北京大学高能效计算与应用中心
Center for Energy-efficient Computing and Applications

Introduction to Parallel & Distributed Computing

Programming with MPI

Scalability & Mixed MPI+OpenMP

Lecture 10, Spring 2014

Instructor: 罗国杰

gluo@pku.edu.cn

Speedup & Efficiency

- **Execution time**

$$T(n,p) \geq \sigma(n) + \varphi(n)/p + \kappa(n,p)$$

serial + parallel + communication

- **Speedup**

$$S(n,p) = T(n,1) / T(n,p)$$

$$\leq (\sigma(n) + \varphi(n)) / (\sigma(n) + \varphi(n)/p + \kappa(n,p))$$

- **Cost (processor-time product)**

$$Cost(n,p) = p * T(n,p)$$

- **Efficiency**

$$E(n,p) = Cost(n,1) / Cost(n,p)$$

Scalability

- **Q: What is a scalable parallel system?**
- **A: Ideally, a scalable parallel system can solve a larger problem with more processors with**
  - **at least the same amount of efficiency**
  - **a bounded amount of memory per processor**

Performance Metrics for Parallel Systems

- **Parallel system**
  - **Parallel program executing on a parallel computer**
- **Execution time**

$$T(n,p) \geq \sigma(n) + \varphi(n)/p + \kappa(n,p)$$

serial + parallel + communication

- **Total parallel overhead**

$$T_O(n,p) = Cost(n,p) - Cost(n,1) = pT(n,p) - T(n,1) \geq (p-1)\sigma(n) + \kappa(n,p)$$

- **Efficiency**

$$E(n,p) = Cost(n,1) / Cost(n,p) = 1 / [1+T_O(n,p)/T(n,1)]$$

Scaling Characteristics of Parallel Systems

- **Efficiency:** $E(n,p) = 1 / [1+T_O(n,p)/T(n,1)]$
- **For a given problem size, as we increase the number of processors, $T_O(n,p)$ increases**

  $T_O(n,p) \geq (p-1)\sigma(n) + \kappa(n,p)$

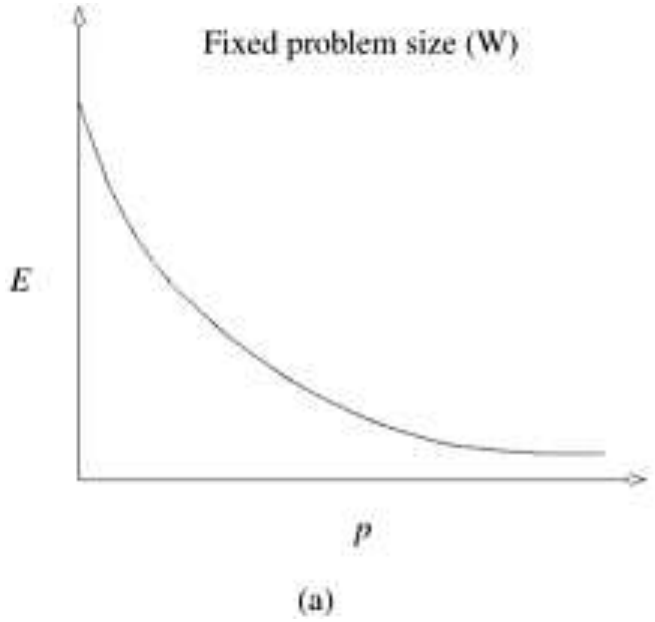
  => the efficiency goes down if p is increased with a fixed n
- **In many cases, $T_O(n,p)$ grows *sublinearly* with respect to $T(n,1)$**

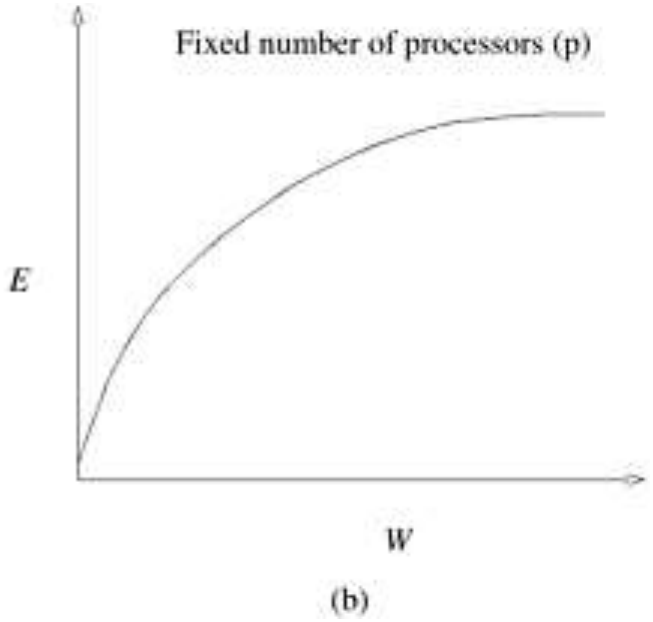
  for a given $c>0$ and a large enough n, $T_O(n,p) < cT(n,1)$

  => In such cases, the efficiency goes up if n is increased with a fixed p
- **For such systems, we can simultaneously increase the problem size and number of processors to keep efficiency constant**
  - **We call such systems *scalable* parallel systems**
  - **Scalability: the ability to increase performance as number of processors increases**

Isoefficiency Metric of Scalability

- For a given problem size, as we increase the number of processors, the overall efficiency of the parallel system goes down for all systems
- For some systems, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processors constant





Isoefficiency Metric of Scalability

- **Isoefficiency relation**
  - **What is the rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed?**
  - **A necessary condition of isoefficiency**

    $E \leq 1 / [1+T_O(n,p)/T(n,1)]$

    $\Rightarrow T(n,1) \geq C\, T_O(n,p)$ **where** $C = E/(1-E)$

    $\Rightarrow n \geq f(p)$ **after algebraic transformation**
- **This rate determines the scalability of the system. The slower this rate, the better**

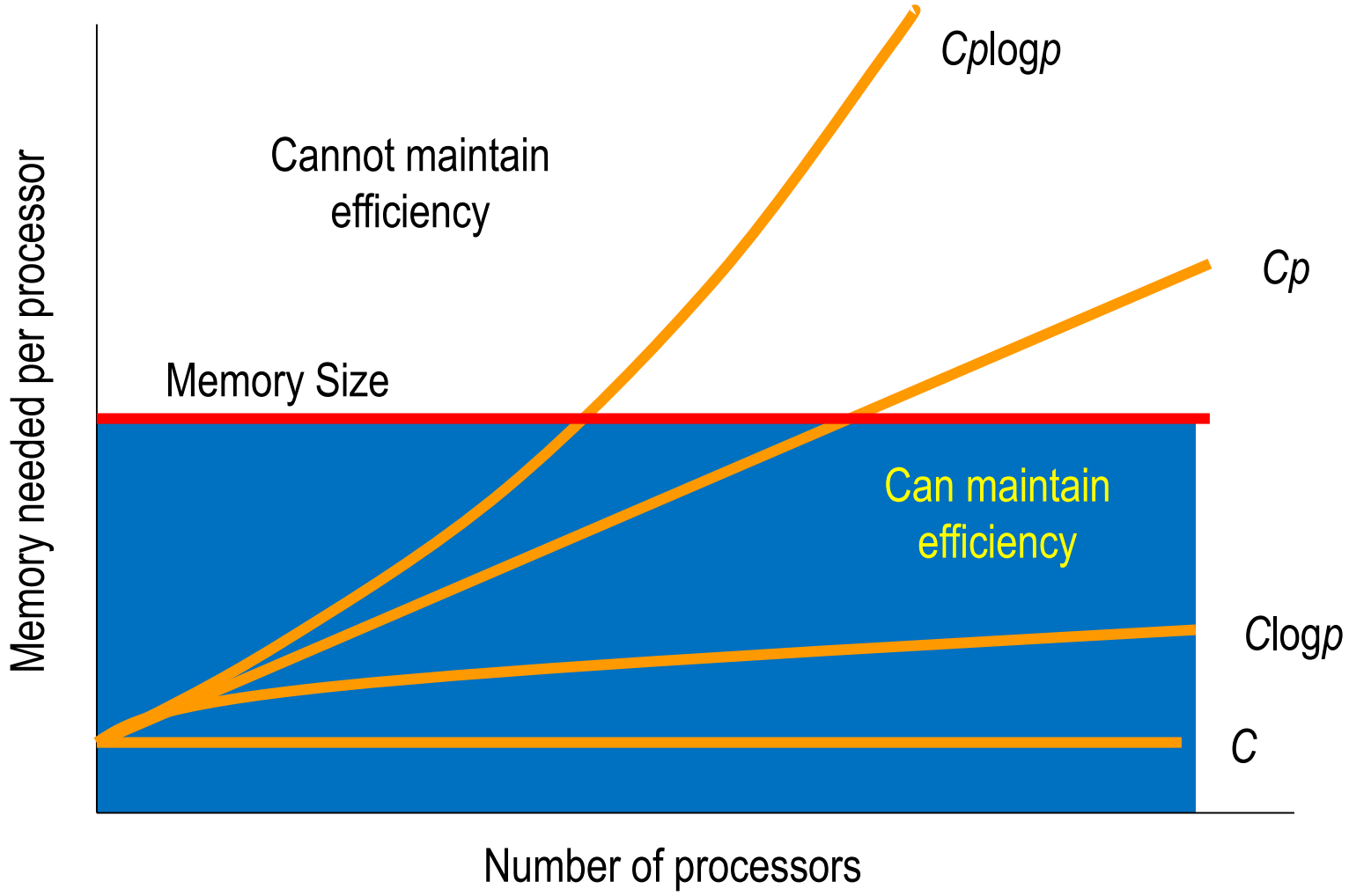
Scalability Function

- **Suppose isoefficiency relation is $n \geq f(p)$**
- **Let $M(n)$ denote memory required for problem size n**
- **$M(f(p))/p$ shows how memory usage per processor must increase to maintain same efficiency**
- **We call $M(f(p))/p$ the scalability function**

