High Performance Computing

高性能計算論

Volume 6

Cyberscience Center, Tohoku Univ

Hiroyuki Takizawa
<takizawa@tohoku.ac.jp>

What you learnt so far (1/2)

- **Parallel Computers**
  - Shared-memory computers
  - Distributed-memory computers
  - Hierarchical (hybrid) systems
  - Networks

What's Parallel Computer (1/3)

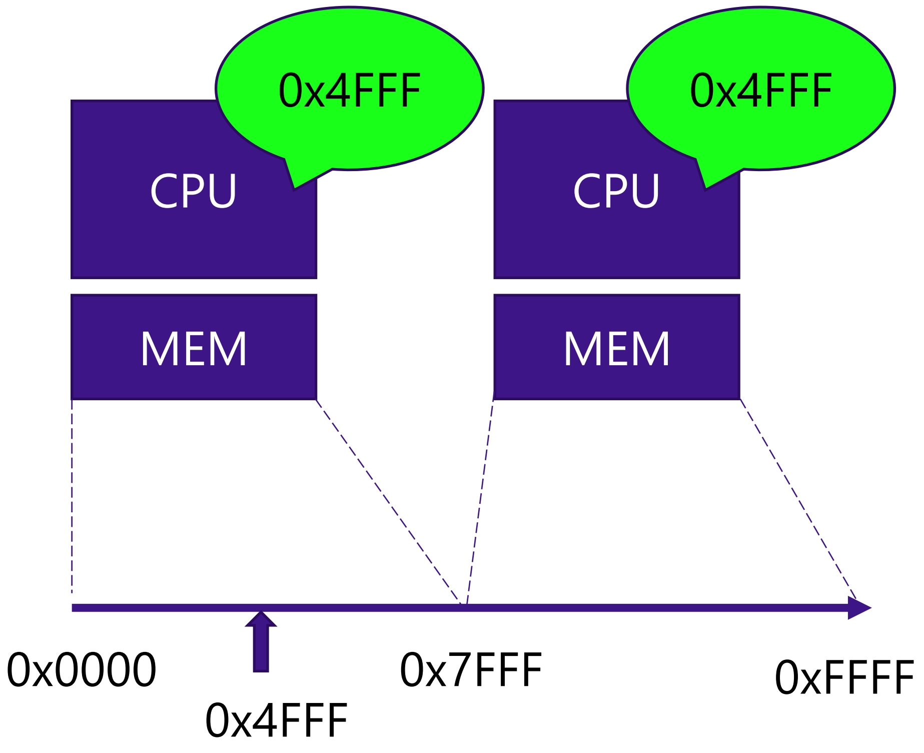
■ Parallel Computer

- A multi-processor computer system supporting parallel programming
- Two major categories of parallel computers
  - Distributed-memory parallel computers
    - Multiple computers and their interconnection network.
    - Employed to build a large-scale system
  - Shared-memory parallel computers
    - Symmetric multi-processor(SMP) and multicore/manycore.
    - Employed by most of current processors.

What's Parallel Computer (2/3)

