Introduction to Parallel & Distributed Computing

Programming with MPI

Scalability & Mixed MPI+OpenMP

Lecture 8, Spring 2023

Instructor: 罗国杰

gluo@pku.edu.cn

Overview

- Isoefficiency metric
  - An attempt to model "scalability"
- Performance modeling

Understanding Scaling

- **Scaling in computational complexity**
  - `s: problem_size → time/space`
  - **How much more time/space to spend for a larger problem?**
- **in parallel computing**
  - **factors: 1) machine size, 2) problem size, 3) time/space**

When We Talk about Scaling

- **Problem-constrained scaling**
  - use a parallel computer to solve the same problem faster
    - pitfalls: small problems may not be realistic workloads for larger machines; big problems may not fit on small machines
  - examples: almost everything we considered parallelizing
  - it is often called “hard scaling”

partly from: Stanford CS149

When We Talk about Scaling

- **Time-constrained scaling**
  - completing more work in a fixed amount of time
  - examples
    - 3D graphics: #polygons, texels sampled, shader length, etc.

    - computational finance: run the most sophicated model in 1 ms, 1 min, overnight, etc.
    - modern websites: respond to user in X ms
    - real-time CV for robots: obstacle detection in 5 ms

partly from: Stanford CS149

When We Talk about Scaling

- **Memory-constrained scaling**
  - run the largest possible problem without memory overflow
  - memory per processor is fixed
    - assumptions: 1) memory resources scale with processor count; 2) spilling to disk is infeasible behavior (too slow)
    - scaling by adding more machines to cluster
  - it is often called “weak scaling”
  - examples
    - large N-body problem
    - large-scale machine learning
    - memcached (more servers = more available cache)

partly from: Stanford CS149

Isoefficiency & Scalability Metric

- **Isoefficiency is proposed in [GGK'93]**
  - **A. Y. Grama, A. Gupta, and V. Kumar, "Isoefficiency: measuring the scalability of parallel algorithms and architectures," IEEE Parallel Distrib. Technol. Syst. Appl., vol. 1, no. 3, pp. 12–21, Aug. 1993, doi: 10.1109/88.242438.**
- **Measuring "scalability" through efficiency**

Speedup & Efficiency

- **Execution time**

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

$$\text{serial} + \text{parallel} + \text{communication}$$

- **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 program?
- A: Ideally, a scalable parallel program 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 Programs

- **Execution time**

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

serial + parallel + communication

- **Total parallel overhead**

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

- **Efficiency**

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

Scaling Characteristics of Parallel Programs

- **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**

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

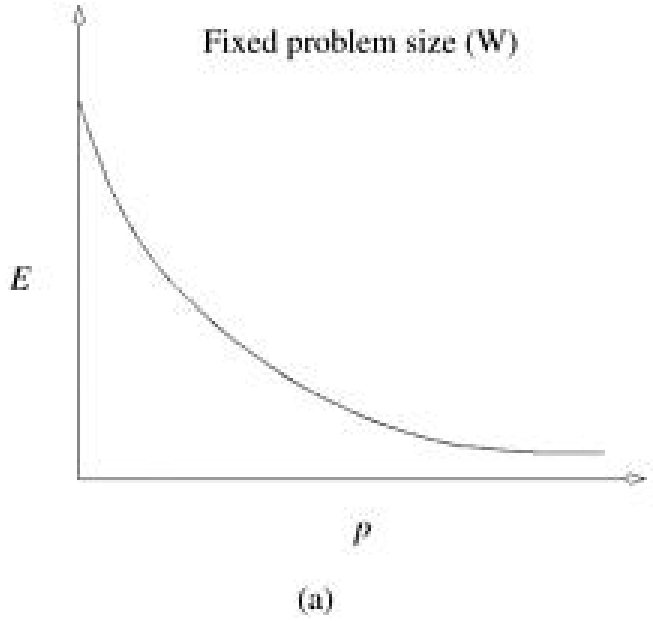
  => the efficiency goes down if p is increased with a fixed n
- **In many cases, $T_O(n,p)$ grows *sublinearly* with respect to $T(n,1)$**

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

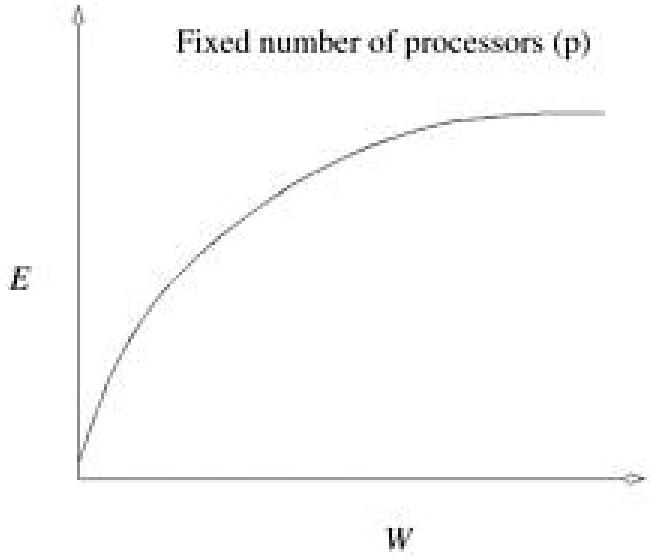
  => In such cases, the efficiency goes up if n is increased with a fixed p
- **For such parallel programs, we can simultaneously increase the problem size and number of processors to keep efficiency constant**
  - **We call such parallel program *scalable***
  - **Scalability: the ability to increase 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 program goes down for all systems
- For many problems, the efficiency of a parallel program increases if the problem size is increased while keeping the number of processors constant