Meaning of Scalability Function

- **To maintain efficiency when increasing p, we must increase n**
- **Maximum problem size limited by available memory, which is linear in p**
- **Scalability function shows how memory usage per processor must grow to maintain efficiency**
- **Scalability function a constant means parallel system is perfectly scalable**

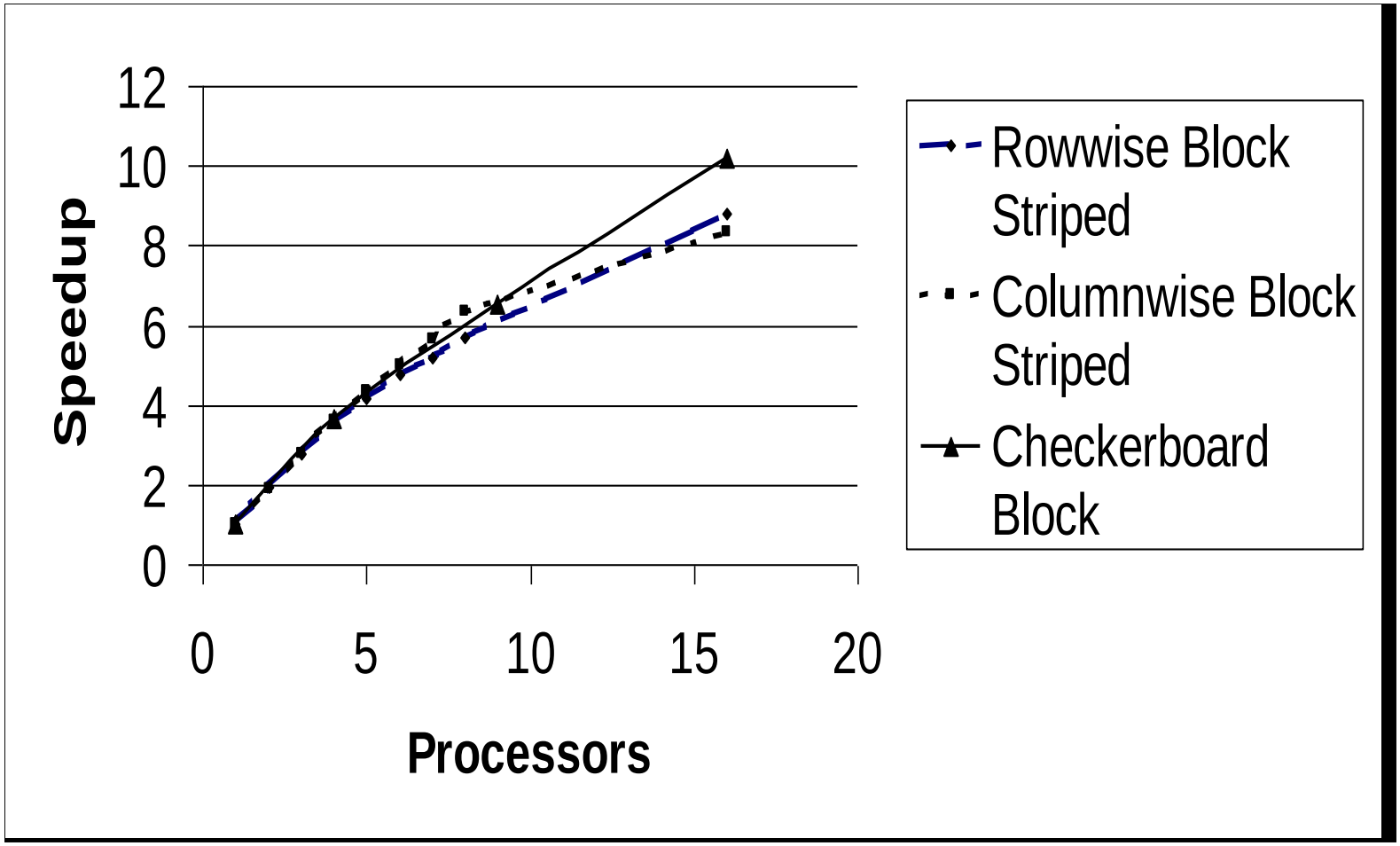
Interpreting Scalability Function



Scalabilities of the Parallel Matrix-Vector Multiplication Algorithms

- **Rowwise block striped:** C^2p
- **Columnwise block striped:** C^2p
- **Checkerboard block decomposition:** C^2log^2p

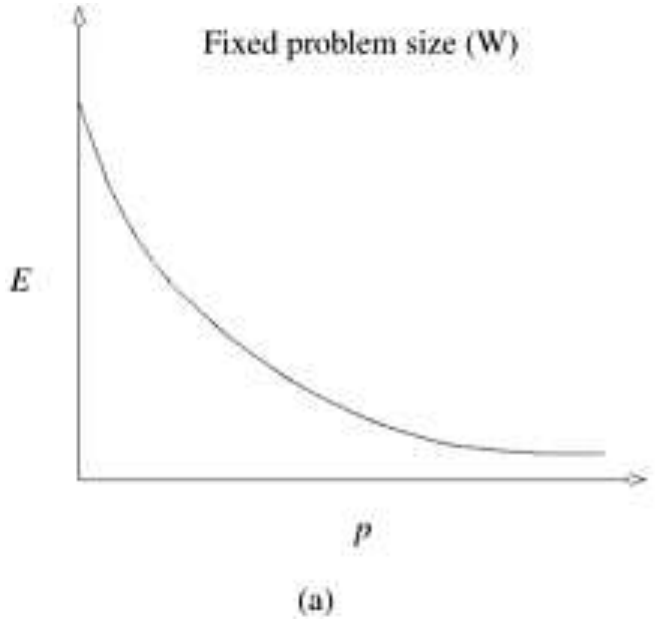
Comparison of Three Algorithms

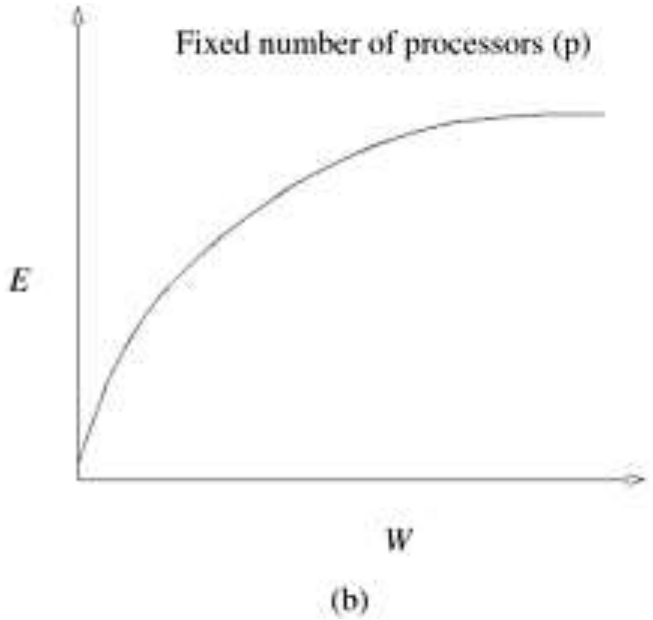


Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Isoefficiency Metric of Scalability

- For a given problem size, as we increase the number of processors, the overall efficiency of the parallel system goes down for all systems
- For *some* systems, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processors constant





Isoefficiency Metric of Scalability

- **Isoefficiency relation**
  - **What is the rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed?**
  - **A necessary condition of isoefficiency**

