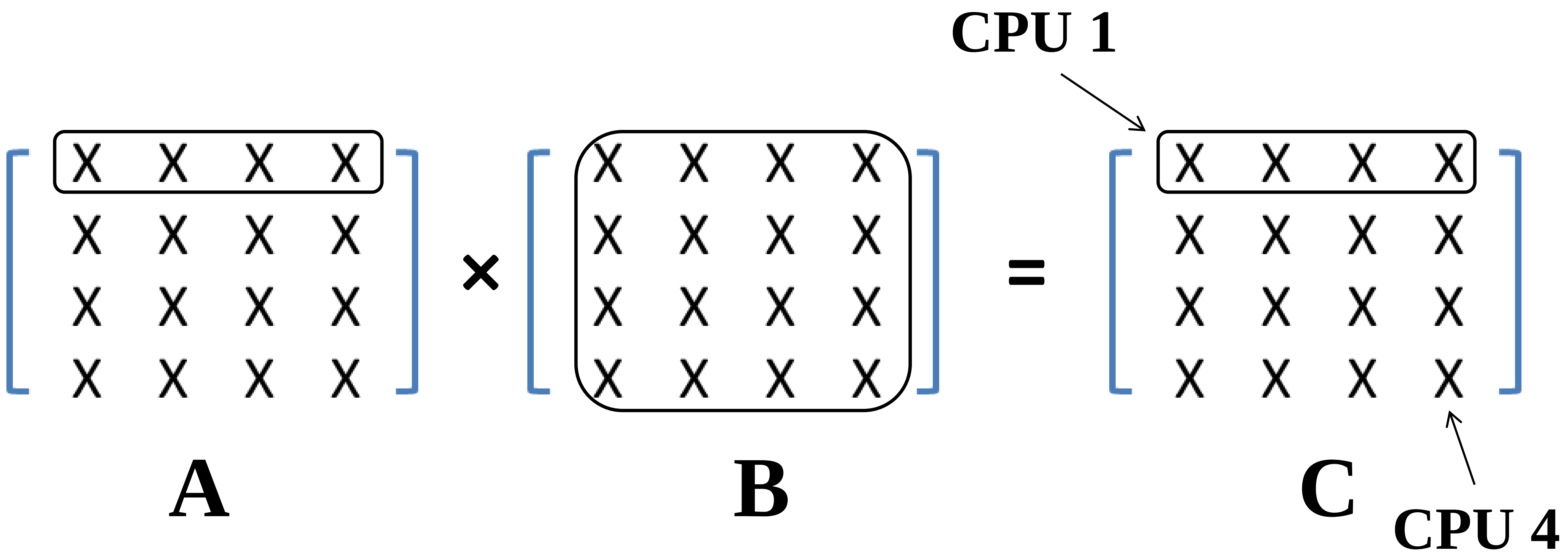
Parallel Algorithm 2

Each processor computes 1 row (N elements) of C

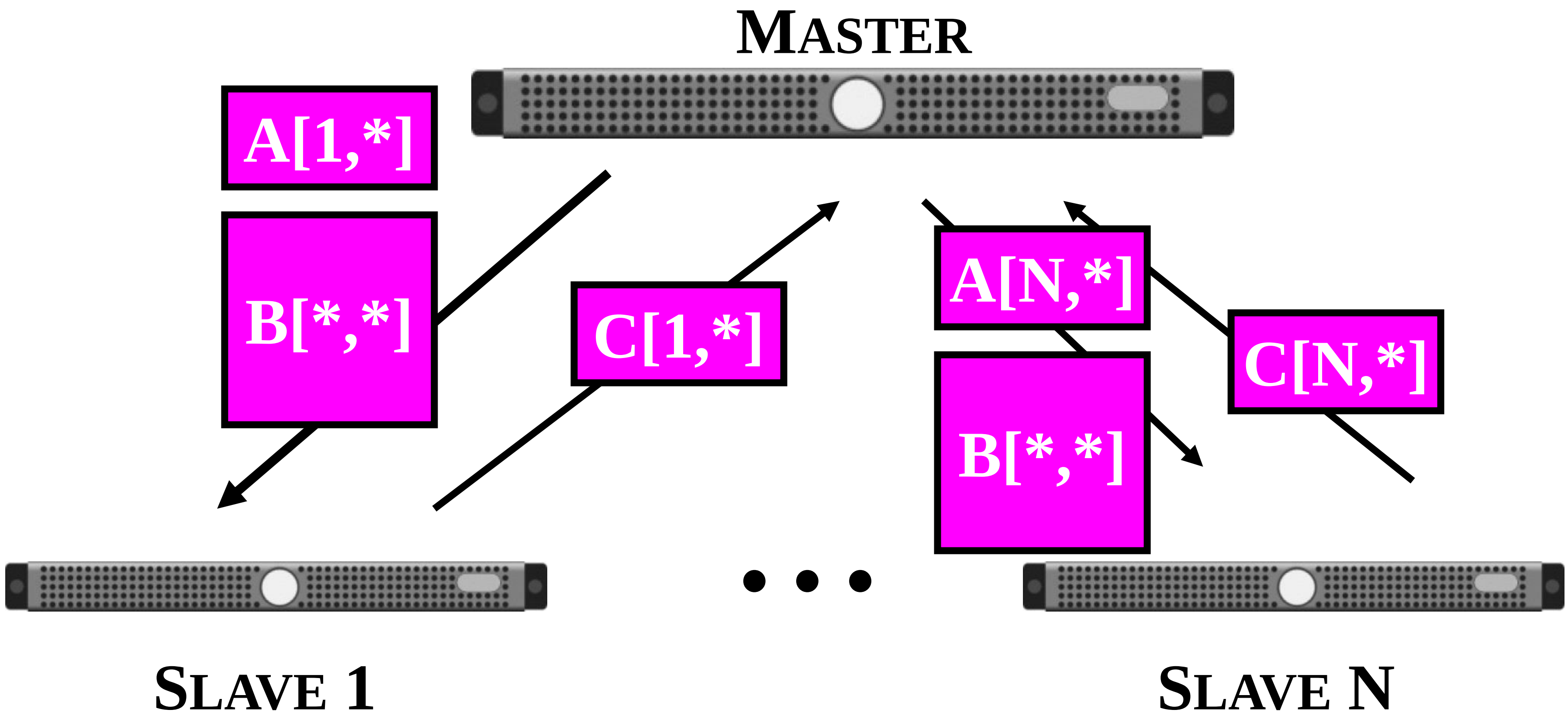
Requires N processors

Need entire B matrix and 1 row of A as input

Can re-use each row of A many (N) times



Structure



Parallel Algorithm 2

Master (processor 0):

```
for (i = 1; i <= N; i++)
    SEND (i, A[i,*], B[*,*], i);
for (x = 1; x <= N; x++)
    RECEIVE_FROM_ANY (&result, &i);
    C[i,*] = result[*];
```

Slaves:

```
int Aix[N], B[N,N], C[N];
RECEIVE(0, &Aix, &B, &i);
for (j = 1; j <= N; j++)
    C[j] = 0;
    for (k = 1; k <= N; k++) C[j] += Aix[k] * B[j,k];
SEND(0, C[*] , i);
```

Problem: need larger granularity

Each processor now needs $O(N^2)$ communication and $O(N^2)$ computation -> Still inefficient

Assumption: N >> P (i.e. we solve a *large* problem)

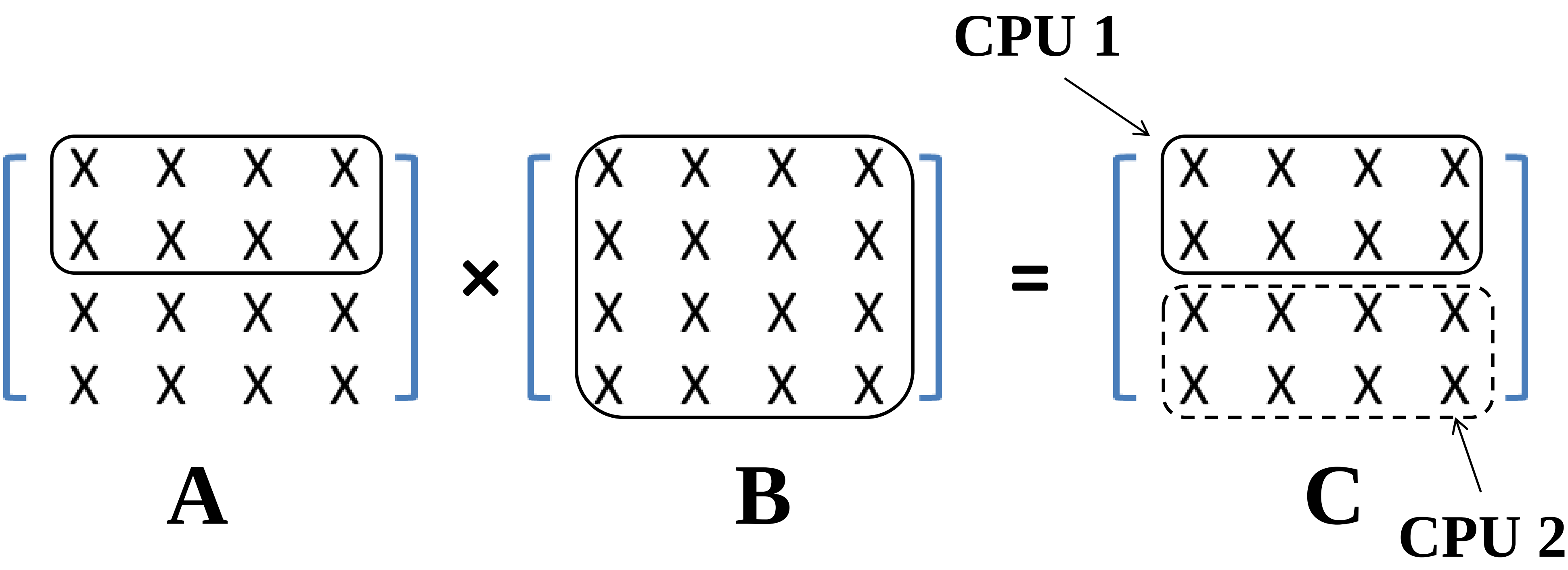
Assign many rows to each processor

Parallel Algorithm 3

Each processor computes N/P rows of C

Need entire B matrix and N/P rows of A as input

Each processor now needs $O(N^2)$ communication and $O(N^3 / P)$ computation



Parallel Algorithm 3 (master)

Master (processor 0):

```
int result [N, N / P];
int inc = N / P; /* number of rows per cpu */
int lb = 1;  /* lb = lower bound */
for (i = 1; i <= P; i++)
      SEND (i, A[lb .. lb+inc-1, *], B[*,*], lb, lb+inc-1);
      lb += inc;
for (x = 1; x <= P; x++)
      RECEIVE_FROM_ANY (&result, &lb);
      for (i = 1; i <= N / P; i++)
            C[lb+i-1, *] = result[i, *];
```

Parallel Algorithm 3 (slave)

Slaves:

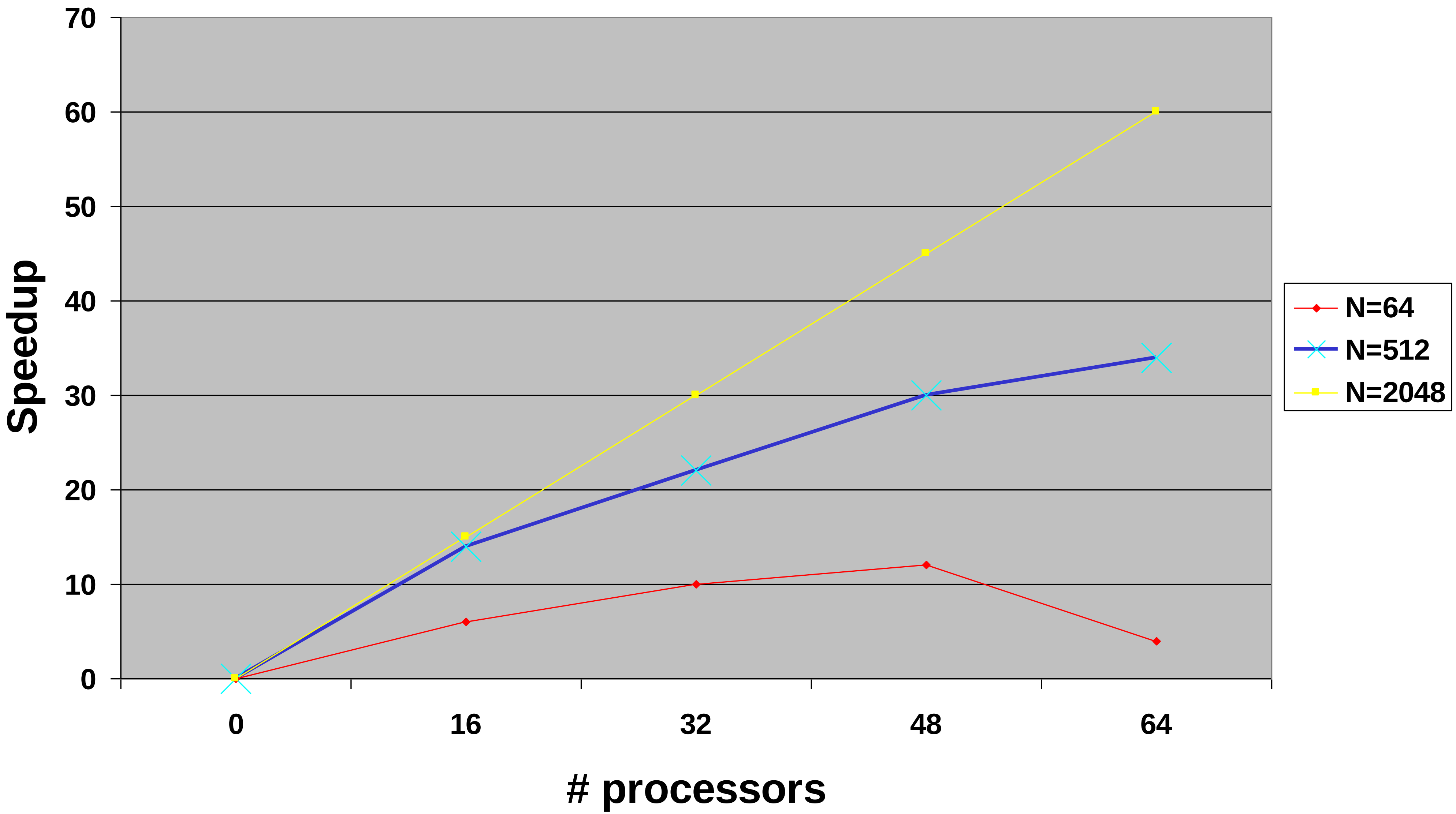
```
int A[N / P, N], B[N,N], C[N / P, N];
RECEIVE(0, &A, &B, &lb, &ub);
for (i = lb; i <= ub; i++)
    for (j = 1; j <= N; j++)
        C[i,j] = 0;
        for (k = 1; k <= N; k++)
            C[i,j] += A[i,k] * B[k,j];
SEND(0, C[*,*] , lb);
```