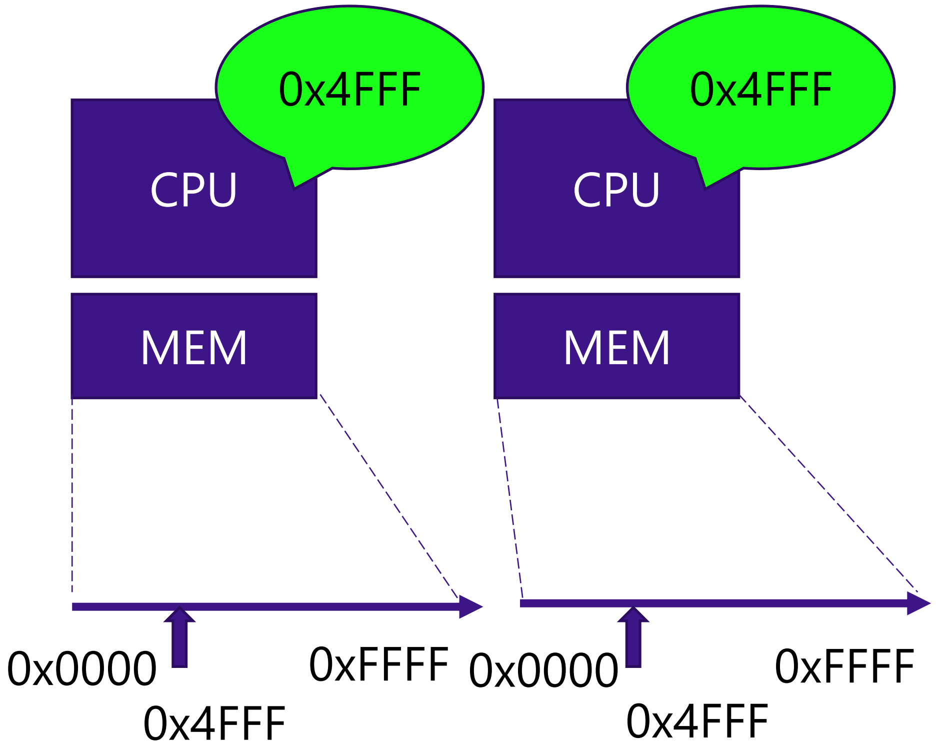
Shared-memory (NUMA)

Each memory device is mapped to a part of the memory space.



Distributed-memory

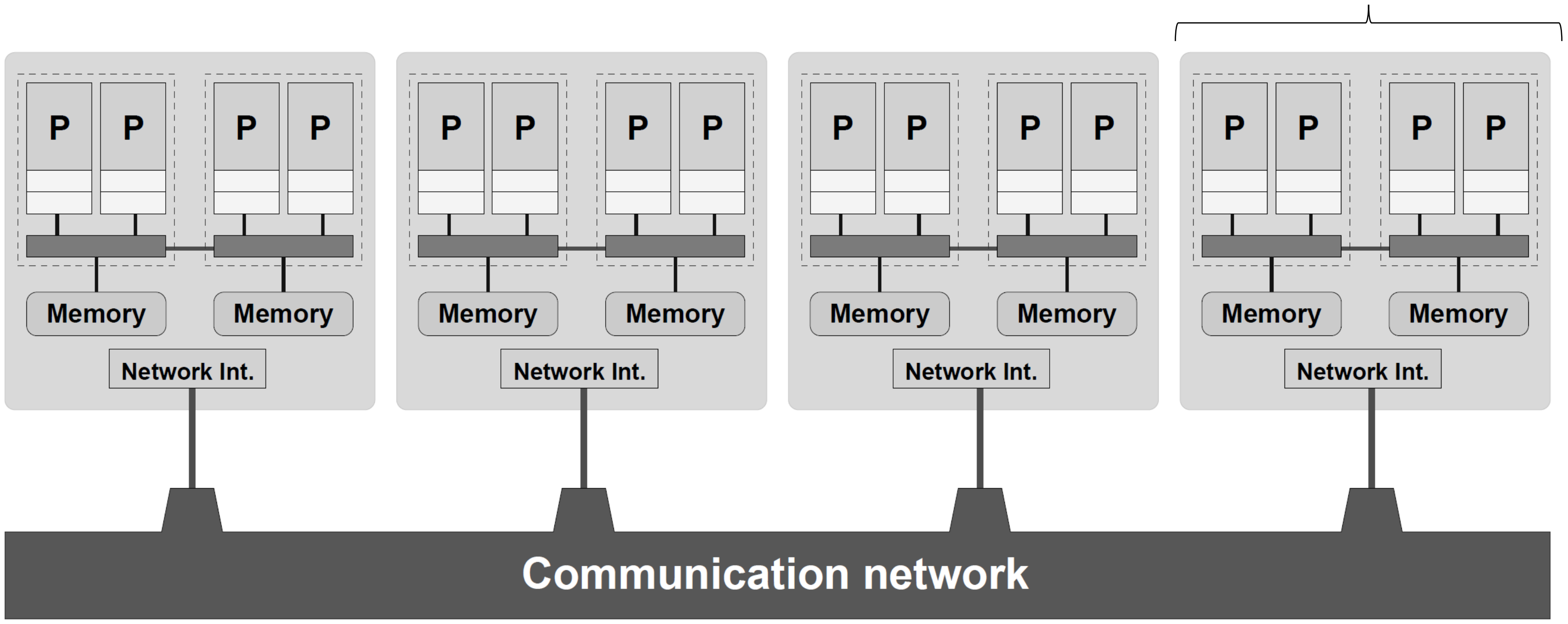
Each memory device has its own memory space.