    $E \leq 1 / [1+T_O(n,p)/T(n,1)]$

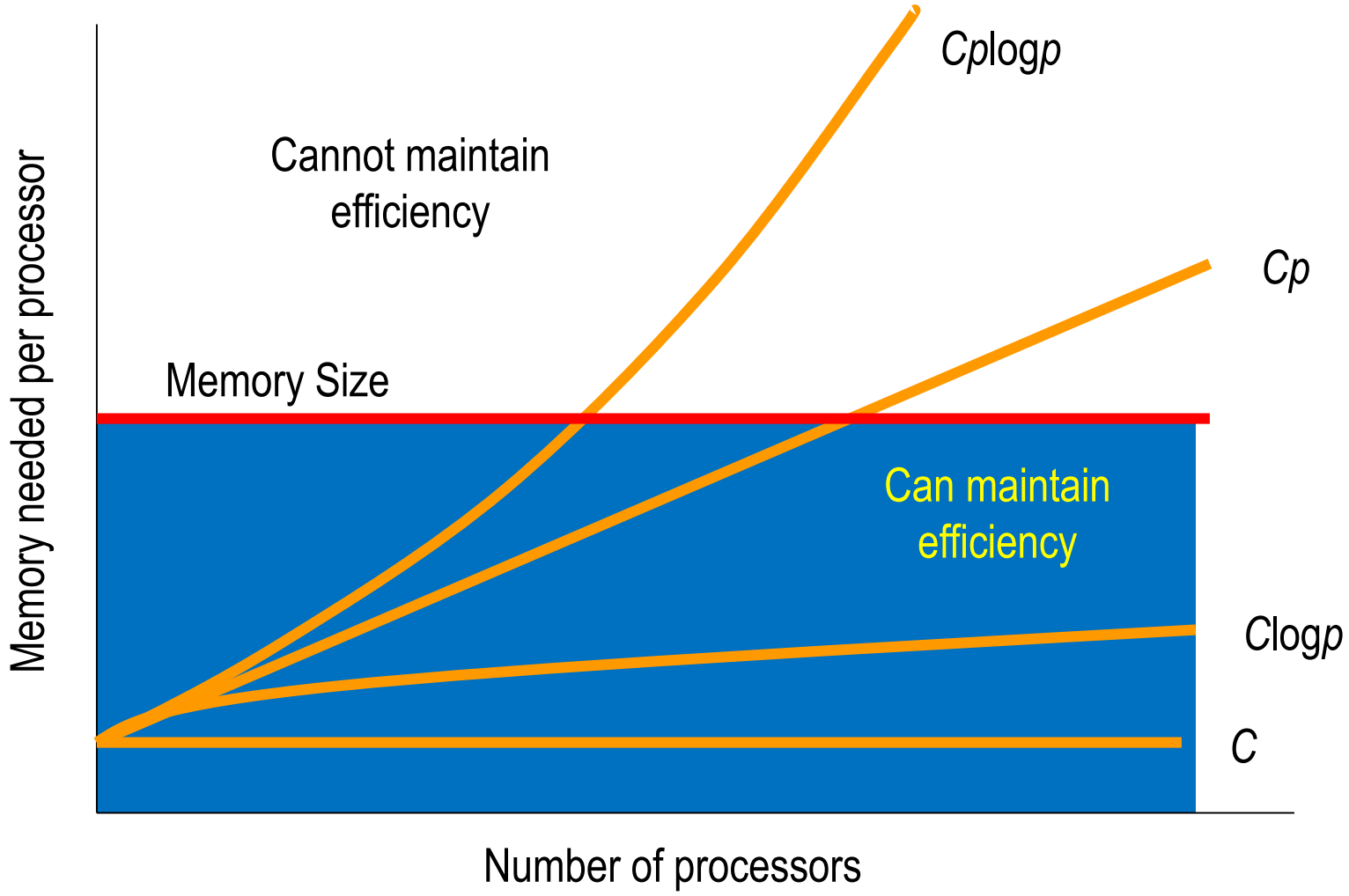
    $\Rightarrow T(n,1) \geq C\, T_O(n,p)$ **where** $C = E/(1-E)$

    $\Rightarrow n \geq f(p)$ **after algebraic transformation**
- **This rate determines the scalability of the system. The slower this rate, the better**

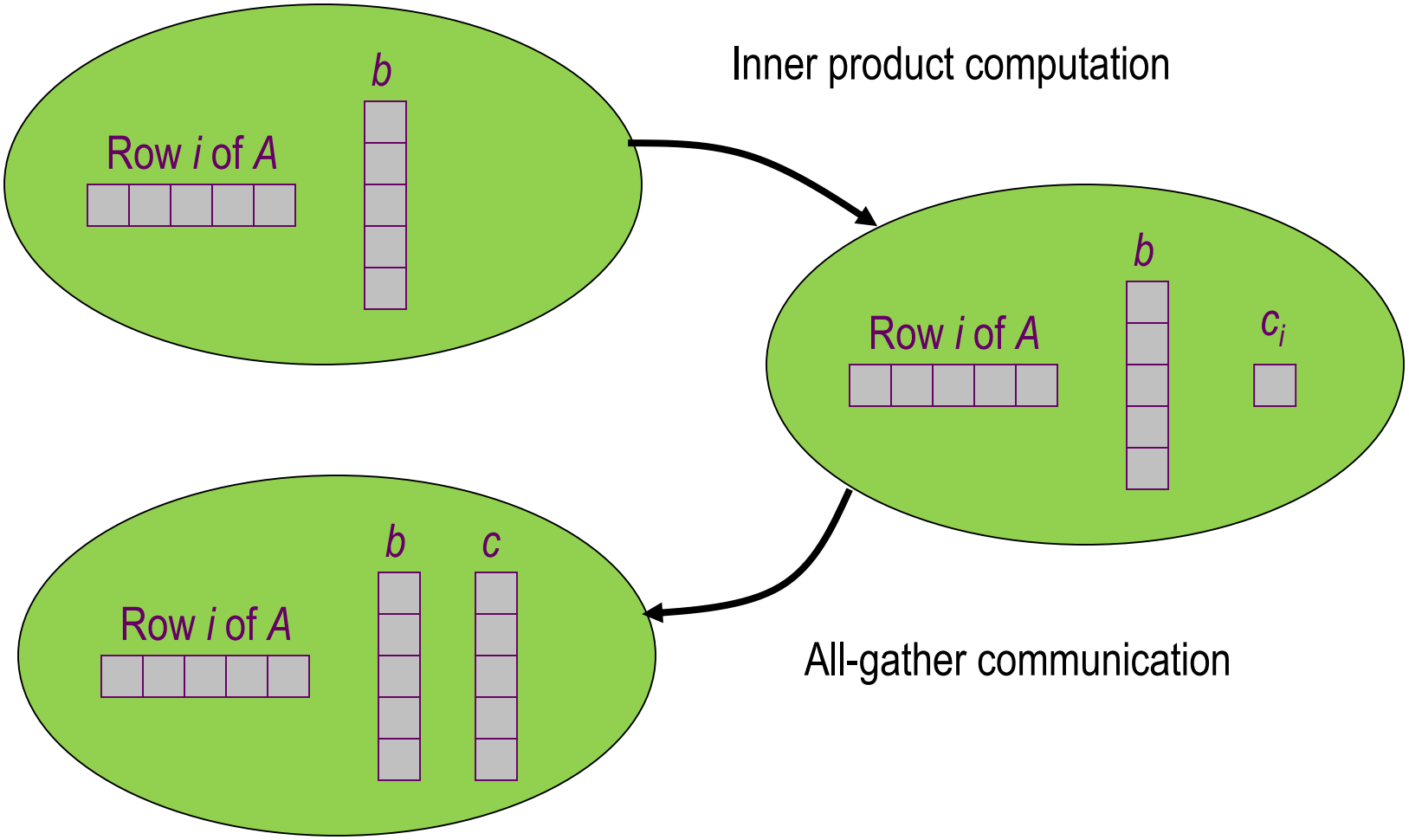
Scalability Function

- **Suppose isoefficiency relation is $n \geq f(p)$**
- **Let $M(n)$ denote memory required for problem size n**
- **We call $M(f(p))/p$ the scalability function**
  - **$M(f(p))/p$ shows how memory usage per processor must increase to maintain same efficiency**
- **What does it mean?**
  - **Assumption: mem $\propto$ #proc**
  - **If $M(f(p))/p$ increases with p, we cannot maintain efficiency**
    - **after p is large enough, even when we solve larger problems**
  - **The system is not scalable**
    - **The speedup will be smaller than on a small-scale system**

Interpreting Scalability Function



Rowwise Block Striped Matrix



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Complexity Analysis

- **Sequential algorithm complexity:** $\Theta(n^2)$
- **Parallel algorithm computational complexity:** $\Theta(n^2/p)$
- **Communication complexity of all-to-all broadcast:**

  $\Theta(\log p + n/p * (p-1))$

  - **(hypercube algorithm)**
- **Overall complexity:** $\Theta(n^2/p + n + \log p)$