(a)



(b)

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 parallel program. The slower this rate, the better**

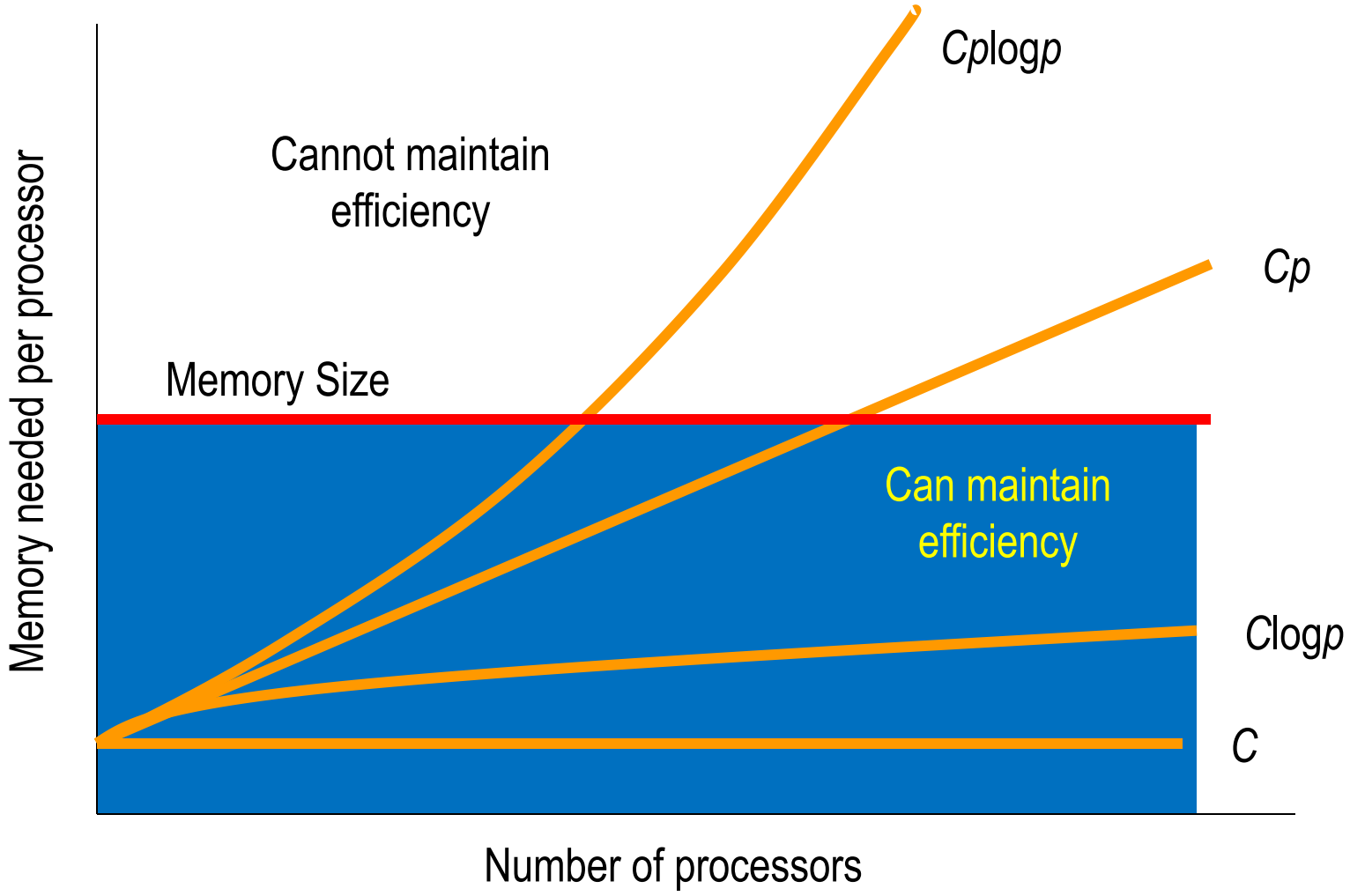
Scalability Function

- **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