Comparison

Algori thm	Parallelism (#jobs)	Communication per job	Computation per job	Ratio comp/comm
1	N^2	$N + N + 1$	N	$O(1)$
2	N	$N + N^2 + N$	N^2	$O(1)$
3	P	$N^2/P + N^2 + N^2/P$	N^3/P	$O(N/P)$

- If N >> P, algorithm 3 will have low communication overhead
- Its grain size is high

Example speedup graph



Discussion

- Matrix multiplication is trivial to parallelize
- Getting good performance is a problem
- Need right grain size
- Need large input problem

Successive Over relaxation (SOR)

Iterative method for solving Laplace equations

Repeatedly updates elements of a grid

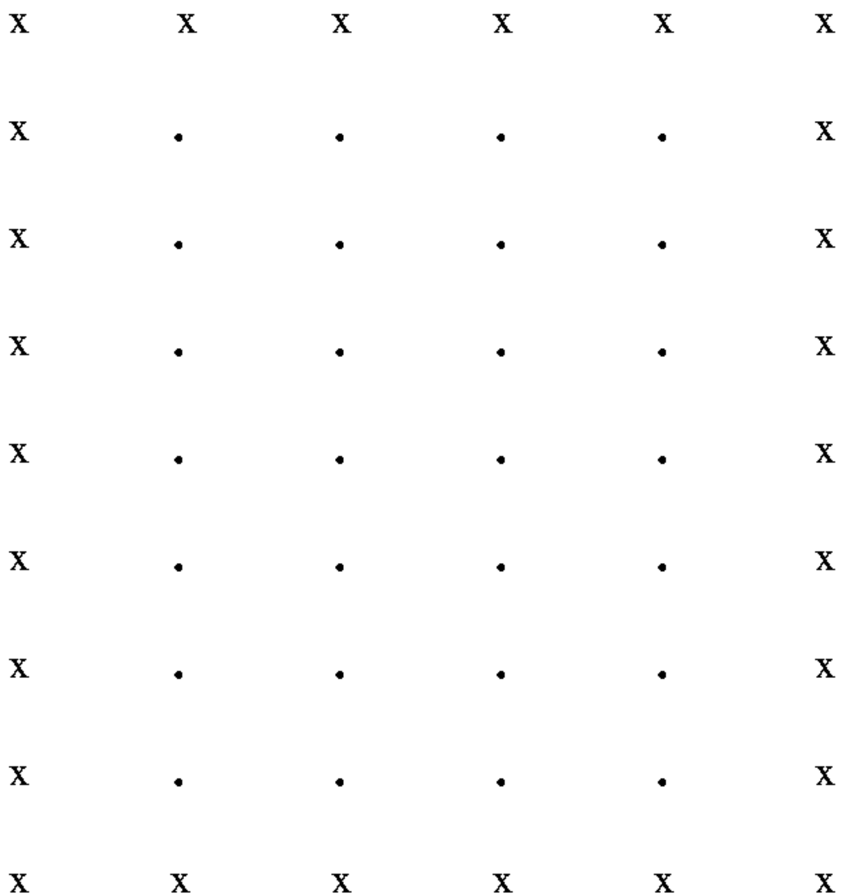
```
x   x   x   x   x   x
x   .   .   .   .   x
x   .   .   .   .   x
x   .   .   .   .   x
x   .   .   .   .   x
x   .   .   .   .   x
x   .   .   .   .   x
x   .   .   .   .   x
x   x   x   x   x   x
```

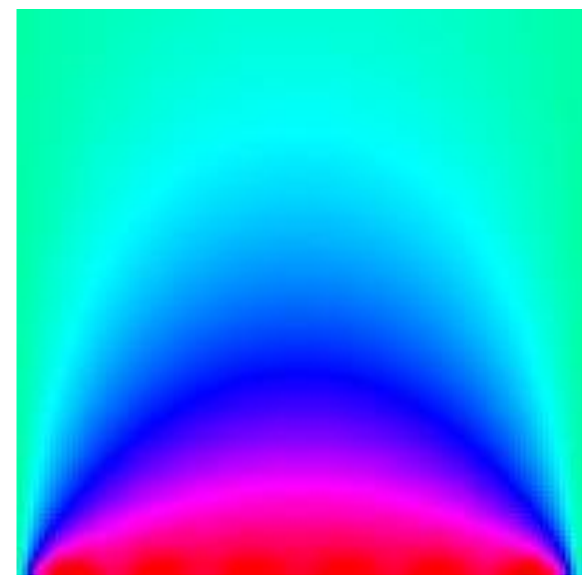
Successive Over relaxation (SOR)

```
float G[1:N, 1:M], Gnew[1:N, 1:M];
for (step = 0; step < NSTEPS; step++)
    for (i = 2; i < N; i++)                    /* update grid */
        for (j = 2; j < M; j++)
            Gnew[i,j] = f(G[i,j], G[i-1,j], G[i+1,j],G[i,j-1], G[i,j+1]);
    G = Gnew;
```

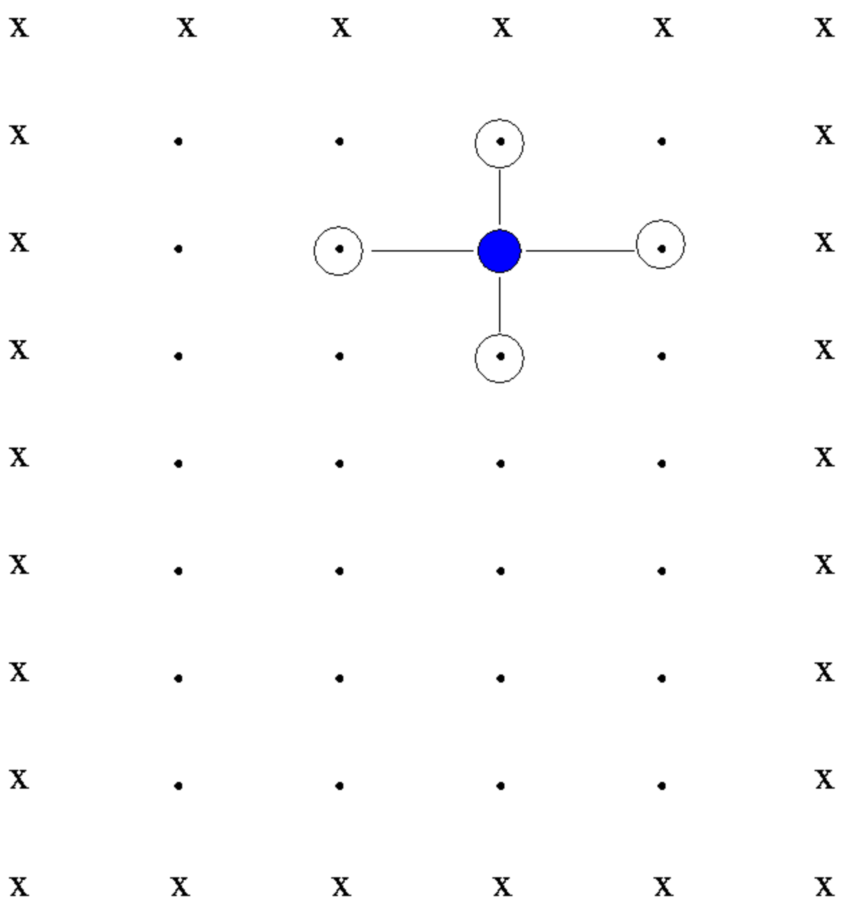
```
x   x   x   x   x   x
x   .   .   .   .   x
x   .   .   .   .   x
x   .   .   .   .   x
x   .   .   .   .   x
x   .   .   .   .   x
x   .   .   .   .   x
x   .   .   .   .   x
x   x   x   x   x   x
```