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

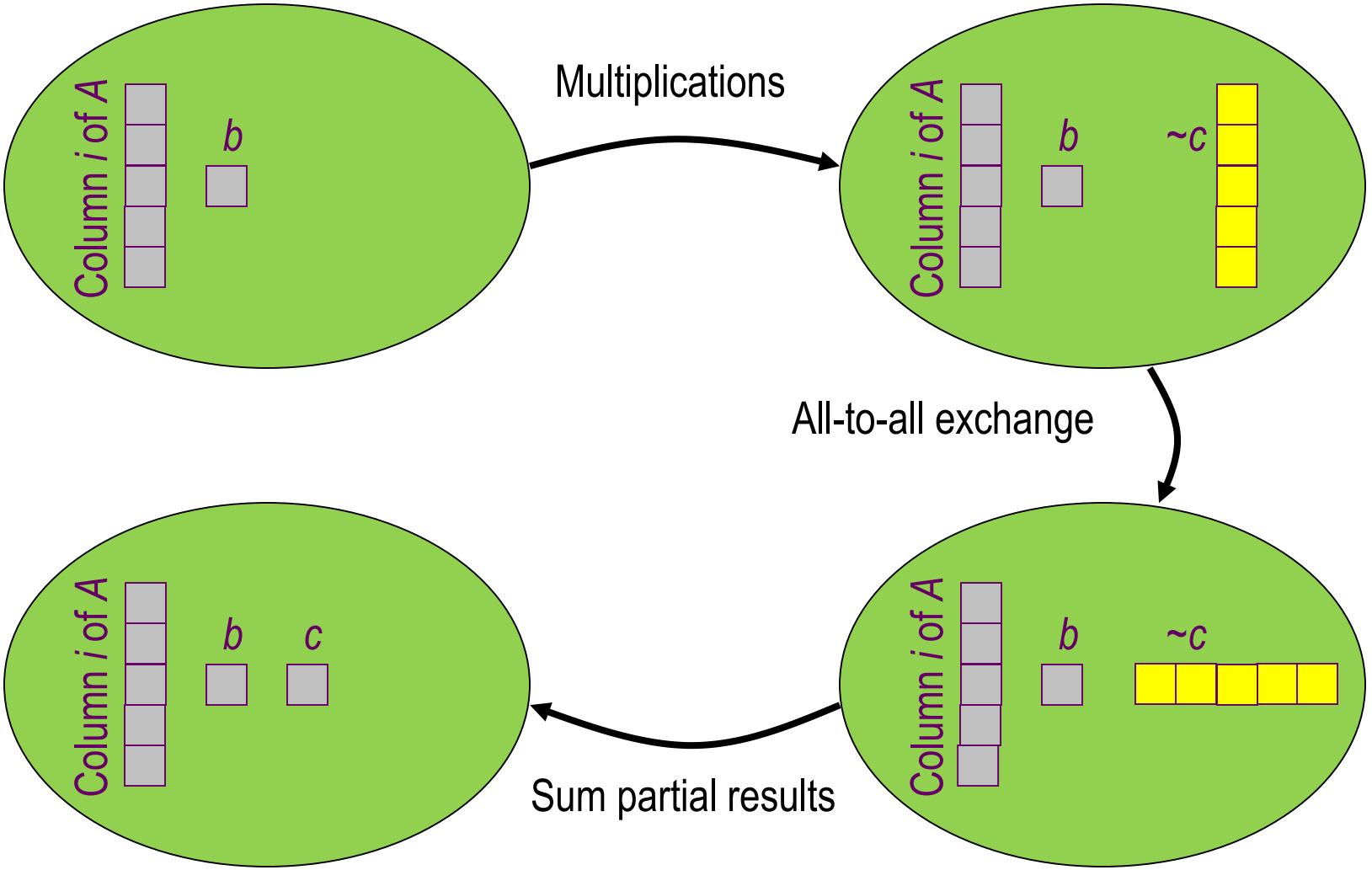
Isoefficiency Analysis

- **Sequential time complexity:** $\Theta(n^2)$
- **Only parallel overhead is all-to-all broadcast**
  - **When n is large, message transmission time dominates message latency**
  - **Parallel communication time:** $\Theta(n)$
- **Scalability function**
  - **Isoefficiency relation** $n^2 \geq Cpn \Rightarrow n \geq Cp$
  - **Memory utilization** $M(n) = n^2$

  ⇒ Scalability function $M(Cp)/p = C^2p^2/p = C^2p$
- **System is not highly scalable**
  - **To maintain constant efficiency, memory utilization per processor must grow linear with the number of processors**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Columnwise Block Strip Matrix



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Complexity Analysis

- **Sequential algorithm complexity:** $\Theta(n^2)$
- **Parallel algorithm computational complexity:** $\Theta(n^2/p)$
- **Communication complexity of all-to-all**
  - **All-to-all exchange:** $\Theta(p+n)$
- **Overall complexity:** $\Theta(n^2/p + n + p)$

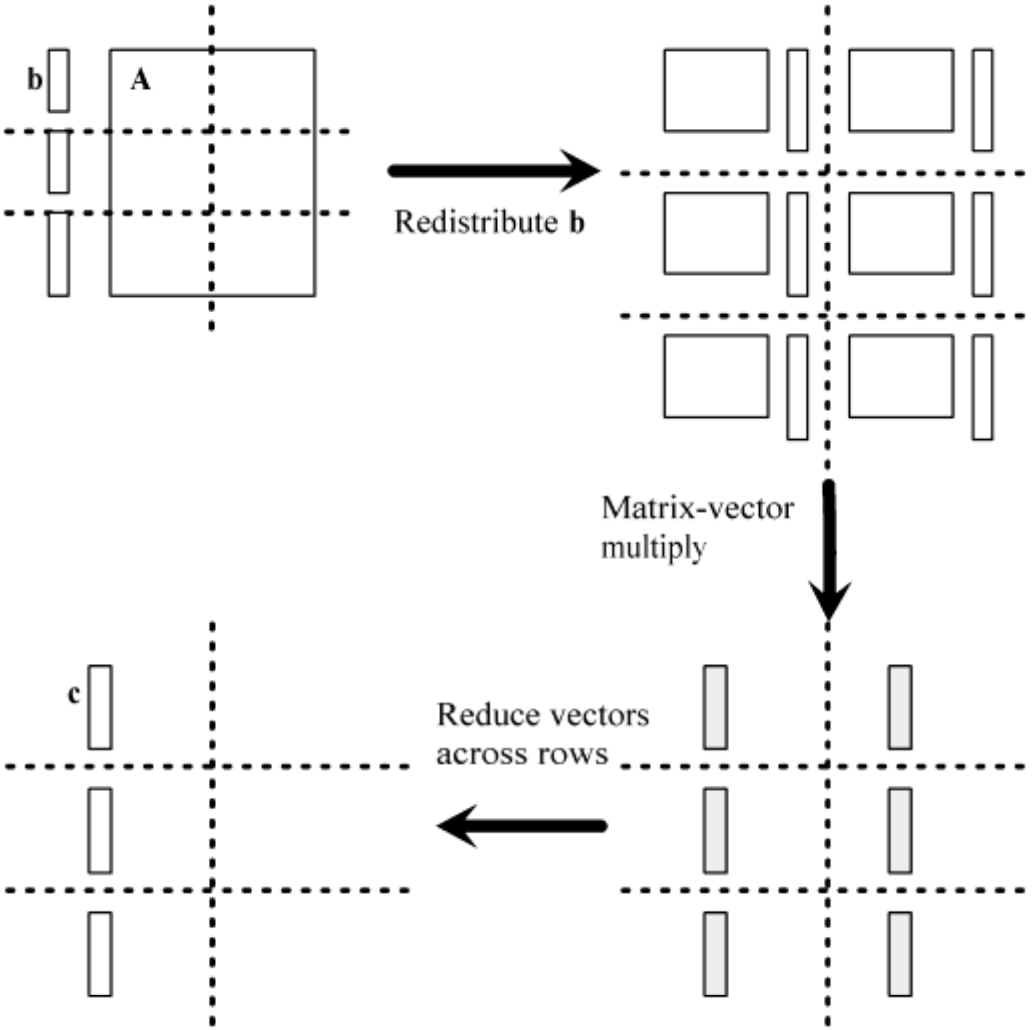
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Isoefficiency Analysis

- **Sequential time complexity:** $\Theta(n^2)$
- **Only parallel overhead is all-to-all**
  - **When n is large, message transmission time dominates message latency**
  - **Parallel communication time:** $\Theta(n)$
- **Isoefficiency relation** $n^2 \geq Cpn \Rightarrow n \geq Cp$
- **Scalability function same as rowwise algorithm:** C^2p

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Checkerboard Block Decomposition



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Complexity Analysis

- **Assume p is a square number**
  - **If grid is $1 \times p$, devolves into columnwise block striped**
  - **If grid is $p \times 1$, devolves into rowwise block striped**
  - **(we've already analyzed these two special cases)**
  - **If grid is $\lceil n/\sqrt{p} \rceil \times \lceil n/\sqrt{p} \rceil$, ...**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Complexity Analysis (continued)