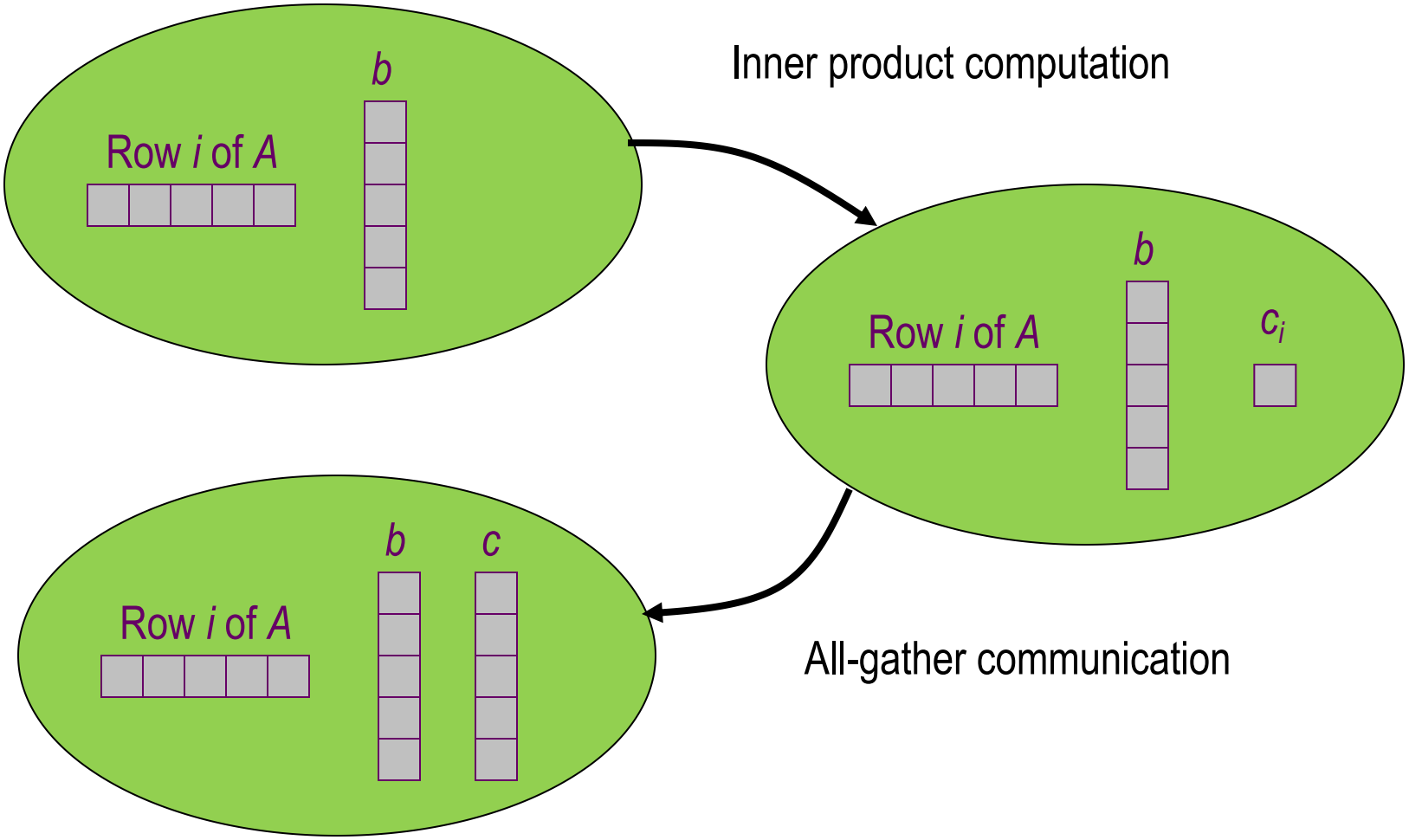
Interpreting Scalability Function



Scalabilities of the Parallel Matrix-Vector Multiplication Algorithms

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

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.