SOR example





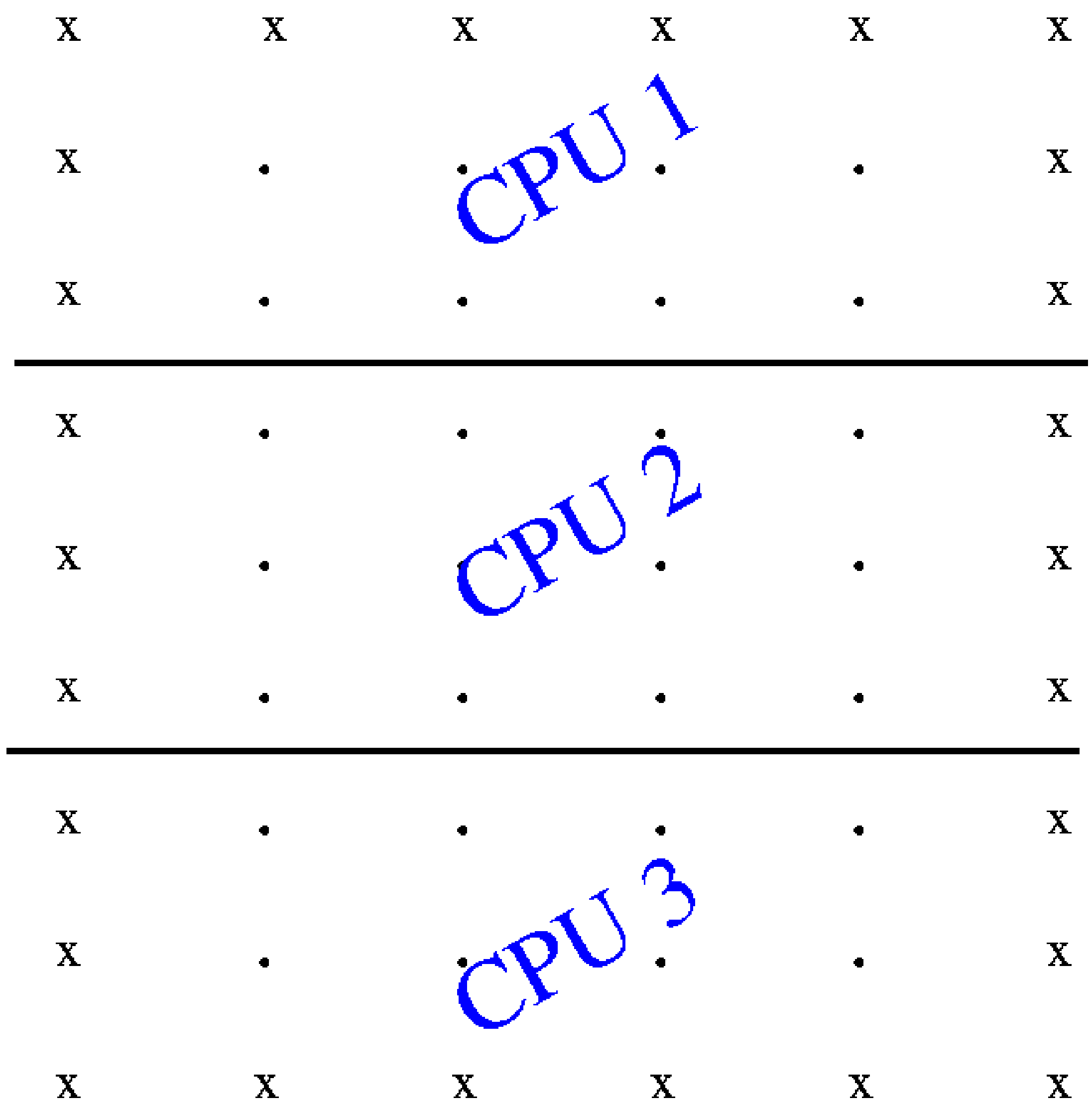
SOR example



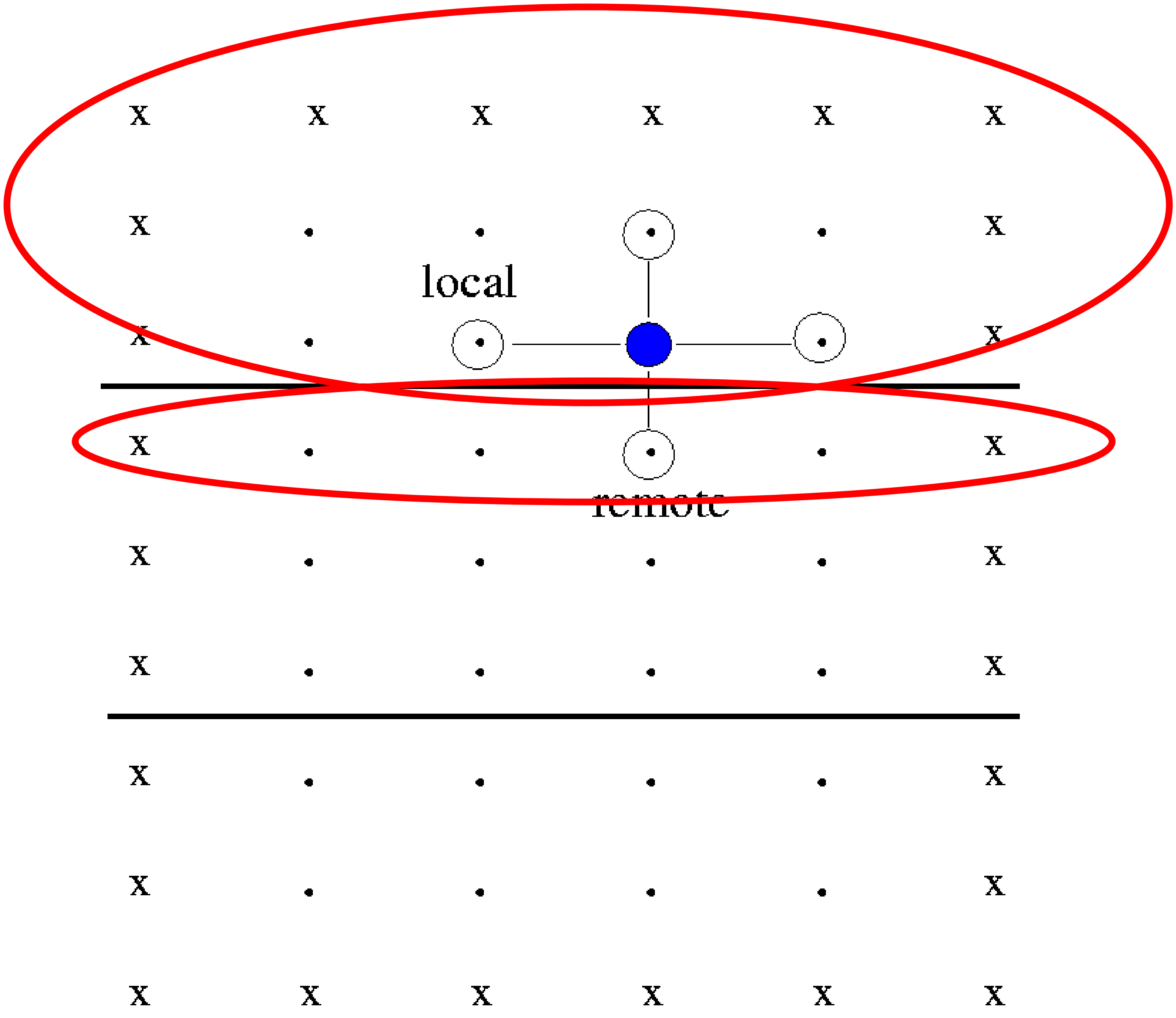
Parallelizing SOR

- Domain decomposition on the grid
- Each processor owns N/P rows
- Need communication between neighbors to exchange elements at processor boundaries

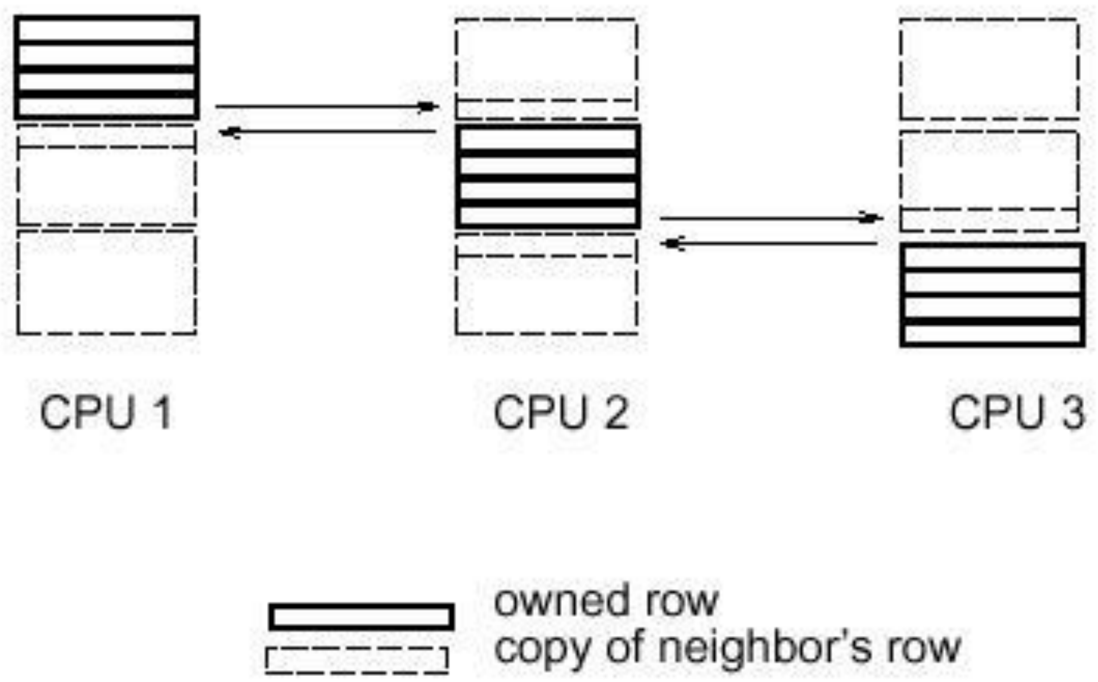
SOR example partitioning



SOR example partitioning



Communication scheme

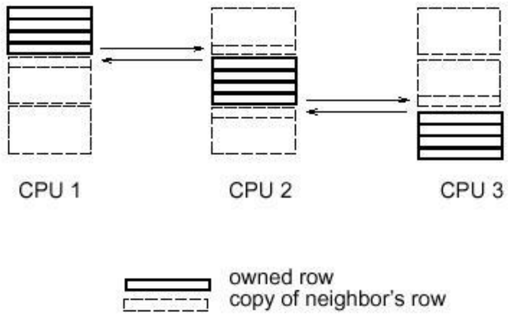


Each CPU communicates with left & right neighbor (if existing)

These elements are called <u>halo cells</u>

Parallel SOR

```
float G[lb-1:ub+1, 1:M], Gnew[lb-1:ub+1, 1:M];
for (step = 0; step < NSTEPS; step++)
    SEND(cpuid-1, G[lb]);              /* send 1st row left */
    SEND(cpuid+1, G[ub]);              /* send last row right */
    RECEIVE(cpuid-1, G[lb-1]);         /* receive from left */
    RECEIVE(cpuid+1, G[ub+1]);         /* receive from right */
    for (i = lb; i <= ub; i++)         /* update my rows */
        for (j = 2; j < M; j++)
            Gnew[i,j] = f(G[i,j], G[i-1,j], G[i+1,j], G[i,j-1], G[i,j+1]);
    G = Gnew;
```



Performance of SOR

Communication and computation during each iteration:

- Each CPU sends/receives 2 messages with M reals
- Each CPU computes N/P * M updates

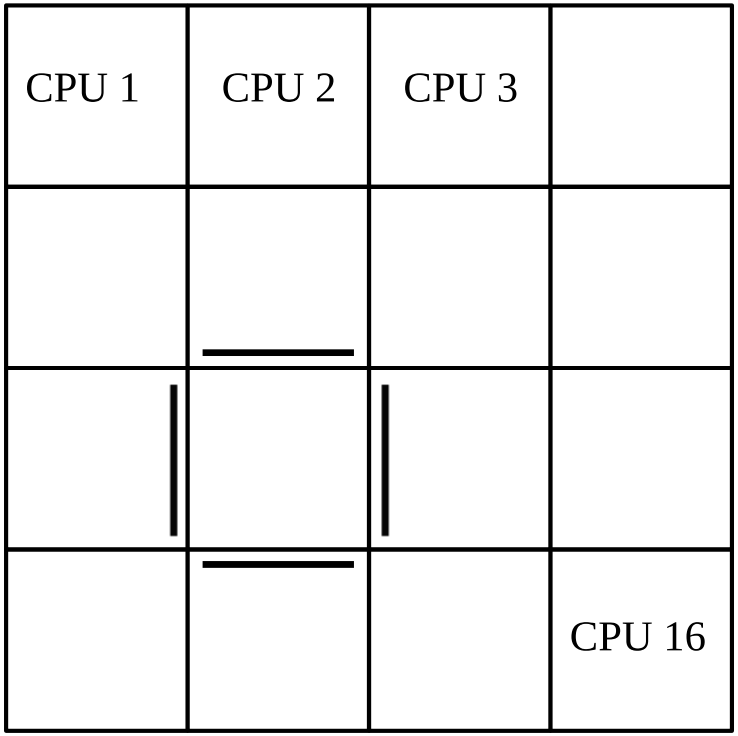
The algorithm will have good performance if

- Problem size is large: N >> P
- Message exchanges can be done in parallel

Question:

- Can we improve the performance of parallel SOR by using a different distribution of data?

Example: block-wise partitioning



- Each CPU gets a N/SQRT(P) by N/SQRT(P) block of data (assuming N=M)

- Each CPU needs sub-rows/columns from 4 neighbors
- Row-wise: only 2 messages, but with N elements
- Block-wise: 4 messages, with N/SQRT(P) elements
- Best partitioning depends on machine/network !
- More on this at HPF lecture

All-pairs Shorts Paths (ASP)

- Given a graph G with a distance table C:

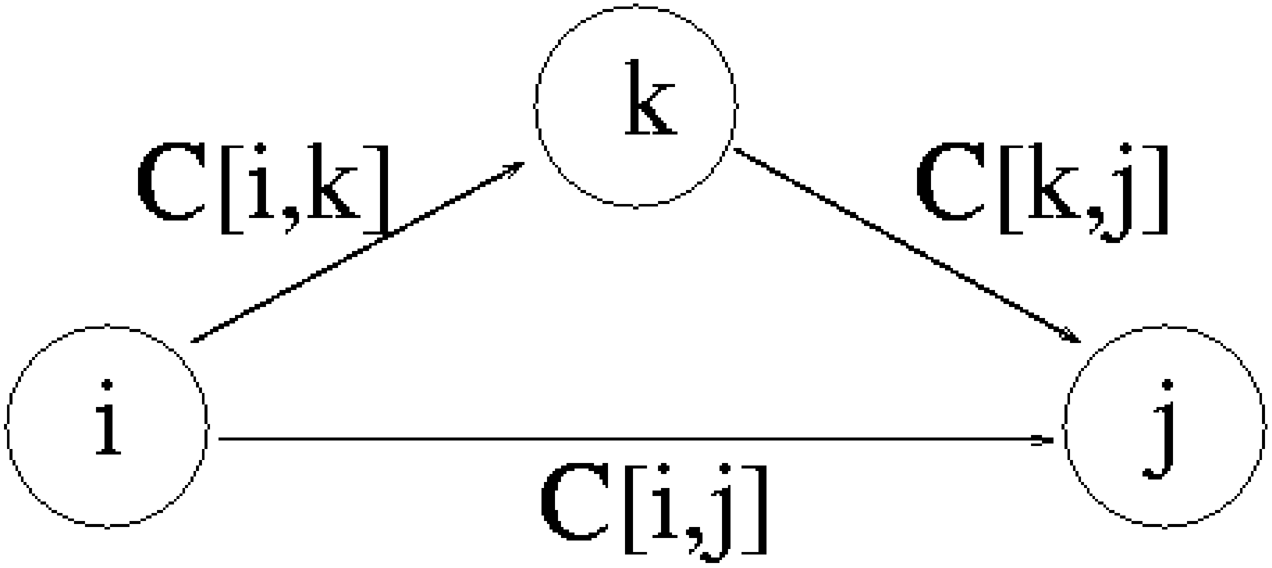
  C [i , j] = length of direct path from node i to node j

- Compute length of shortest path between any two nodes in G

	Amsterdam	Berlin	Copenhagen	London	Moscow	Rome	Warsaw
Amsterdam		365	381	220	1325	808	673
Berlin	365		225	575	995	730	320
Copenhagen	381	225		590	970	948	415
London	220	575	590		1540	890	890
Moscow	1325	995	970	1540		1462	710
Rome	808	730	948	890	1462		810
Warsaw	673	320	415	890	710	810	

Floyd's Sequential Algorithm

- Basic step:




```
for (k = 1; k <= N; k++)
   for (i = 1; i <= N; i++)
    for (j = 1; j <= N; j++)
        C [ i , j ] = MIN ( C [i, j],
.                     C [i ,k] +C [k, j]);
```

During iteration k, you can visit only intermediate nodes in the set {1 .. k}

k=0 => initial problem, no intermediate nodes

k=N => final solution

Parallelizing ASP

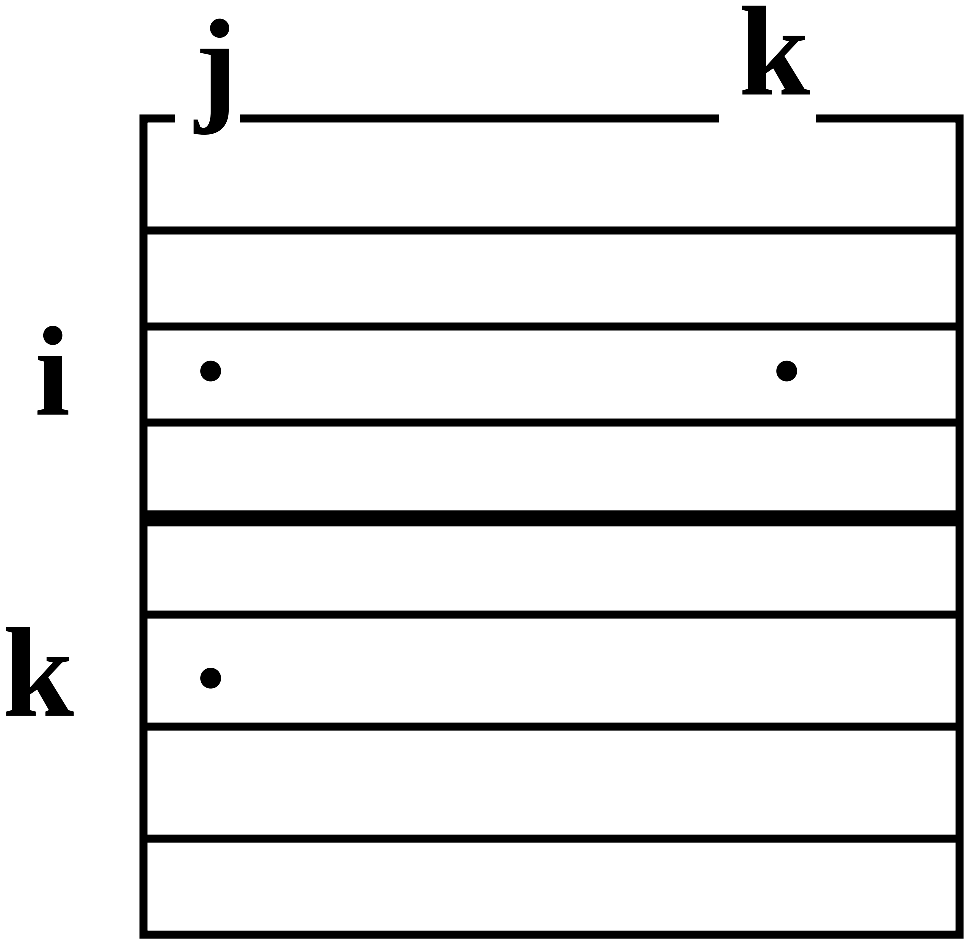
- Distribute rows of C among the P processors