- **Each process does its share of computation:**

  $\Theta(n^2/p)$

- **Redistribute b:**

  $\Theta(n/\sqrt{p} * \log p)$

- **Reduction of partial results vectors:**

  $\Theta(n/\sqrt{p} * \log p)$

- **Overall parallel complexity:**

  $\Theta(n^2/p + n \log p / \sqrt{p})$

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Isoefficiency Analysis

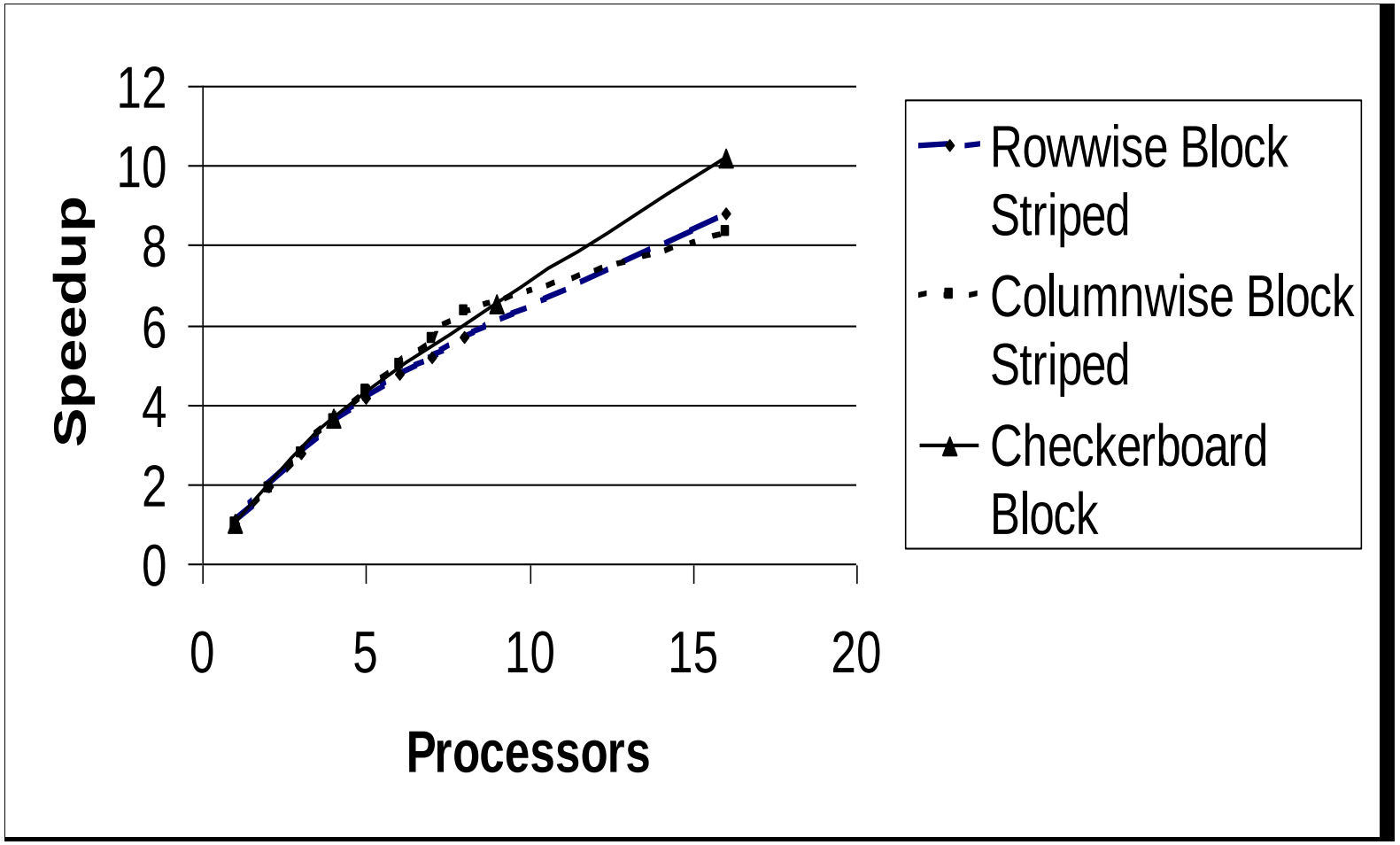
- **Sequential complexity:** $\Theta(n^2)$
- **Parallel communication complexity:** $\Theta(n \log p / \sqrt{p})$
- **Isoefficiency function:** $n^2 \geq Cn\sqrt{p}\log p \Rightarrow n \geq C\sqrt{p}\log p$

$$M(C\sqrt{p}\log p)/p = C^2 p \log^2 p / p = C^2 \log^2 p$$

- **This system is much more scalable than the previous two implementations**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Comparison of Three Algorithms

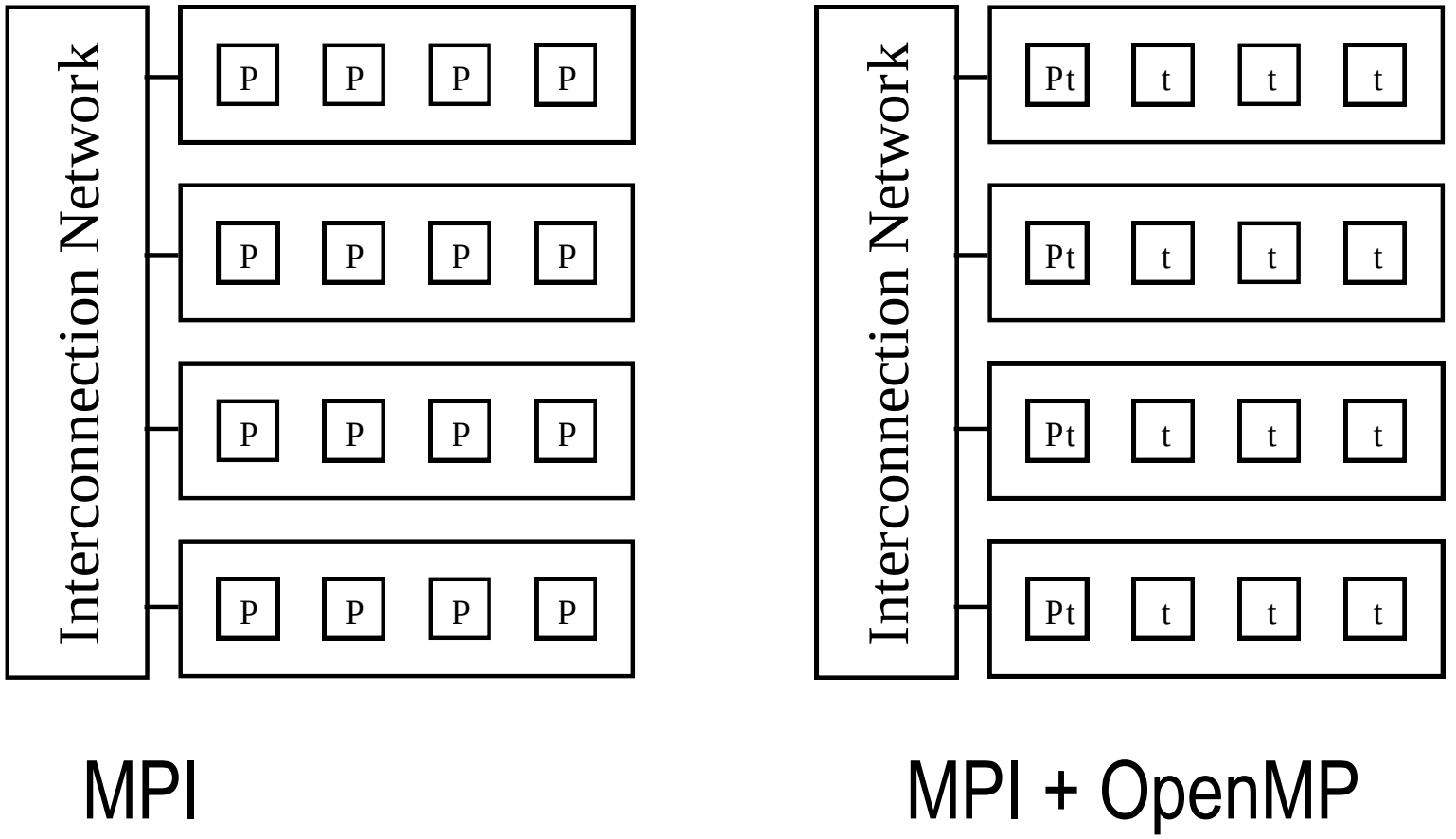


Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

MPI + OpenMP

- **Advantages of using both MPI and OpenMP**
  - **Case Study: Jacobi method**
    - **An iterative methods to solve linear systems**

MPI vs. MPI+OpenMP



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Why MPI + OpenMP Can Execute Faster?