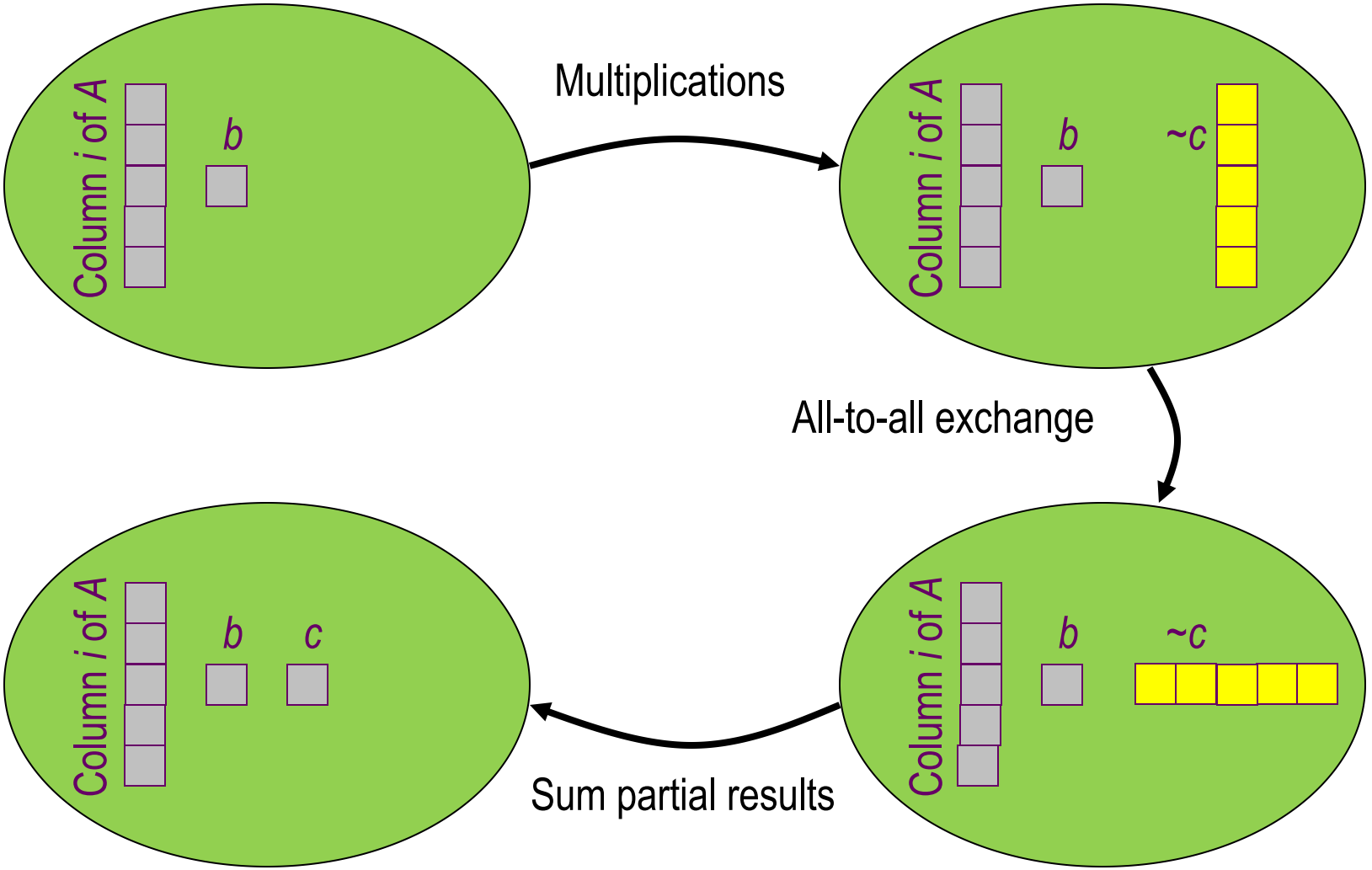
Isoefficiency Analysis

- **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** $M(n) = n^2$

  $\Rightarrow$ **Scalability function** $M(Cp)/p = C^2p^2/p = C^2p$
- **This parallel algorithm 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:** $\Theta(n^2/p + n + p)$