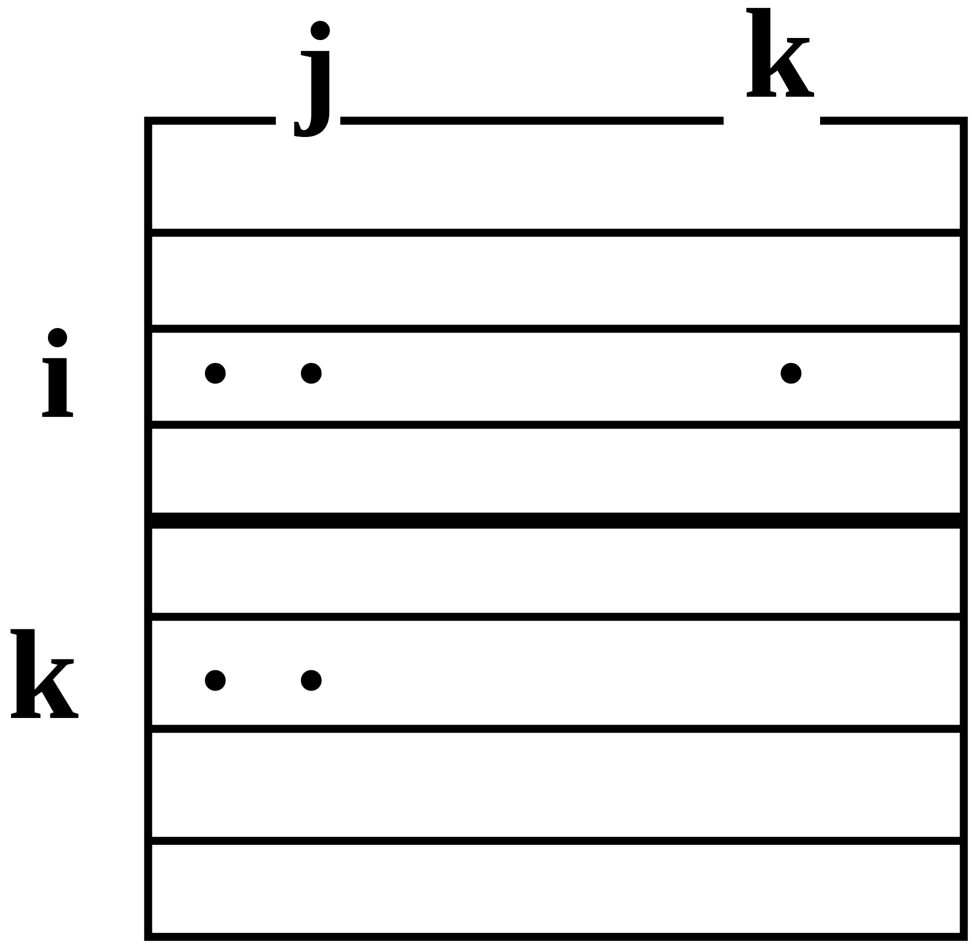
- During iteration *k*, each processor executes

  C [i,j] = MIN (C[i ,j], C[i,k] + C[k,j]);

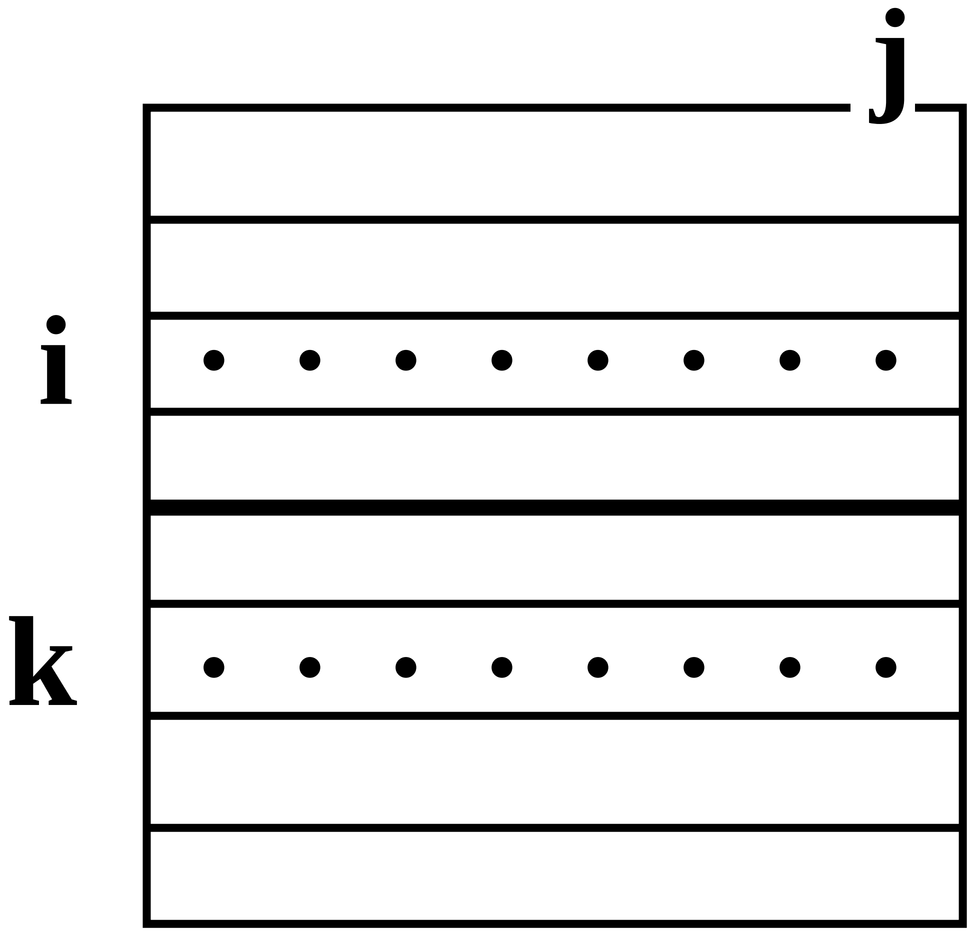
  on its own rows i, so it needs these rows and row *k*

- Before iteration k, the processor owning row *k* sends it to all the others


j
k
i
k



j
k
i
k


j
i
k

Parallel ASP Algorithm

```
int lb, ub;      /* lower/upper bound for this CPU */
int rowK[N], C[lb:ub, N];    /* pivot row ; matrix */

for (k = 1; k <= N; k++)
   if (k >= lb && k <= ub)   /* do I have it? */
        rowK = C[k,*];
        for (proc = 1; proc <= P; proc++)      /* broadcast row */
             if (proc != myprocid) SEND(proc, rowK);
   else
        RECEIVE_FROM_ANY(&rowK);       /* receive row */
   for (i = lb; i <= ub; i++)                  /* update my rows */
        for (j = 1; j <= N; j++)
             C[i,j] = MIN(C[i,j], C[i,k] + rowK[j]);
```

Performance Analysis ASP

Per iteration:

- 1 CPU sends P -1 messages with *N* integers
- Each CPU does N/P x *N* comparisons

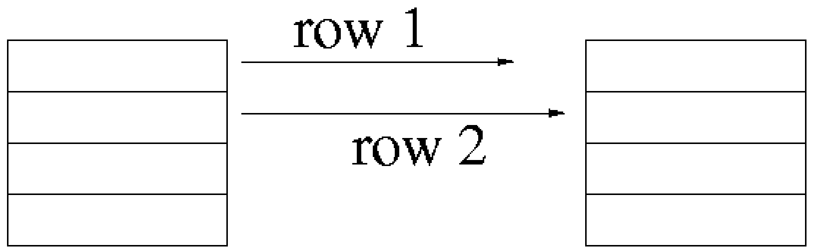
Communication/ computation ratio is small if *N* >> P

... but, is the Algorithm Correct?

Parallel ASP Algorithm

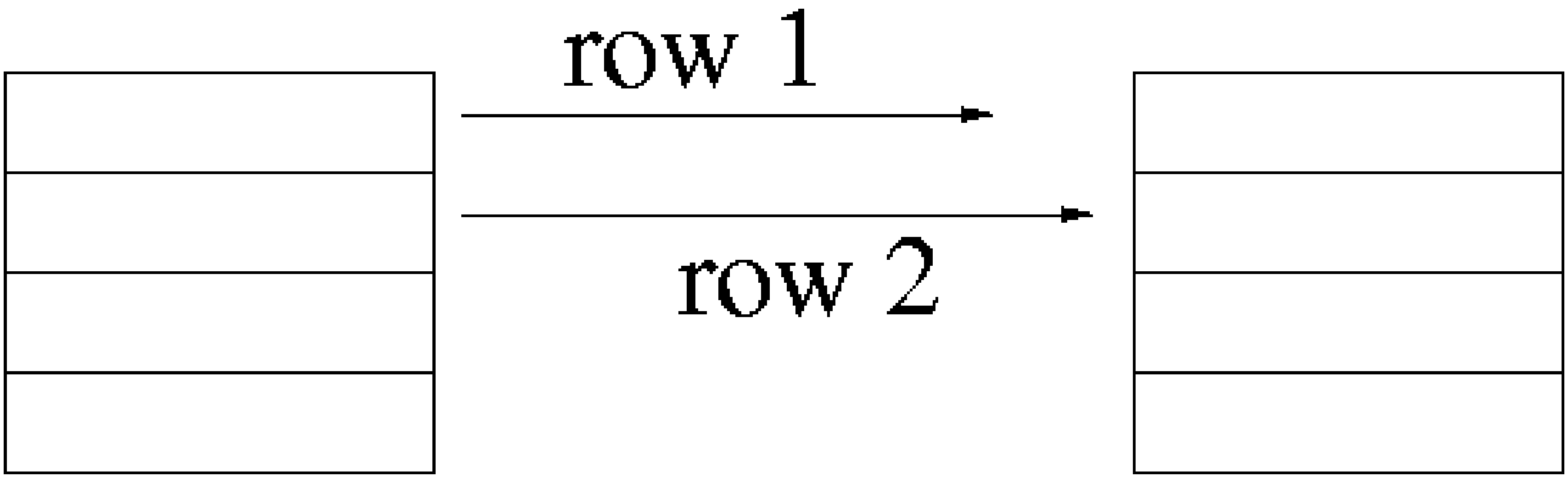
```
int lb, ub;       /* lower/upper bound for this CPU */
int rowK[N], C[lb:ub, N];     /* pivot row ; matrix */

for (k = 1; k <= N; k++)
    if (k >= lb && k <= ub)    /* do I have it? */
        rowK = C[k,*];
        for (proc = 1; proc <= P; proc++)       /* broadcast row */
            if (proc != myprocid) SEND(proc, rowK);
    else
        RECEIVE_FROM_ANY(&rowK);       /* receive row */
    for (i = lb; i <= ub; i++)                  /* update my rows */
        for (j = 1; j <= N; j++)
            C[i,j] = MIN(C[i,j], C[i,k] + rowK[j]);
```