- **Lower communication overhead**
  - **Message passing with *mk* processes, versus**
  - **Message passing with *m* processes with *k* threads each**
- **More portions of program may be practical to parallelize**
- **May allow more overlap of communications with computations**
  - **For example, if 3 MPI processes are waiting for messages, and 1 MPI process is active, it is worthwhile to fork some threads to speedup the 4th process**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Hybrid Light-Weight Threads & Heavier-Weight Processes

- For example, a serial program runs in 100s
  - S: 5s is inherent sequential
  - P_1: 5s are parallelizable but not worth message passing
  - P_2: 90s are perfectly parallelizable
- MPI-only with 16 processes
  - Replicate the P_1 part of program on all processes
  - Speedup = 1 / (0.10 + 0.90/16) = 6.4
- Hybrid MPI & threads
  - Execute the replicated P_1 with 2 threads
  - Speedup = 1 / (0.05 + 0.05/2 + 0.90/16) = 7.6
  - 19% faster than MPI-only

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Code for matrix_vector_product

```
void matrix_vector_product (int id, int p,
   int n, double **a, double *b, double *c)
{
   int    i, j;
   double tmp;       /* Accumulates sum */
   for (i=0; i<BLOCK_SIZE(id,p,n); i++) {
      tmp = 0.0;
      for (j = 0; j < n; j++)
         tmp += a[i][j] * b[j];
      piece[i] = tmp;
   }
   new_replicate_block_vector (id, p,
      piece, n, (void *) c, MPI_DOUBLE);
}
```

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Adding OpenMP Directives

- **Want to minimize fork/join overhead by making parallel the outermost possible loop**
- **Outer loop may be executed in parallel if each thread has a private copy of *tmp* and *j***

```
#pragma omp parallel for private(j,tmp)
    for (i=0; i<BLOCK_SIZE(id,p,n); i++) {
```

User Control of Threads

- **Want to give user opportunity to specify number of active threads per process**
- **Add a call to *omp_set_num_threads* to function main**
- **Argument comes from command line**

```
omp_set_num_threads (atoi(argv[3]));
```

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

What Happened?

- **We transformed a MPI program to a MPI+OpenMP program by adding only two lines to our program!**

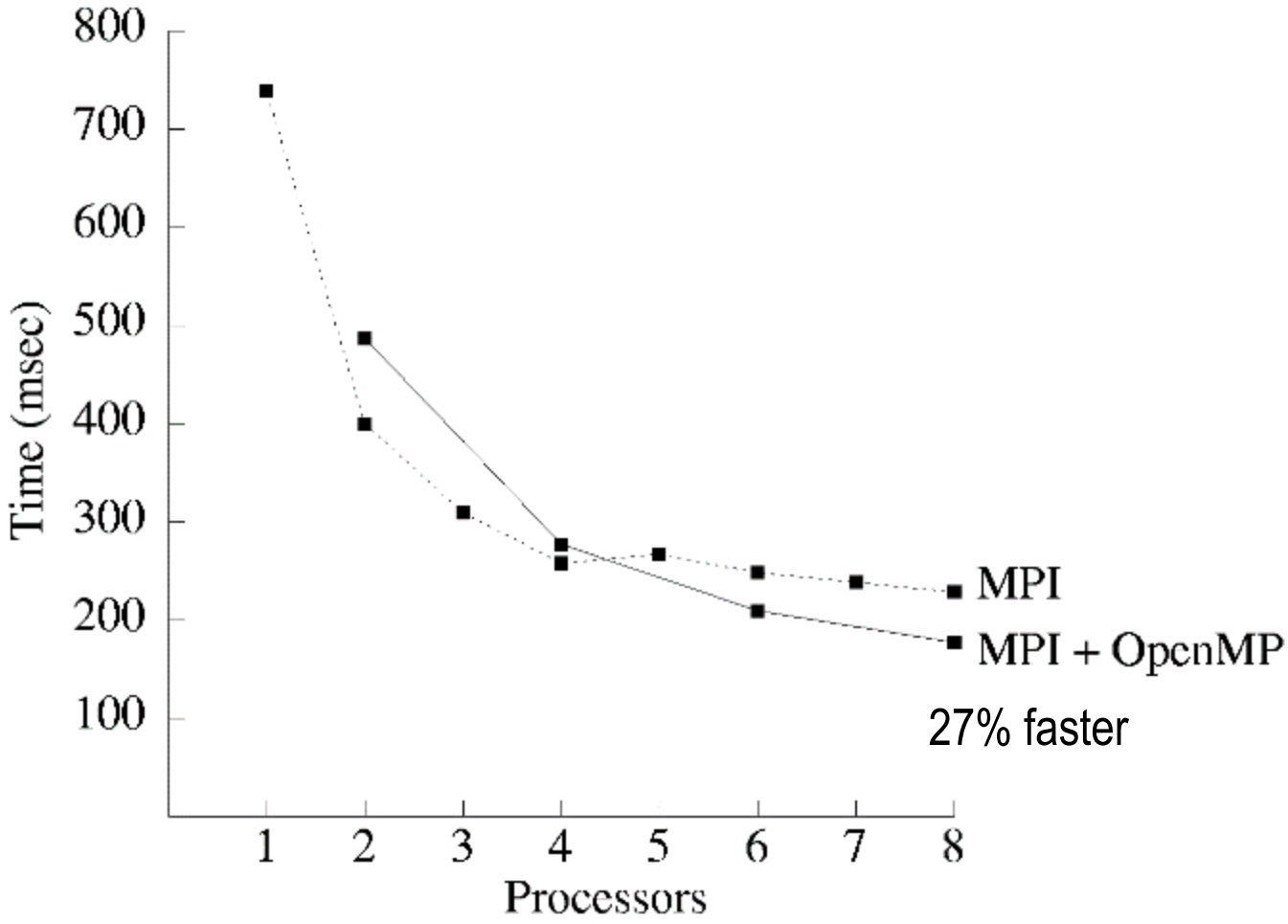
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Benchmarking

- Target system: a commodity cluster with four dual-processor nodes
- MPI program executes on 1, 2, ..., 8 CPUs
  - On 1, 2, 3, 4 CPUs, each process on different node, maximizing memory bandwidth per CPU
- MPI+OpenMP program executes on 1, 2, 3, 4 processes
  - Each process has two threads
  - C+MPI+OpenMP program executes on 2, 4, 6, 8 threads

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Results of Benchmarking



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Analysis of Results

- **MPI+OpenMP program slower on 2, 4 CPUs because MPI+OpenMP threads are sharing memory bandwidth, while C+MPI processes are not**
- **MPI+OpenMP programs faster on 6, 8 CPUs because they have lower communication cost**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Case Study: Jacobi Method

- Begin with MPI program that uses Jacobi method to solve steady state heat distribution problem
- Program based on rowwise block striped decomposition of two-dimensional matrix containing finite difference mesh

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Gaussian Elimination

- **Used to solve $Ax = b$ when A is dense**
- **Reduces $Ax = b$ to upper triangular system $Tx = c$**
- **Back substitution can then solve $Tx = c$ for x**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Gaussian Elimination Example (1/4)

$$
\begin{aligned}
4x_0 \quad &+6x_1 \quad +2x_2 \quad -2x_3 \quad = \quad 8 \\
2x_0 \quad & \qquad\quad +5x_2 \quad -2x_3 \quad = \quad 4 \\
-4x_0 \quad &-3x_1 \quad -5x_2 \quad +4x_3 \quad = \quad 1 \\
8x_0 \quad &+18x_1 \quad -2x_2 \quad +3x_3 \quad = \quad 40
\end{aligned}
$$

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Gaussian Elimination Example (2/4)

$$
\begin{aligned}
4x_0 \quad &+6x_1 \quad +2x_2 \quad -2x_3 \quad = \quad 8 \\
-3x_1 \quad &+4x_2 \quad -1x_3 \quad = \quad 0 \\
+3x_1 \quad &-3x_2 \quad +2x_3 \quad = \quad 9 \\
+6x_1 \quad &-6x_2 \quad +7x_3 \quad = \quad 24
\end{aligned}
$$

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Gaussian Elimination Example (3/4)

$$
\begin{aligned}
4x_0 \quad &+6x_1 \quad +2x_2 \quad -2x_3 \quad = \quad 8 \\
-3x_1 \quad &+4x_2 \quad -1x_3 \quad = \quad 0 \\
&1x_2 \quad +1x_3 \quad = \quad 9 \\
&2x_2 \quad +5x_3 \quad = \quad 24
\end{aligned}
$$

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Gaussian Elimination Example (4/4)

$$
\begin{aligned}
4x_0 \quad +6x_1 \quad +2x_2 \quad -2x_3 &= 8 \\
-3x_1 \quad +4x_2 \quad -1x_3 &= 0 \\
1x_2 \quad +1x_3 &= 9 \\
3x_3 &= 6
\end{aligned}
$$