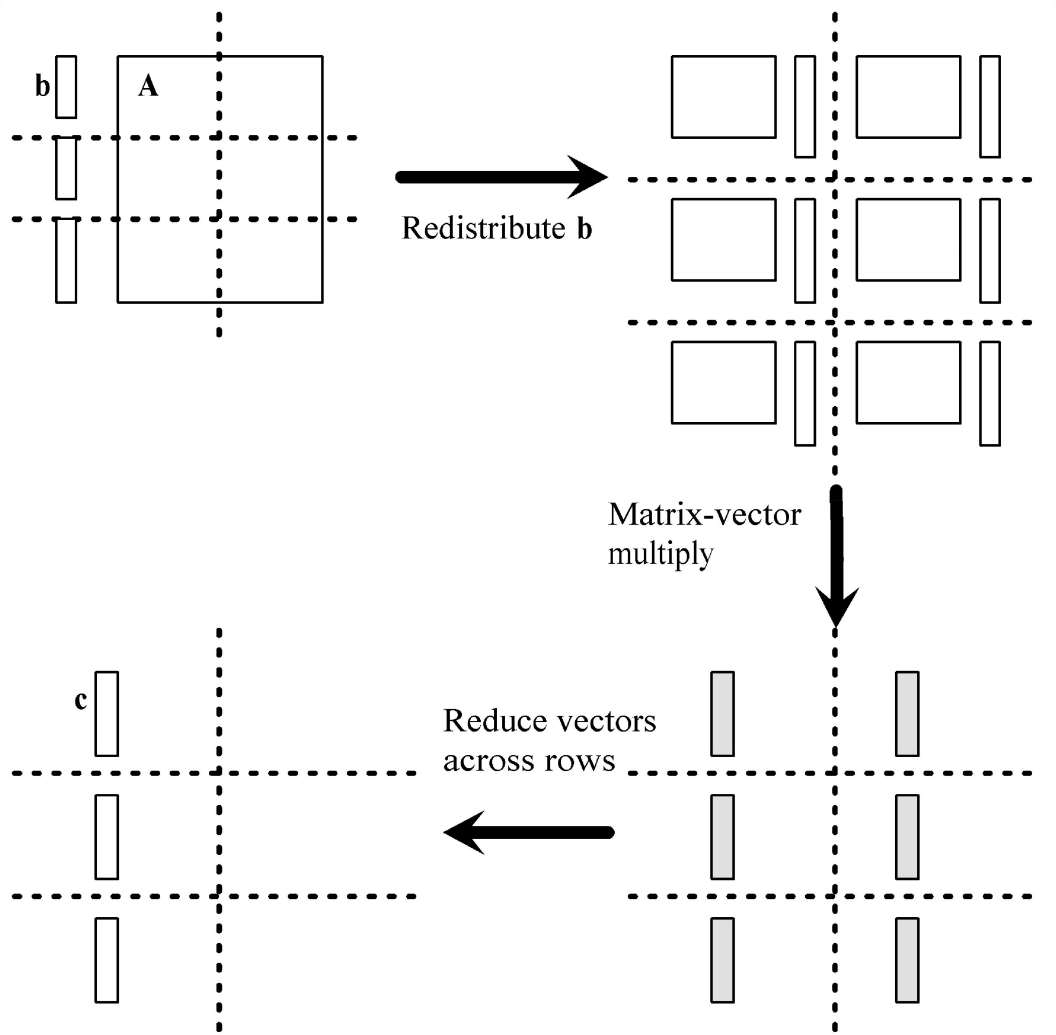
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

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:** $n^2 \geq Cn\sqrt{p}\log p \Rightarrow n \geq C\sqrt{p}\log p$
- **Scalability**

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

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

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

Scalability Analysis

- **Scalability**
  - **Rowwise:** $C^2 p$
  - **Columnwise:** $C^2 p$
  - **Checkerboard:** $C^2 \log^2 p$
- **The first two algorithms will eventually use up the per-node memory, if we want to maintain the efficiency with increasing problem size and computing resources**

Rowwise: Run-Time Expression

- **Inner product loop iteration time:** $\chi = T(n,1)/n^2$
- **Computational time:** $\chi \, n \lceil n/p \rceil$
- **Communicational time**
  - **(hypercube for all-gather)**
  - **All-gather requires** $\lceil \log p \rceil$ **messages**
  - **Total vector elements transmitted:** $(2^{\lceil \log p \rceil} - 1)\, n / 2^{\lceil \log p \rceil}$
- **Total execution time:**

$$\chi \, n \lceil n/p \rceil + \lambda \lceil \log p \rceil + 8/\beta \, (2^{\lceil \log p \rceil} - 1)\, n / (2^{\lceil \log p \rceil})$$

  - λ **is the startup time of message passing**
  - β **is the bandwidth (bytes/s)**
  - **8 bytes per double floating-point number**

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

Rowwise: Benchmarking Results

	Execution Time (msec)			
p	*Predicted*	*Actual*	*Speedup*	*MFLOPS*
1	63.4	63.4	1.00	31.6
2	32.4	32.7	1.94	61.2
3	22.3	22.7	2.79	88.1
4	17.0	17.8	3.56	112.4
5	14.1	15.2	4.16	131.6
6	12.0	13.3	4.76	150.4
7	10.5	12.2	5.19	163.9
8	9.4	11.1	5.70	180.2
16	5.7	7.2	8.79	277.8

experiments on a 450MHz Pentium II cluster…
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Columnwise: Run-Time Expression

- **Computation:** $\chi\, n\lceil n/p\rceil$
- **All-to-all exchange requires**
  - p-1 **messages, each of length about** $\lceil n/p\rceil$
- **Total execution time:**

$$\chi\, n\lceil n/p\rceil + \lambda\,(p-1) + 8\beta\,(p-1)\lceil n/p\rceil$$

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

Columnwise: Benchmarking Results

	Execution Time (msec)			
p	Predicted	Actual	Speedup	MFLOPS
1	63.4	63.8	1.00	31.4
2	32.4	32.9	1.92	60.8
3	22.2	22.6	2.80	88.5
4	17.2	17.5	3.62	114.3
5	14.3	14.5	4.37	137.9
6	12.5	12.6	5.02	158.7
7	11.3	11.2	5.65	178.6
8	10.4	10.0	6.33	200.0
16	8.5	7.6	8.33	263.2

experiments on a 450MHz Pentium II cluster…
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Checkerboard: Run-time Expression