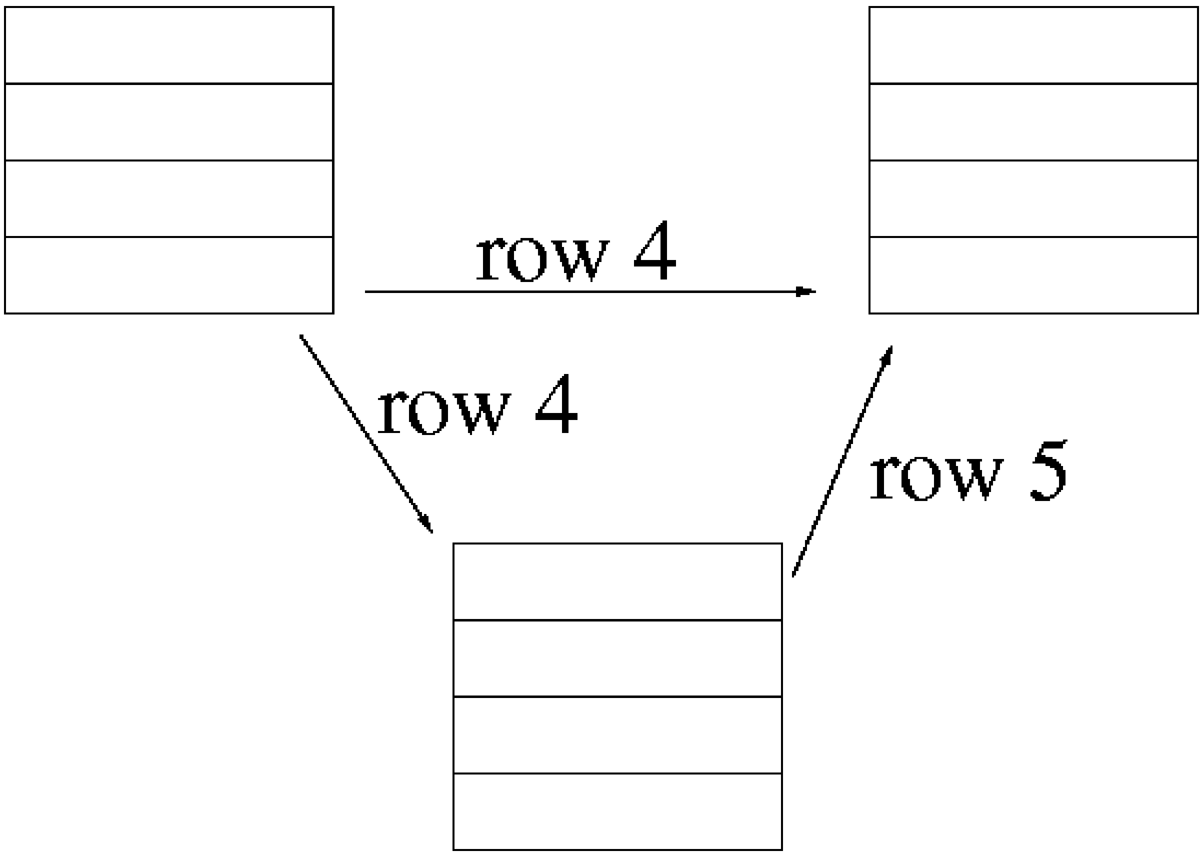
Non-FIFO Message Ordering

Row 2 may be received before row 1



FIFO Ordering

Row 5 may be received before row 4



Correctness

Problems:

- Asynchronous non-FIFO SEND
- Messages from different senders may overtake each other

Solution is to use a combination of:

- Synchronous SEND (less efficient)
- Barrier at the end of outer loop (extra communication)
- Order incoming messages (requires buffering)
- RECEIVE (cpu, msg) (more complicated)

Introduction

- Language notation: message passing
- Distributed-memory machine
  - (e.g., workstations on a network)

- 5 parallel algorithms of increasing complexity:
  - Matrix multiplication
  - Successive overrelaxation
  - All-pairs shortest paths
  - Linear equations
  - Traveling Salesman problem

Linear equations

- Linear equations:

  $a_{1,1}x_1 + a_{1,2}x_2 + \ldots a_{1,n}x_n = b_1$

  ...

  $a_{n,1}x_1 + a_{n,2}x_2 + \ldots a_{n,n}x_n = b_n$

- Matrix notation: $Ax = b$
- Problem: compute x, given A and b
- Linear equations have many important applications

  Practical applications need huge sets of equations

Solving a linear equation

- Two phases:

  Upper-triangularization -> U x = y

  Back-substitution -> x

- Most computation time is in upper-triangularization
- Upper-triangular matrix:

  U [i, i] = 1

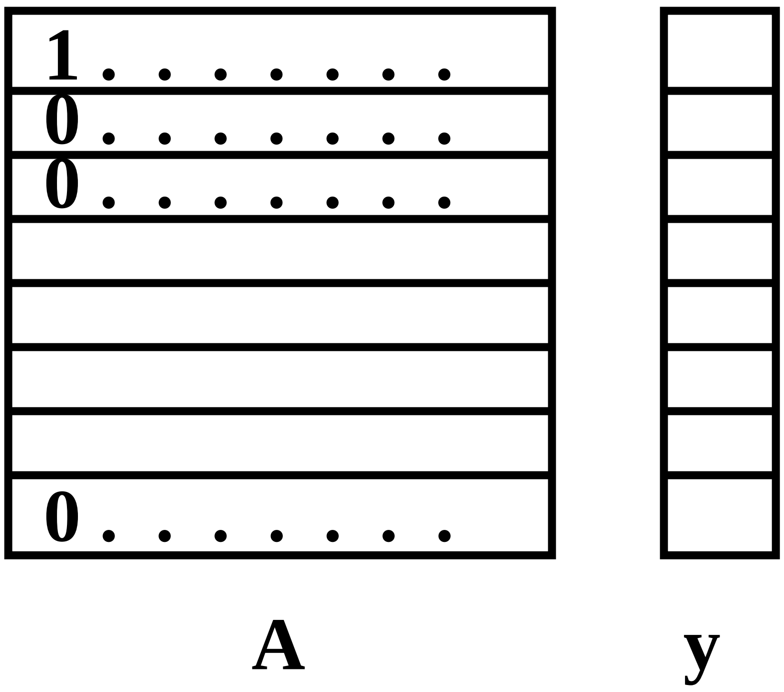
  U [i, j] = 0 if $i > j$

1 . . . .
0 1 . . . . . .
0 0 1 . . . . .
0 0 0 1 . . . .
0 0 0 0 1 . . .
0 0 0 0 0 1 . .
0 0 0 0 0 0 1 .
0 0 0 0 0 0 0 1

Sequential Gaussian elimination

```
for (k = 1; k <= N; k++)
    for (j = k+1; j <= N; j++)
        A[k,j] = A[k,j] / A[k,k]
    y[k] = b[k] / A[k,k]
    A[k,k] = 1
    for (i = k+1; i <= N; i++)
        for (j = k+1; j <= N; j++)
            A[i,j] = A[i,j] - A[i,k] * A[k,j]
        b[i] = b[i] - A[i,k] * y[k]
        A[i,k] = 0
```

- Converts Ax = b into Ux = y
- Sequential algorithm uses 2/3 N^3 operations



Parallelizing Gaussian elimination

- Block-wise partitioning scheme

  Each cpu gets a number of consecutive rows

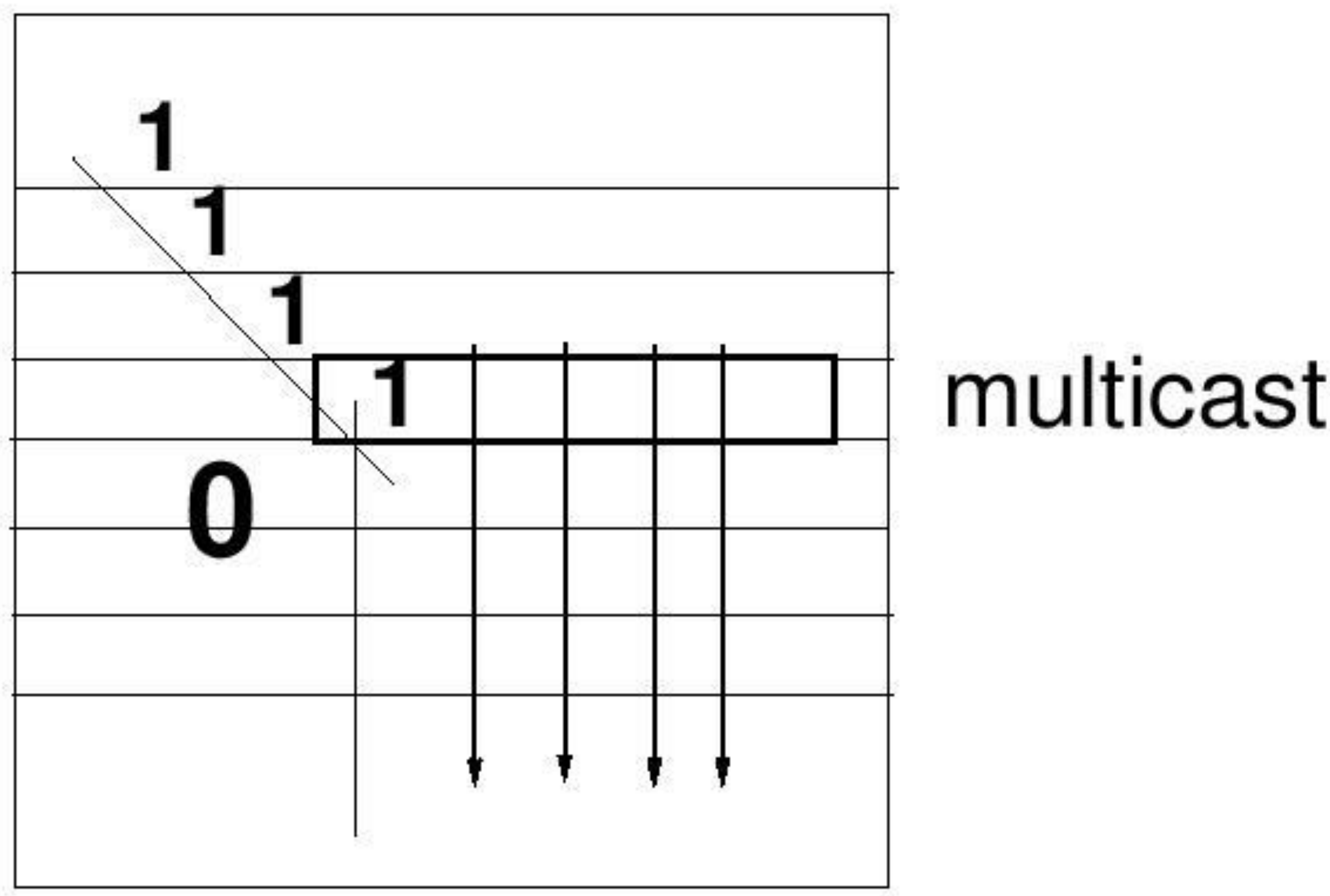
  Execute one (outer-loop) iteration at a time

- Communication requirement:

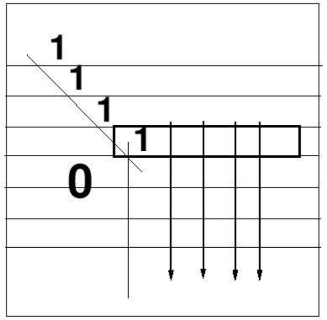
  During iteration k, cpus containing rows k+1 .. N need part of row *k*

  -> need partial broadcast (multicast)

Communication



Performance problems



multicast

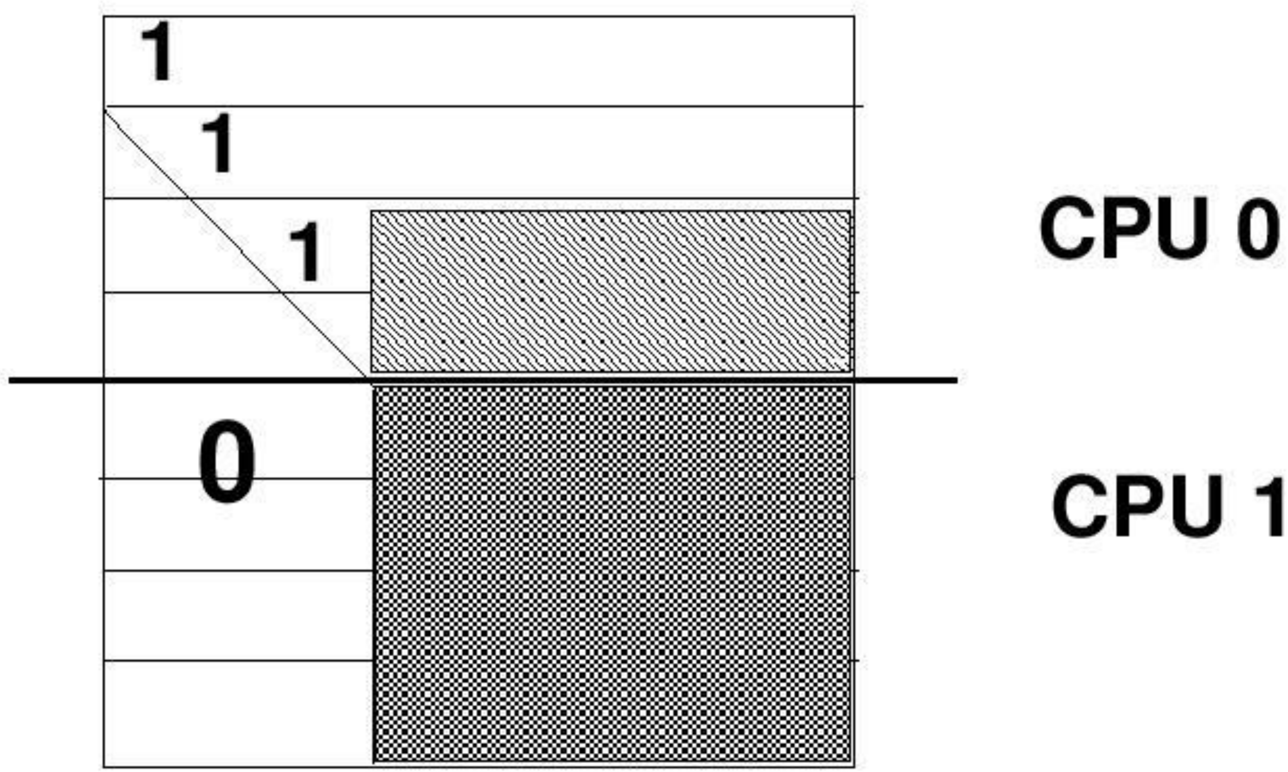
- Communication overhead (multicast)
- Load imbalance

  CPUs with rows <k are idle during iteration *k*

  Bad load balance means bad speedups, as some CPUs have *too much* work

- Block-wise distribution thus has high load-imbalance
- Alternative:
  - <u>Cyclic</u> distribution of rows
  - Has less load-imbalance