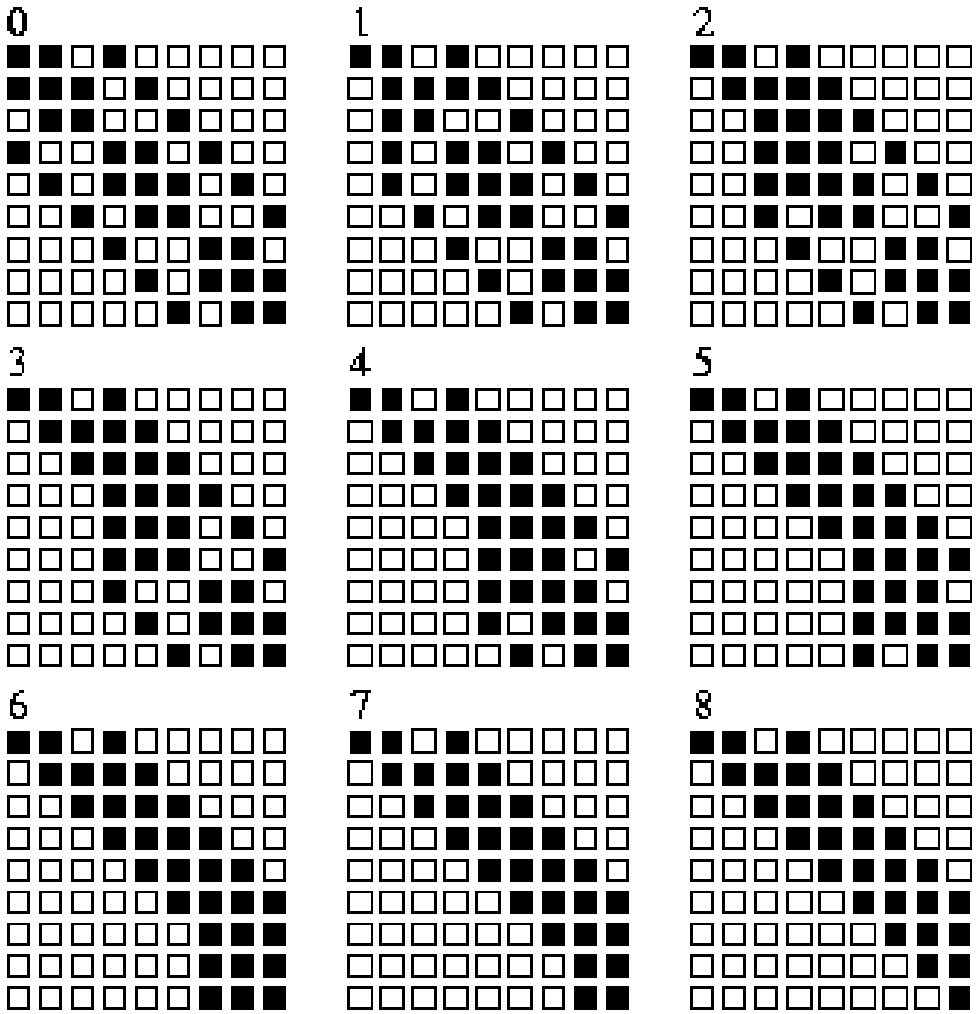
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Sparse Systems

- Gaussian elimination not well-suited for sparse systems
- Coefficient matrix gradually fills with nonzero elements
  - Increases storage requirements
  - Increases total operation count

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Example of “Fill” in Gaussian Elimination



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Iterative Methods

- **Iterative method: algorithm that generates a series of approximations to solution's value**
- **Require less storage than direct methods**
- **Since they avoid computations on zero elements, they can save a lot of computations**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

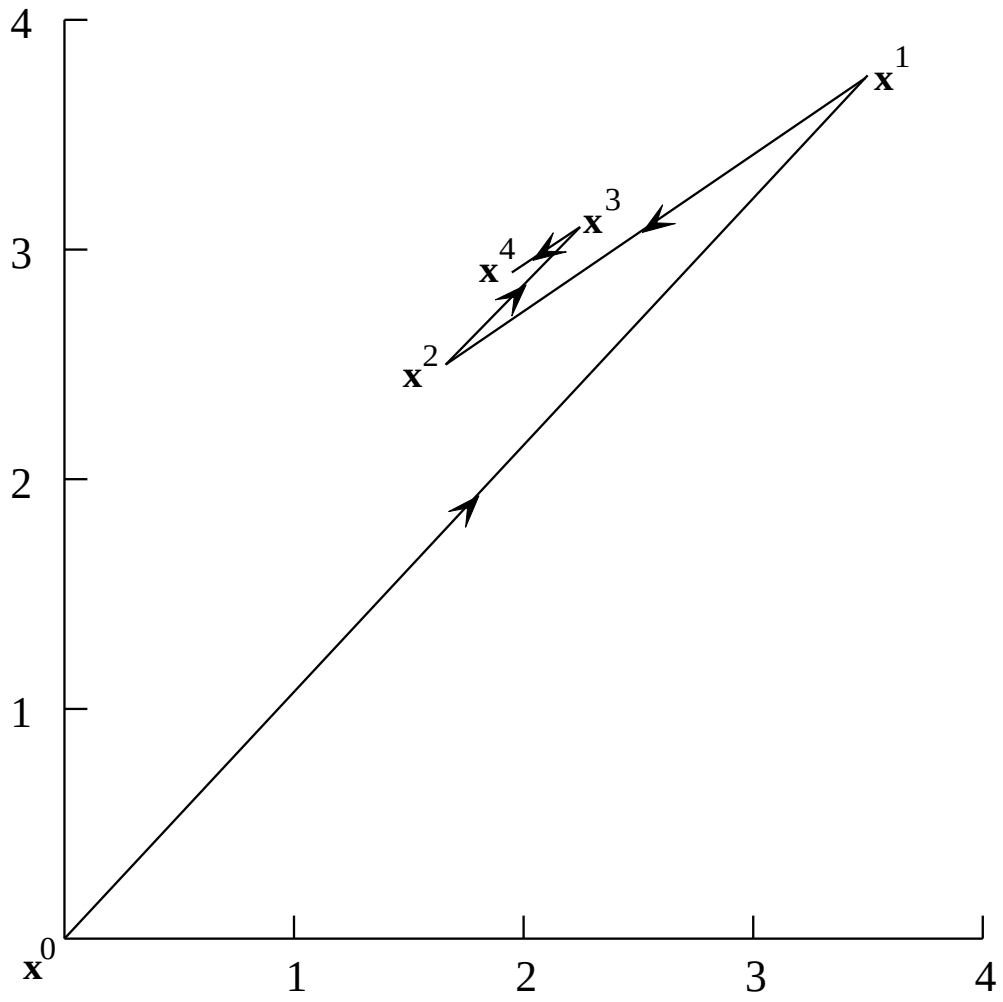
Jacobi Method

$$x_i^{(k+1)} = \frac{1}{a_{i,j}}\left(b_i - \sum_{j \neq i} a_{i,j} x_j^{(k)}\right)$$

Values of elements of vector x at iteration k+1 depend upon values of vector x at iteration k

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Jacobi Method Iterations



$$\begin{cases} 2x_1 + x_2 = 7 \\ x_1 + 3x_2 = 11 \end{cases}$$

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Rate of Convergence

- **Even when Jacobi method, rate of convergence often too slow to make them practical**
- **We will move on to an iterative method with much faster convergence**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

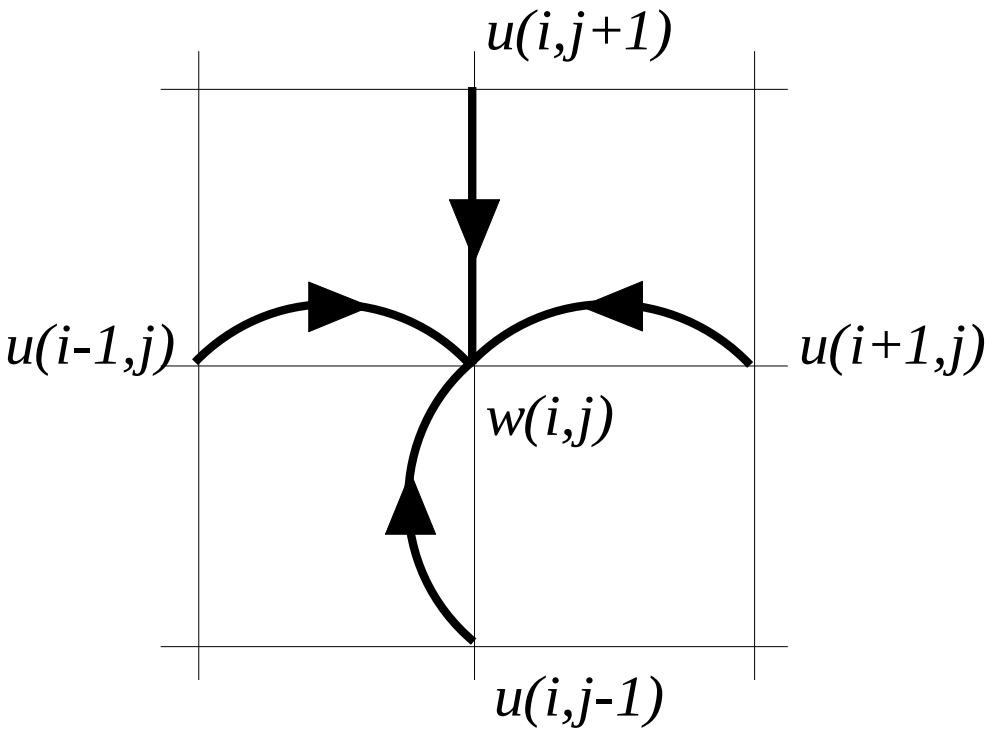
Methodology

- **Profile execution of MPI program**
- **Focus on adding OpenMP directives to most compute-intensive function**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Heart of Sequential C Program

```
w[i][j] = (u[i-1][j] + u[i+1][j] +
          u[i][j-1] + u[i][j+1]) / 4.0;
```