- **Redistribute** b
  - **Send/recv:** $\lambda + 8/\beta \lceil n/\sqrt{p} \rceil$
  - **Broadcast:** $\log \sqrt{p} (\lambda + 8/\beta \lceil n/\sqrt{p} \rceil)$
- **Computation:** $\chi \lceil n/\sqrt{p} \rceil \lceil n/\sqrt{p} \rceil$
  - **Suppose** p **a square number**
- **Reduce partial results:** $\log \sqrt{p} (\lambda + 8/\beta \lceil n/\sqrt{p} \rceil)$
- **Total execution time**

  $\chi \lceil n/\sqrt{p} \rceil \lceil n/\sqrt{p} \rceil + \lambda (1 + 2 \log \sqrt{p}) + 8/\beta \lceil n/\sqrt{p} \rceil (1+ 2 \log \sqrt{p})$

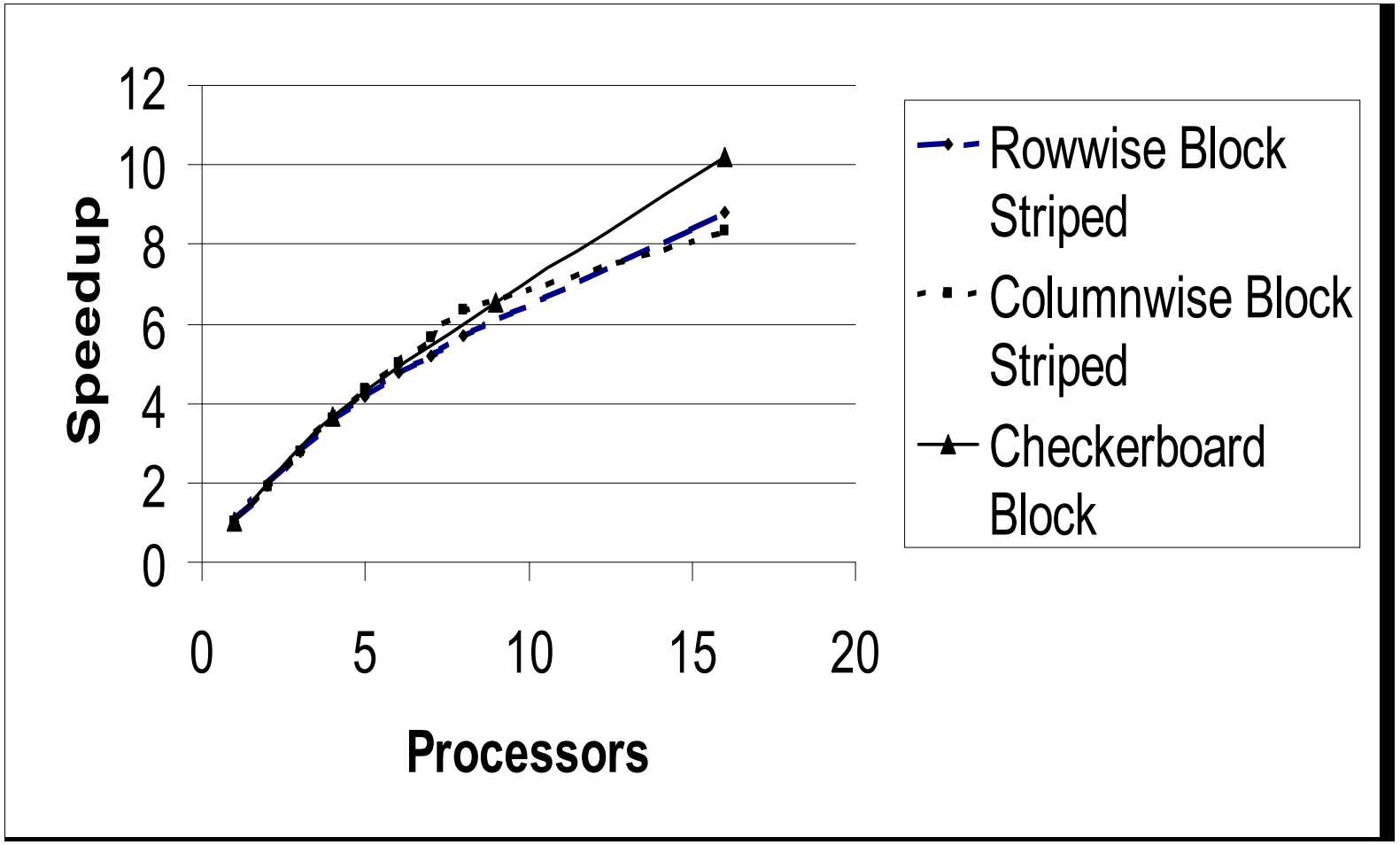
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Checkerboard: Benchmarking

Procs	Predicted (msec)	Actual (msec)	Speedup	MFLOPS
1	63.4	63.4	1.00	31.6
4	17.8	17.4	3.64	114.9
9	9.7	9.7	6.53	206.2
16	6.2	6.2	10.21	322.6

experiments on a 450MHz Pentium II cluster…
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