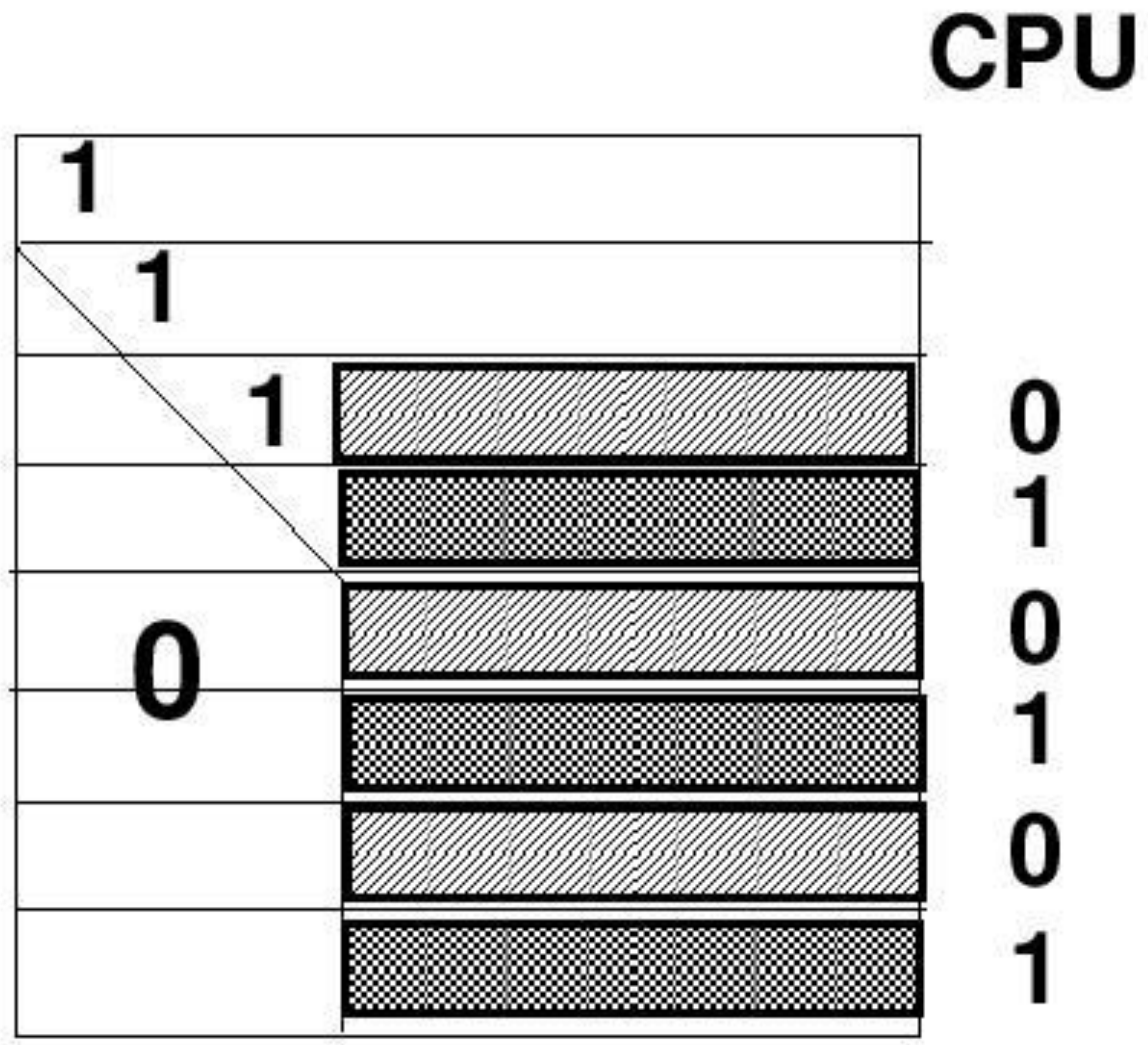
Block-wise distribution



- CPU 0 gets first N/2 rows
- CPU 1 gets last N/2 rows

- CPU 0 has much less work to do
- CPU 1 becomes the bottleneck

Cyclic distribution

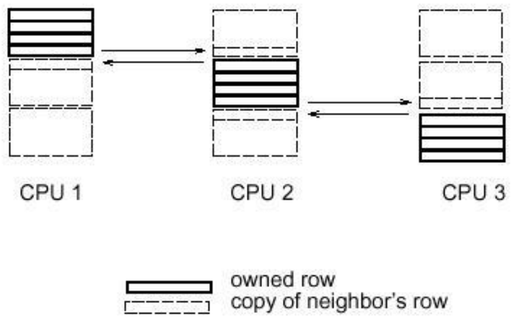


- CPU 0 gets odd rows
- CPU 1 gets even rows

- CPU 0 and 1 have more or less the same amount of work

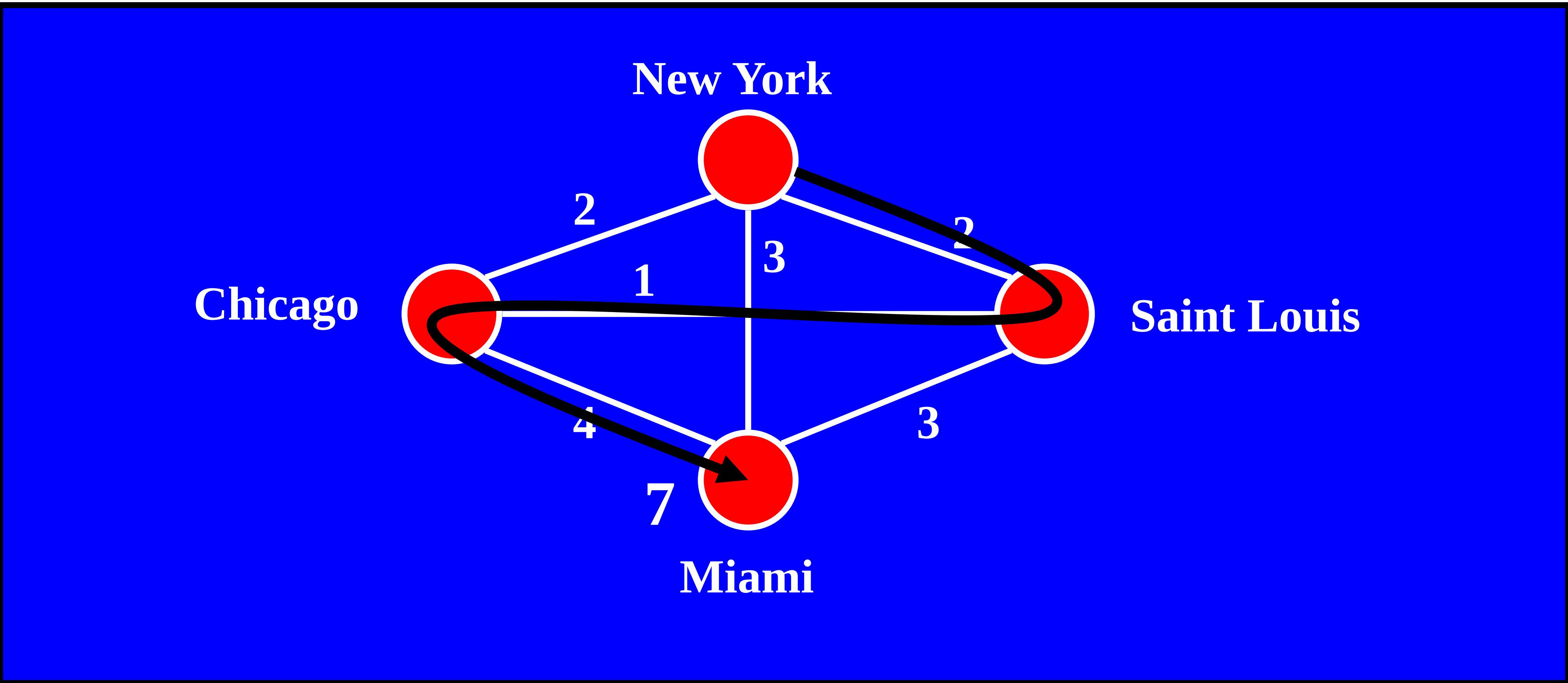
Cyclic distributions

- Useful for algorithms with predictable load imbalance
  - Form of *static* load balancing
- Not suitable for all communication patterns
  - SOR (nearest-neighbor communication) would suffer
  - Every neighboring row would be on a remote machine
  - Trade off: minimize communication + load imbalance overhead



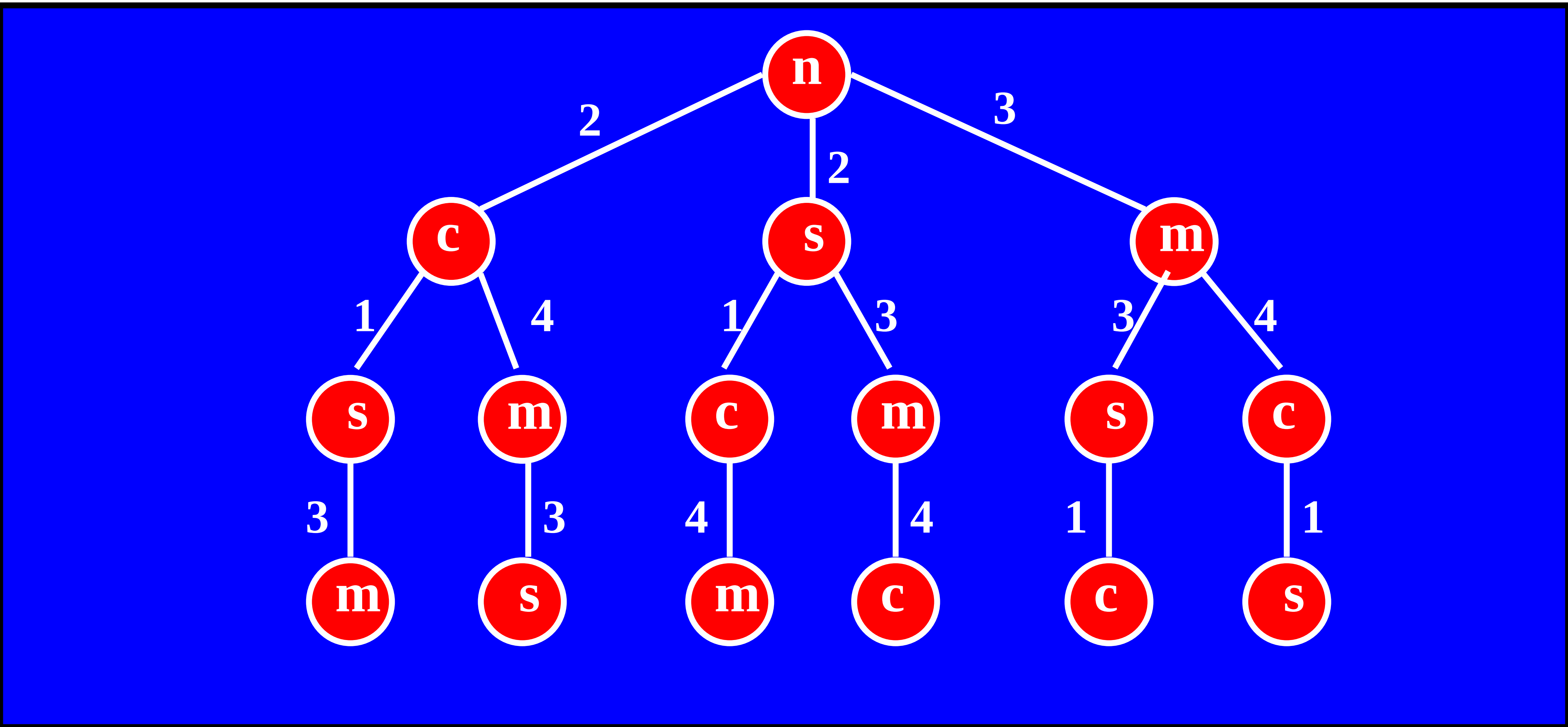
Traveling Salesman Problem (TSP)

- Find shortest route for salesman among given set of cities (NP-hard problem)
- Each city must be visited once, no return to initial city



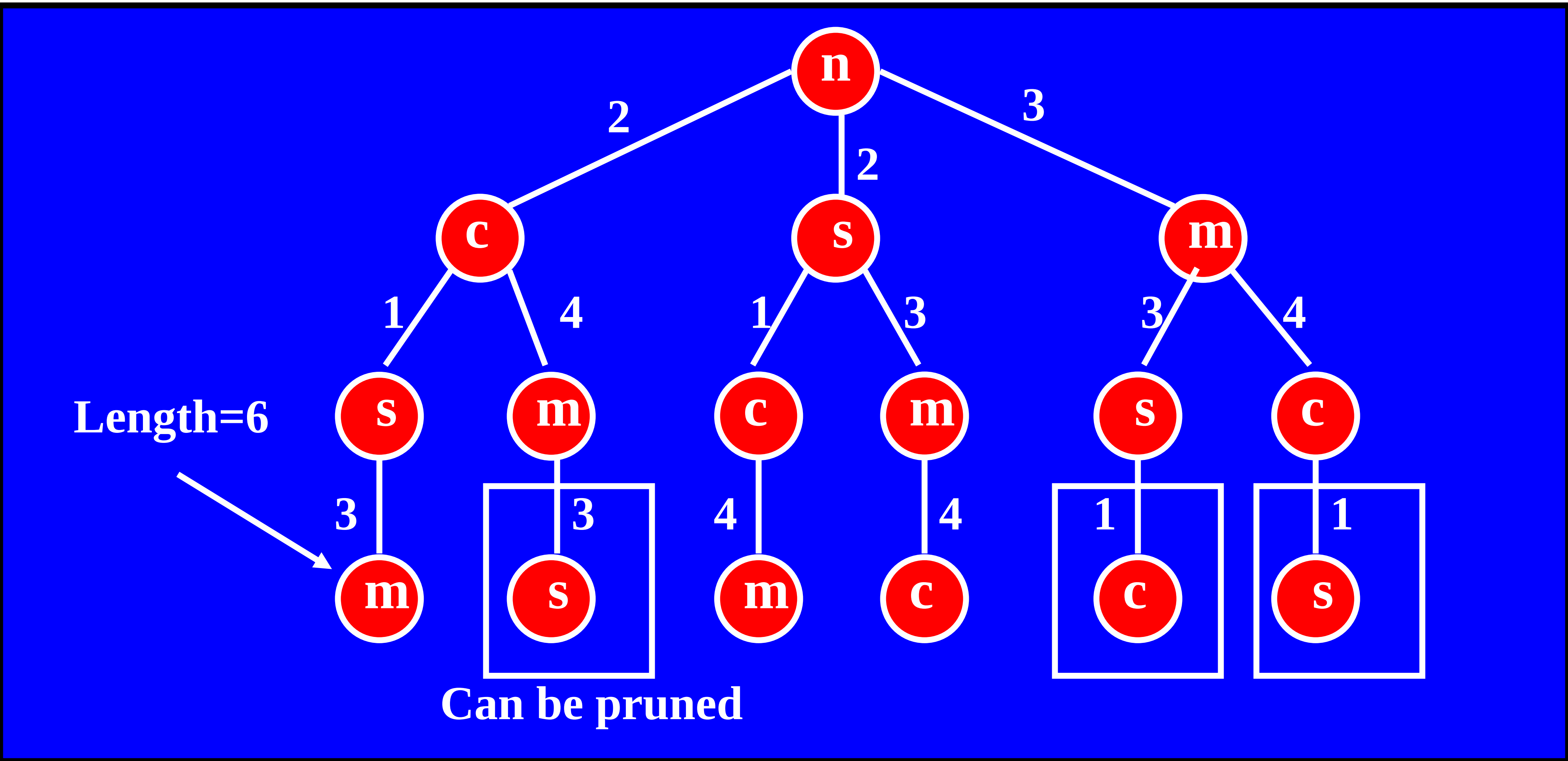
Sequential branch-and-bound

- Structure the entire search space as a tree, sorted using nearest-city first heuristic