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Result of Profiling

Function	*1 CPU*	*8 CPUs*
initialize_mesh	0.01%	0.03%
find_steady_state	98.48%	93.49%
print_solution	1.51%	6.48%

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Function find_steady_state (1/2)

```
its = 0;
for (;;) {
   if (id > 0)
      MPI_Send (u[1], N, MPI_DOUBLE, id-1, 0,
         MPI_COMM_WORLD);
   if (id < p-1) {
      MPI_Send (u[my_rows-2], N, MPI_DOUBLE, id+1,
         0, MPI_COMM_WORLD);
      MPI_Recv (u[my_rows-1], N, MPI_DOUBLE, id+1,
         0, MPI_COMM_WORLD, &status);
   }
   if (id > 0)
      MPI_Recv (u[0], N, MPI_DOUBLE, id-1, 0,
         MPI_COMM_WORLD, &status);
```

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Function find_steady_state (2/2)

```
diff = 0.0;
for (i = 1; i < my_rows-1; i++)
   for (j = 1; j < N-1; j++) {
      w[i][j] = (u[i-1][j] + u[i+1][j] +
                 u[i][j-1] + u[i][j+1])/4.0;
      if (fabs(w[i][j] - u[i][j]) > diff)
         diff = fabs(w[i][j] - u[i][j]);
   }
for (i = 1; i < my_rows-1; i++)
   for (j = 1; j < N-1; j++)
      u[i][j] = w[i][j];
MPI_Allreduce (&diff, &global_diff, 1,
   MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
if (global_diff <= EPSILON) break;
its++;
```

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Making Function Parallel (1/2)

- **Except for two initializations and a return statement, function is a big `for` loop**
- **Cannot execute `for` loop in parallel**
  - **Not in canonical form**
  - **Contains a `break` statement**
  - **Contains calls to MPI functions**
  - **Data dependences between iterations**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Making Function Parallel (2/2)

- **Focus on first `for` loop indexed by `i`**
- **How to handle multiple threads testing/updating `diff`?**
- **Putting `if` statement in a critical section would increase overhead and lower speedup**
- **Instead, create private variable `tdiff`**
- **Thread tests `tdiff` against `diff` before call to `MPI_Allreduce`**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Modified Function

```
 diff = 0.0;
#pragma omp parallel private (i, j, tdiff)
{
      tdiff = 0.0;
#pragma omp for
      for (i = 1; i < my_rows-1; i++)
          ...
#pragma omp for nowait
      for (i = 1; i < my_rows-1; i++)
#pragma omp critical
      if (tdiff > diff) diff = tdiff;
}
      MPI_Allreduce (&diff, &global_diff, 1,
         MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
```

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Making Function Parallel (3/3)

- **Focus on second `for` loop indexed by `i`**
- **Copies elements of `w` to corresponding elements of `u`: no problem with executing in parallel**

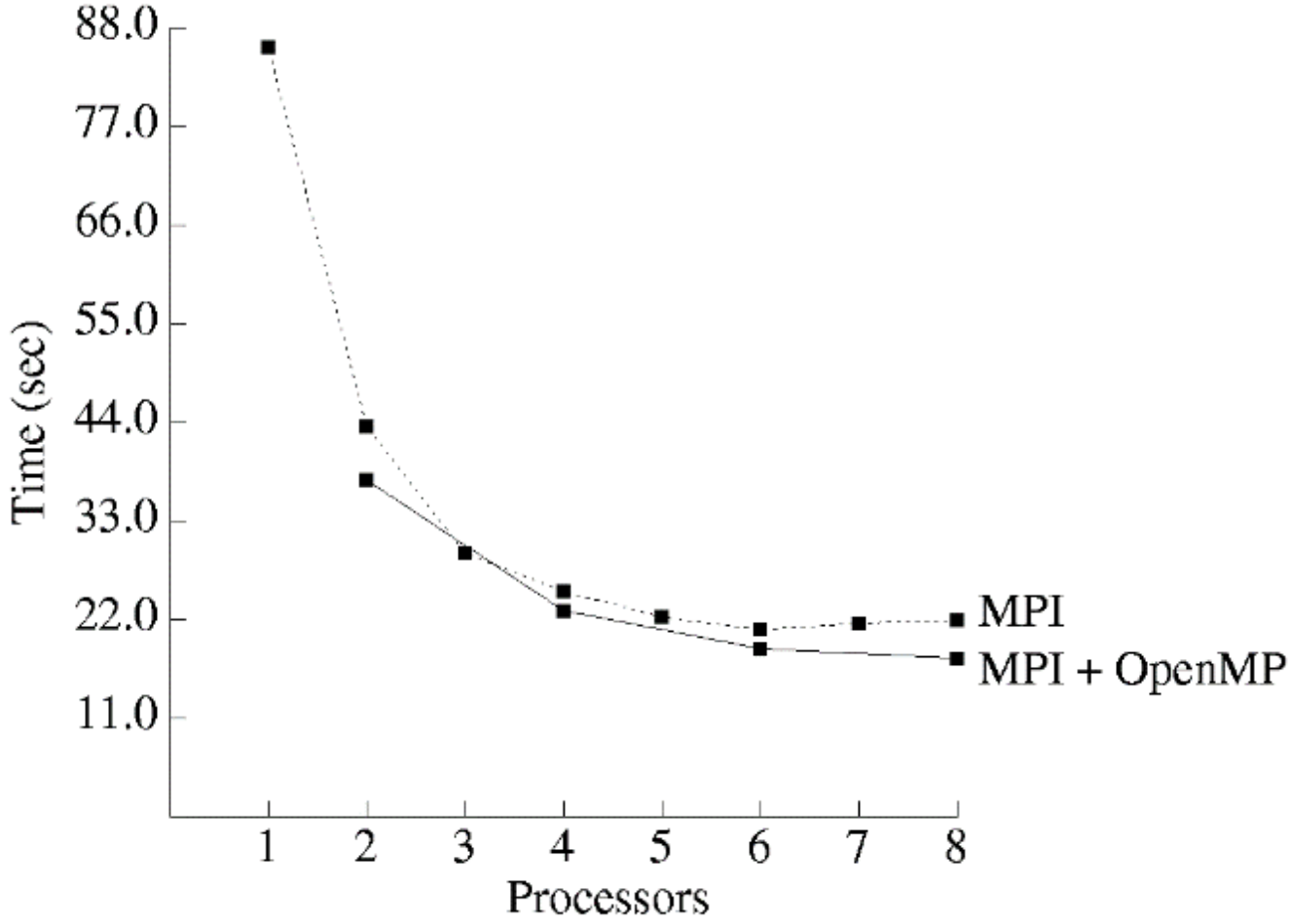
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Benchmarking

- Target system: a commodity cluster with four dual-processor nodes
- C+MPI program executes on 1, 2, ..., 8 processes
  - On 1, 2, 3, 4 CPUs, each process on different node, maximizing memory bandwidth per CPU
- C+MPI+OpenMP program executes on 1, 2, 3, 4 processes
  - Each process has two threads
  - C+MPI+OpenMP program executes on 2, 4, 6, 8 threads

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Benchmarking Results



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Analysis of Results

- **Hybrid C+MPI+OpenMP program uniformly faster than C+MPI program**
- **Computation/communication ratio of hybrid program is superior**
- **Number of mesh points per element communicated is twice as high per node for the hybrid program**
- **Lower communication overhead leads to 19% better speedup on 8 CPUs**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Summary

- Many contemporary parallel computers consists of a collection of multiprocessors
- On these systems, performance of C+MPI+OpenMP programs can exceed performance of C+MPI programs
- OpenMP enables us to take advantage of shared memory to reduce communication overhead
- Often, conversion requires addition of relatively few pragmas

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.