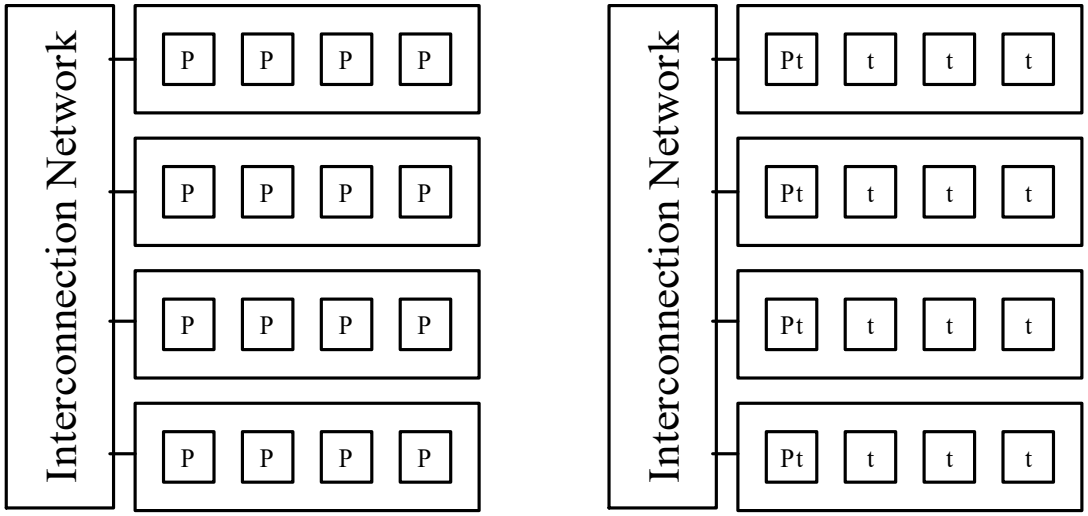
Comparison of Three Algorithms



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

MPI + OpenMP

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++) {
```

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

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]));
```

What Happened?

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

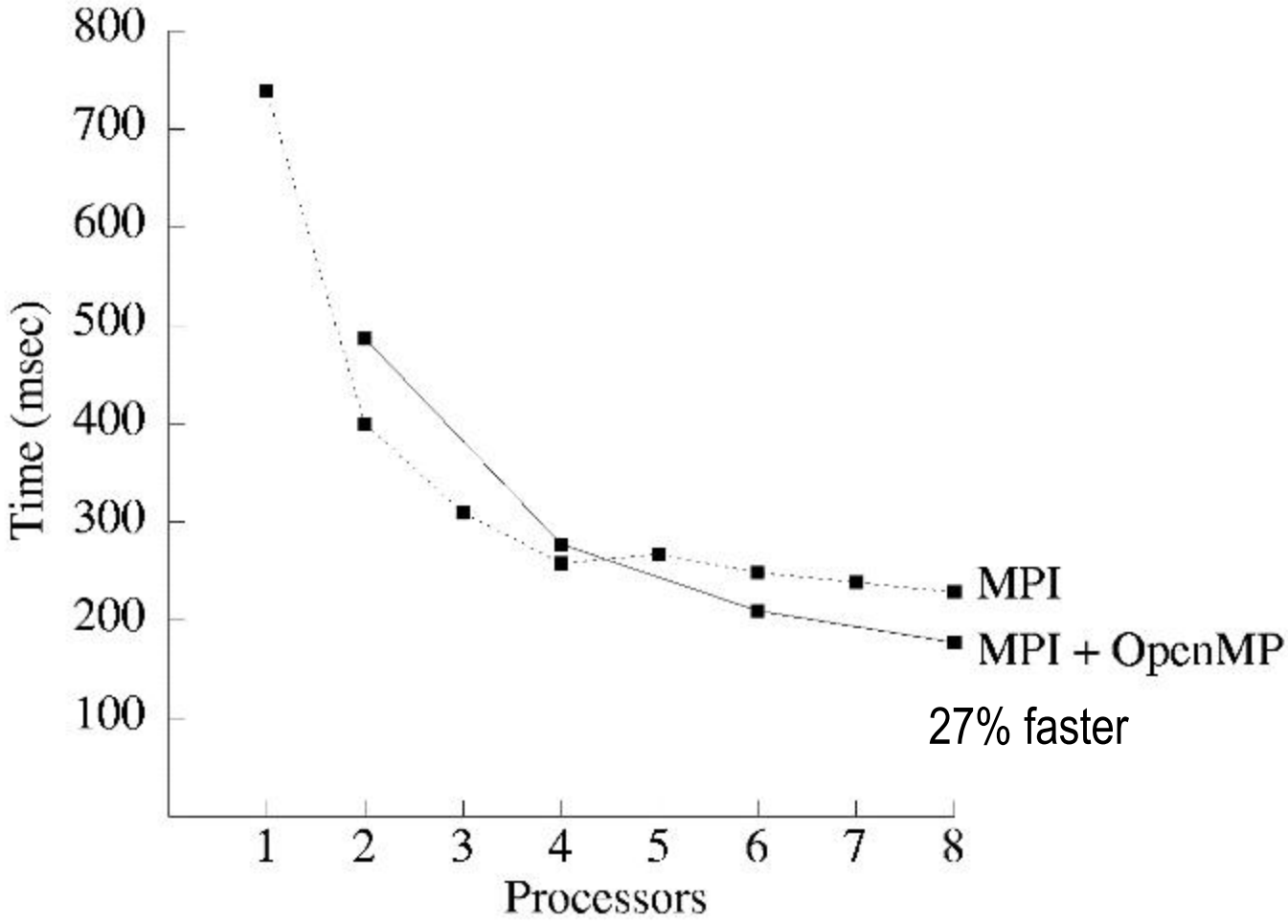
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