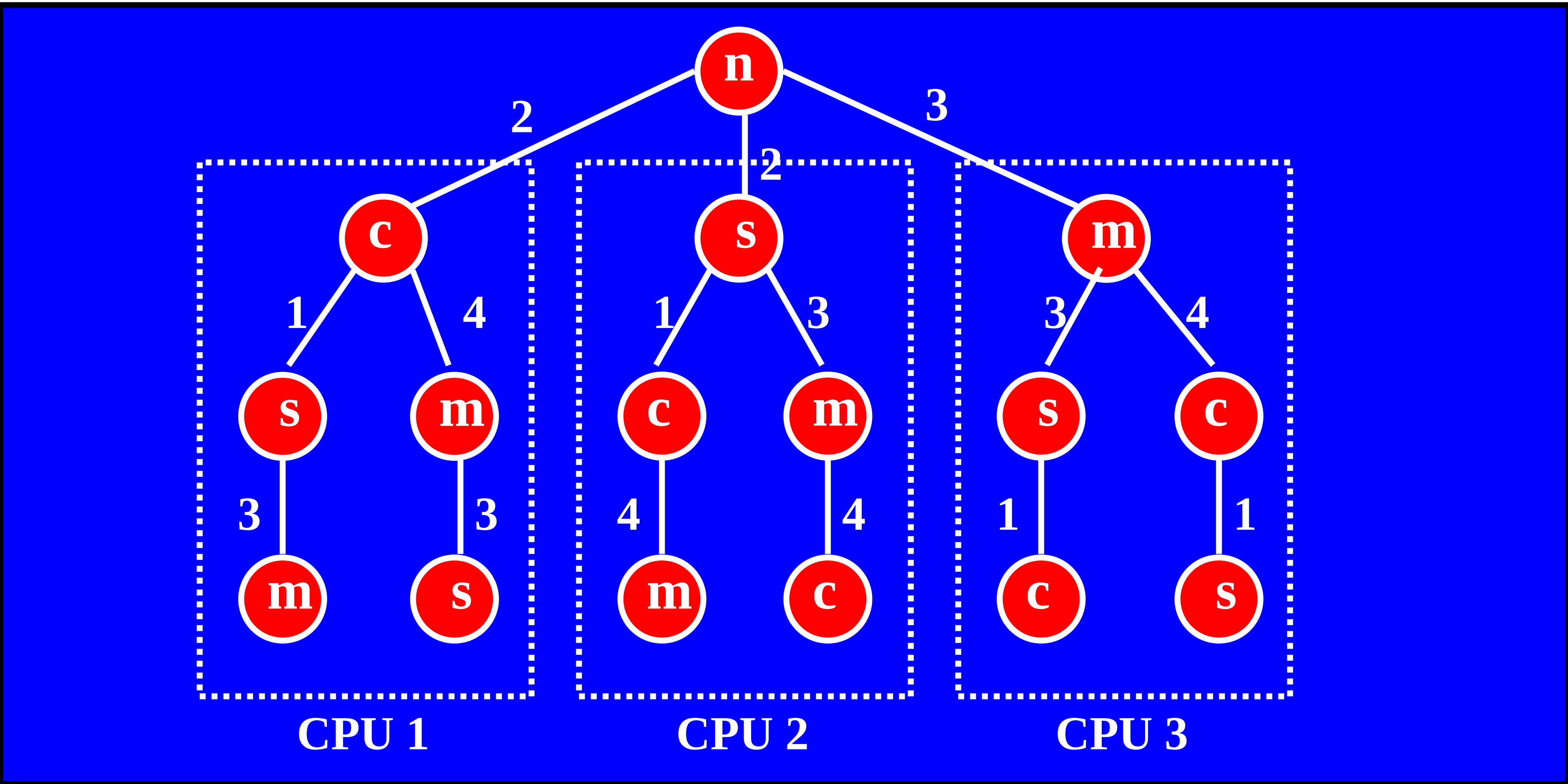
Pruning the search tree

- Keep track of best solution found so far (the “bound”)
- Cut-off partial routes >= bound



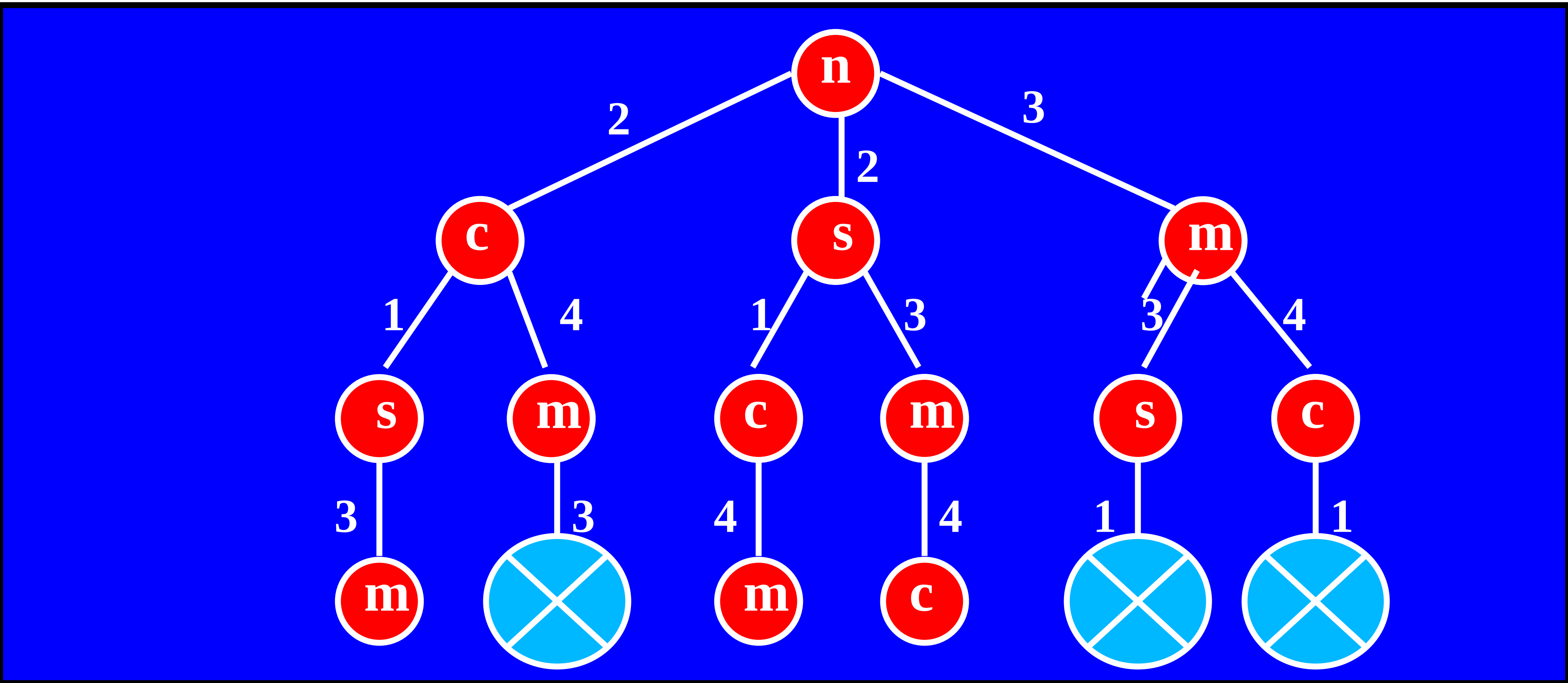
Parallelizing TSP

- Distribute the search tree over the CPUs
- Results in reasonably large-grain jobs



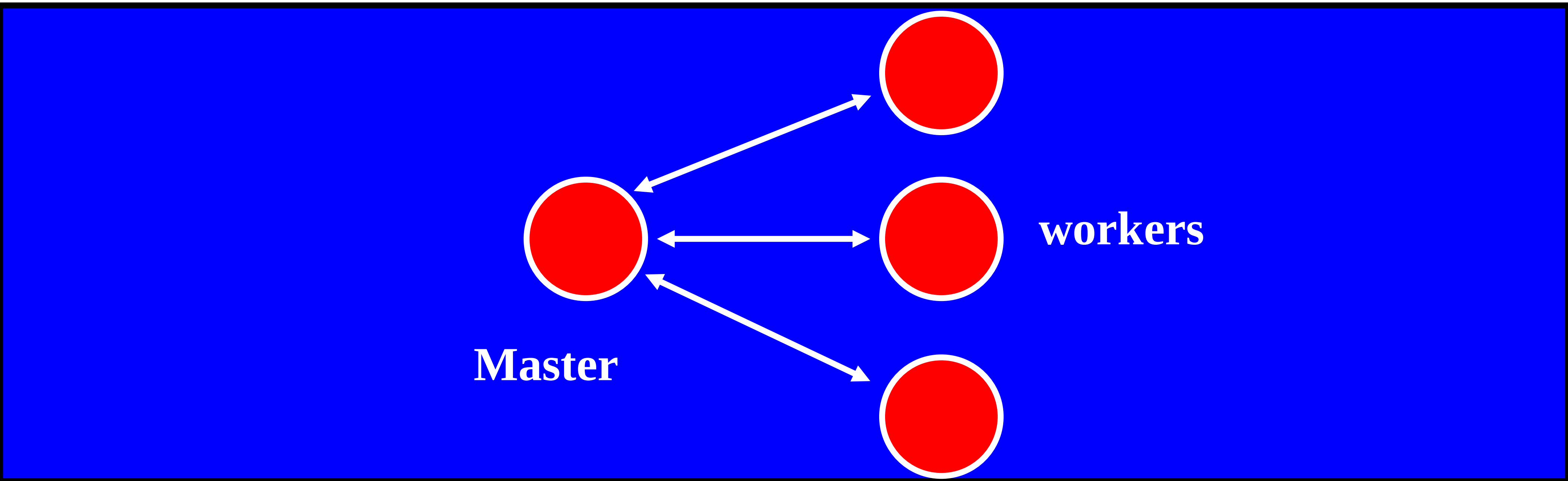
Distribution of the tree

- Static distribution: each CPU gets fixed part of tree
  - Load imbalance: subtrees take different amounts of time
  - Impossible to predict load imbalance statically (as for Gaussian)



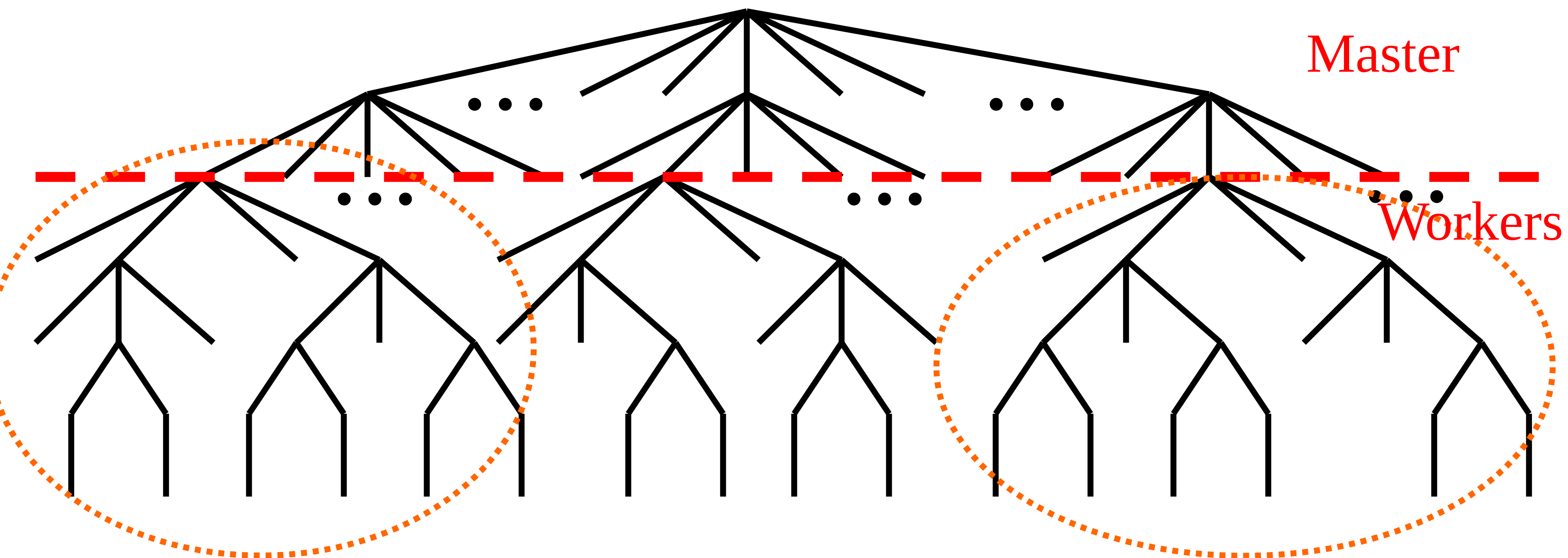
Dynamic load balancing: Replicated Workers Model