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.

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

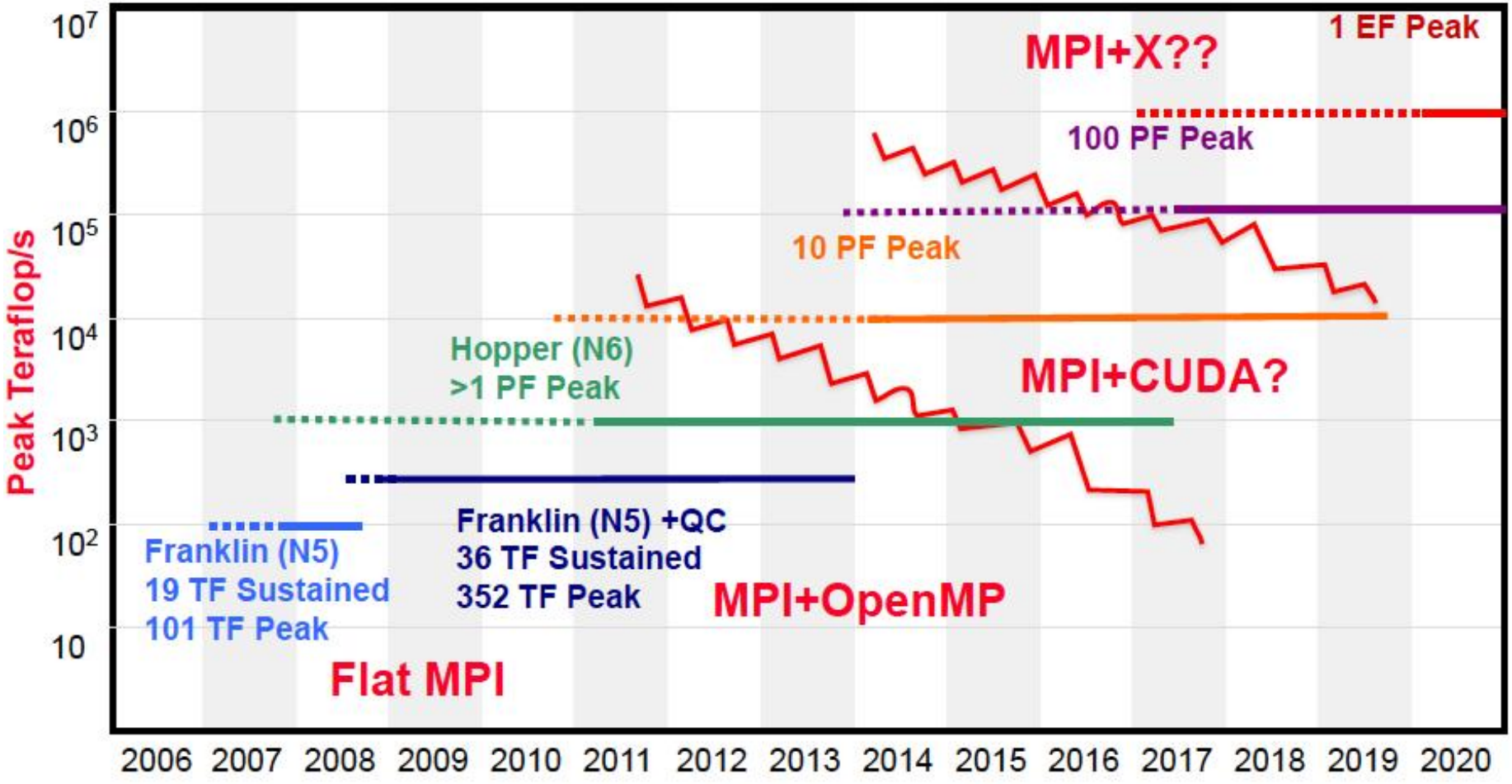
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Summary

- Scalability metric
- Performance model
- C+MPI+OpenMP programs

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

What is the Ecosystem for Exascale?



Want to avoid two paradigm disruptions *on road to Exa-scale*