- Master process generates large number of jobs (subtrees) and repeatedly hands them out
- Worker processes repeatedly get work and execute it
- Runtime overhead for fetching jobs dynamically
- Efficient for TSP because the jobs are large



Real search spaces are huge

- NP-complete problem -> exponential search space
- Master searches MAXHOPS levels, then creates jobs
  - Eg for 20 cities & MAXHOPS=4 -> 20*19*18*17 (>100,000) jobs, each searching 16 remaining cities
- Few jobs: load imbalance; many jobs: communication



Parallel TSP Algorithm (1/3)

process master (CPU 0):

```
 generate-jobs([]);   /* generate all jobs, start with empty path  */
 for (proc=1; proc <= P; proc++)  /* inform workers we're done */
   RECEIVE(proc, &worker-id);  /* get work request */
   SEND(proc, []);                        /* return empty path */

generate-jobs (List path) {
 if (size(path) == MAXHOPS)    /* if path has MAXHOPS cities … */
   RECEIVE-FROM-ANY (&worker-id); /* wait for work request */
   SEND (worker-id, path);           /* send partial route to worker */
 else
   for (city = 1; city <= NRCITIES; city++)  /* (should be ordered) */
      if (city not on path) generate-jobs(path||city)  /* append city */
}
```

Parallel TSP Algorithm (2/3)

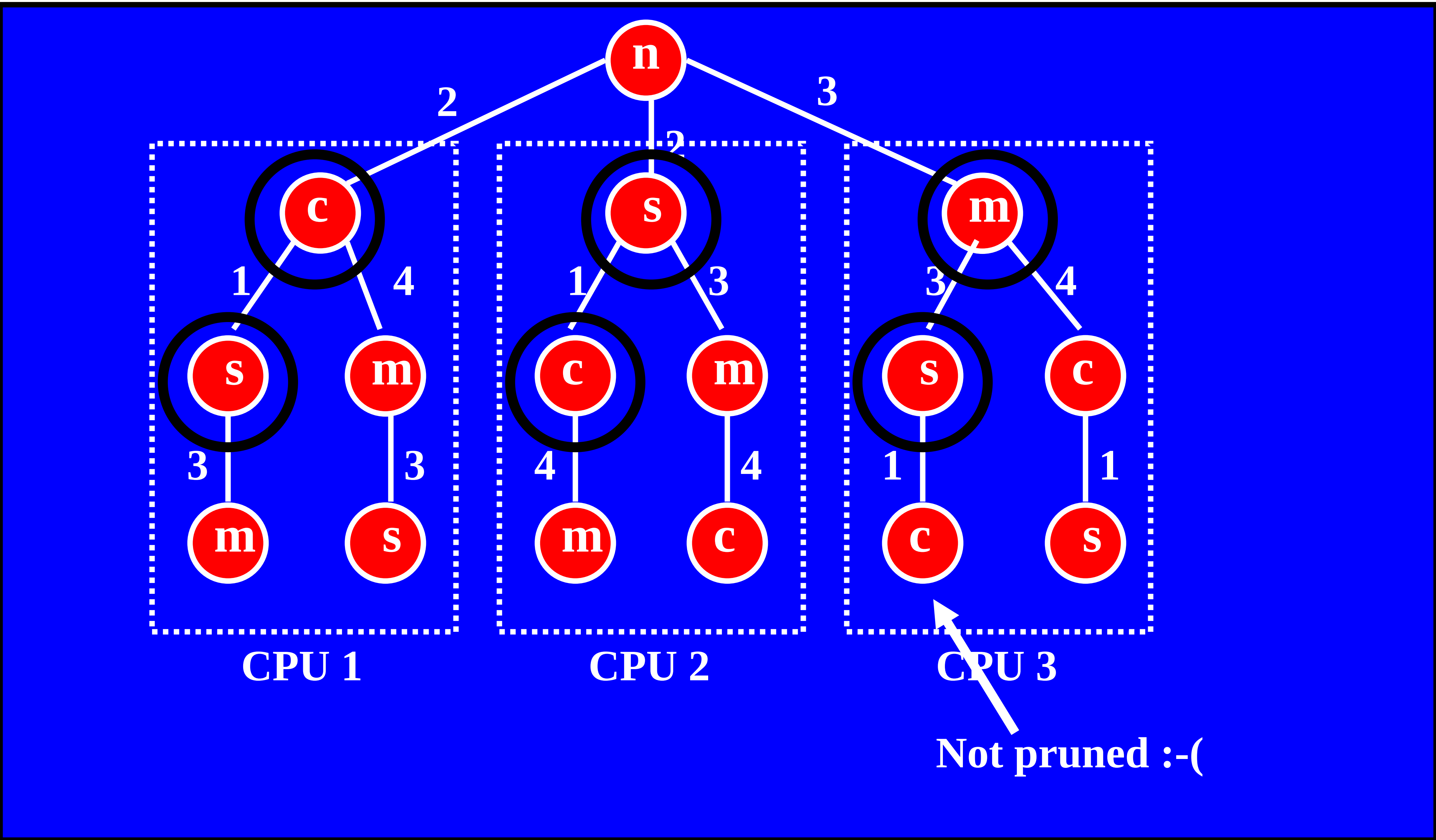
process worker (CPUs 1..P):

```
int Minimum = maxint; /* Length of current best path (bound) */
List path;
for (;;)
  SEND (0, myprocid)      /* send work request to master */
  RECEIVE (0, path);      /* get next job from master */
  if (path == []) exit(); /* we're done */
  tsp(path, length(path)); /* compute all subsequent paths */
```

Parallel TSP Algorithm (3/3)

```
tsp(List path, int length) {
   if (NONBLOCKING_RECEIVE_FROM_ANY (&m))
     /* is there an update message? */
     if (m < Minimum) Minimum = m;  /* update global minimum */
  if (length >= Minimum) return  /* not a shorter route */
  if (size(path) == NRCITIES)  /* complete route? */
     Minimum  = length;  /* update global minimum */
     for (proc = 1; proc <= P; proc++)
           if (proc != myprocid) SEND(proc, length) /* broadcast it */
  else
    last = last(path)  /* last city on the path */
    for (city = 1; city <= NRCITIES; city++)  /* should be ordered */
       if (city not on path) tsp(path||city, length+distance[last,city])
}
```

Search overhead



Search overhead

- Path <n – m – s > is started (in parallel) before the outcome (6) of <n – c – s – m> is known, so it cannot be pruned
- The parallel algorithm therefore does more work than the sequential algorithm
- This is called *search overhead*
- It can occur in algorithms that do speculative work, like parallel search algorithms
- Can also have negative search overhead, resulting in superlinear speedups!

Performance of TSP

- Communication overhead (small)
  - Distribution of jobs + updating the global bound
  - Small number of messages
- Load imbalances
  - Small: does automatic (dynamic) load balancing
- Search overhead
  - Main performance problem

Discussion

Several kinds of performance overhead

- Communication overhead:
  - communication/computation ratio must be low
- Load imbalance:
  - all processors must do same amount of work
- Search overhead:
  - avoid useless (speculative) computations

Making algorithms correct is nontrivial

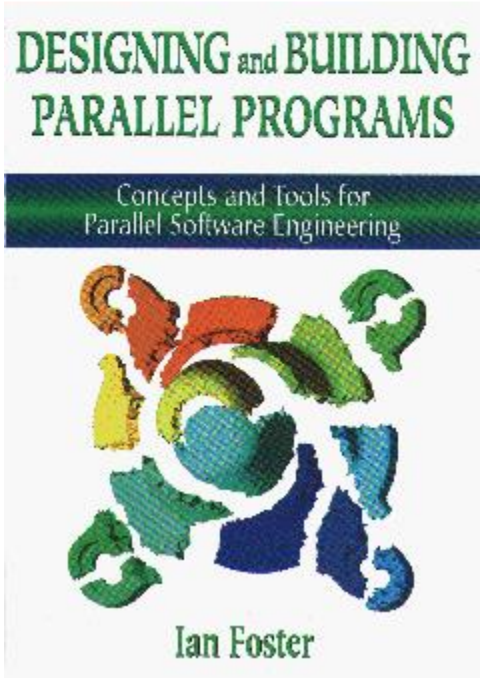
- Message ordering

Designing Parallel Algorithms

Source: Designing and building parallel programs (Ian Foster, 1995)

(available on-line at http://www.mcs.anl.gov/dbpp)

- Partitioning
- Communication
- Agglomeration
- Mapping



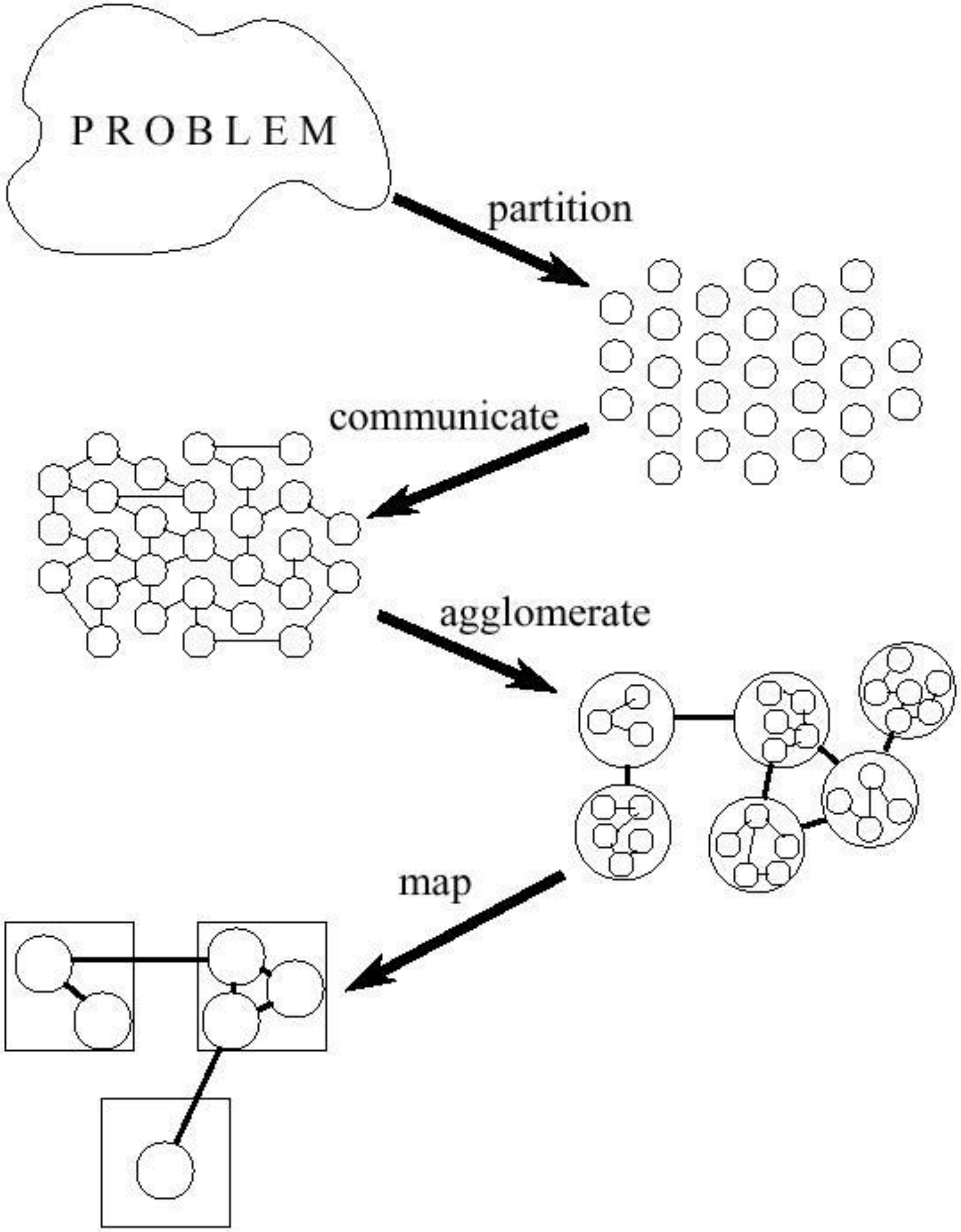


Figure 2.1 from Foster's book

Partitioning

- Domain decomposition

  Partition the data

  Partition computations on data:

  owner-computes rule
- Functional decomposition

  Divide computations into subtasks (e.g. search algorithms)

  Multi-model computations (climate simulations that model atmosphere, land, ice, ocean)

Also called *data-parallelism* versus *task-parallelism*

Data-parallel: same computations on different data

Task-parallel: different functions per machine

Communication

- Analyze data-dependencies between partitions
- Use communication to transfer data
- Many forms of communication, e.g.
  - Local communication with neighbors (SOR)
  - Global communication with all processors (ASP)
  - Synchronous (blocking) communication
  - Asynchronous (non blocking) communication

Agglomeration

- Reduce communication overhead by
  - increasing granularity
  - improving locality

Mapping

- On which processor to execute each subtask?
- Put concurrent tasks on different CPUs
- Put frequently communicating tasks on same CPU?
- Avoid load imbalances

Summary

Hardware and software models

Example applications

- Matrix multiplication - Trivial parallelism (independent tasks)
- Successive over relaxation - Neighbor communication
- All-pairs shortest paths - Broadcast communication
- Linear equations - Load balancing problem
- Traveling Salesman problem - Search overhead

Designing parallel algorithms