What's Parallel Computer (3/3)

- Large-scale parallel computers
  = mixture of shared and distrib.-parallel.

One OS instance manages a node.



In addition, each node may have accelerators such as GPUs.

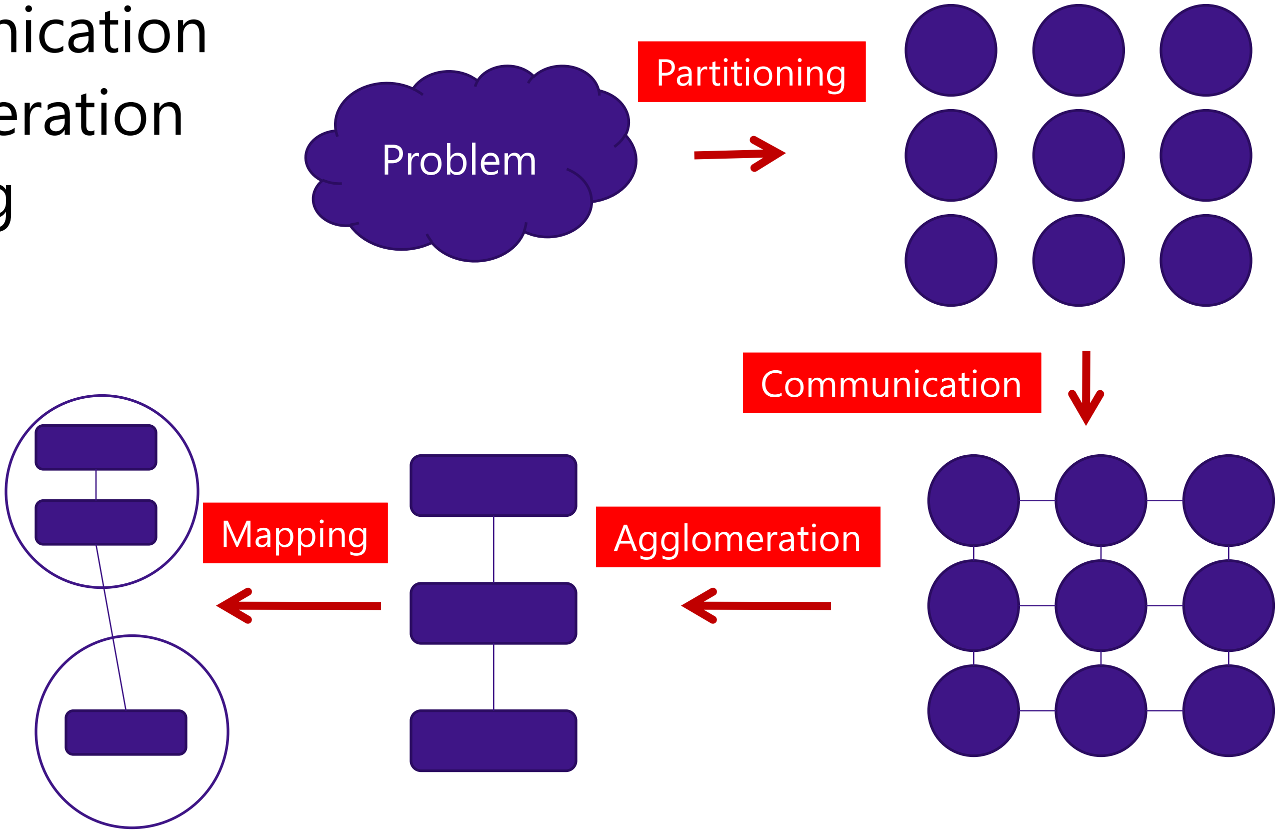
What you learnt so far (2/2)

Parallel Algorithm Design

- Task/Channel Model
- Foster’s Design Methodology
- Communication Patterns

Foster's Design Methodology

- Four steps for designing parallel algorithms
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping

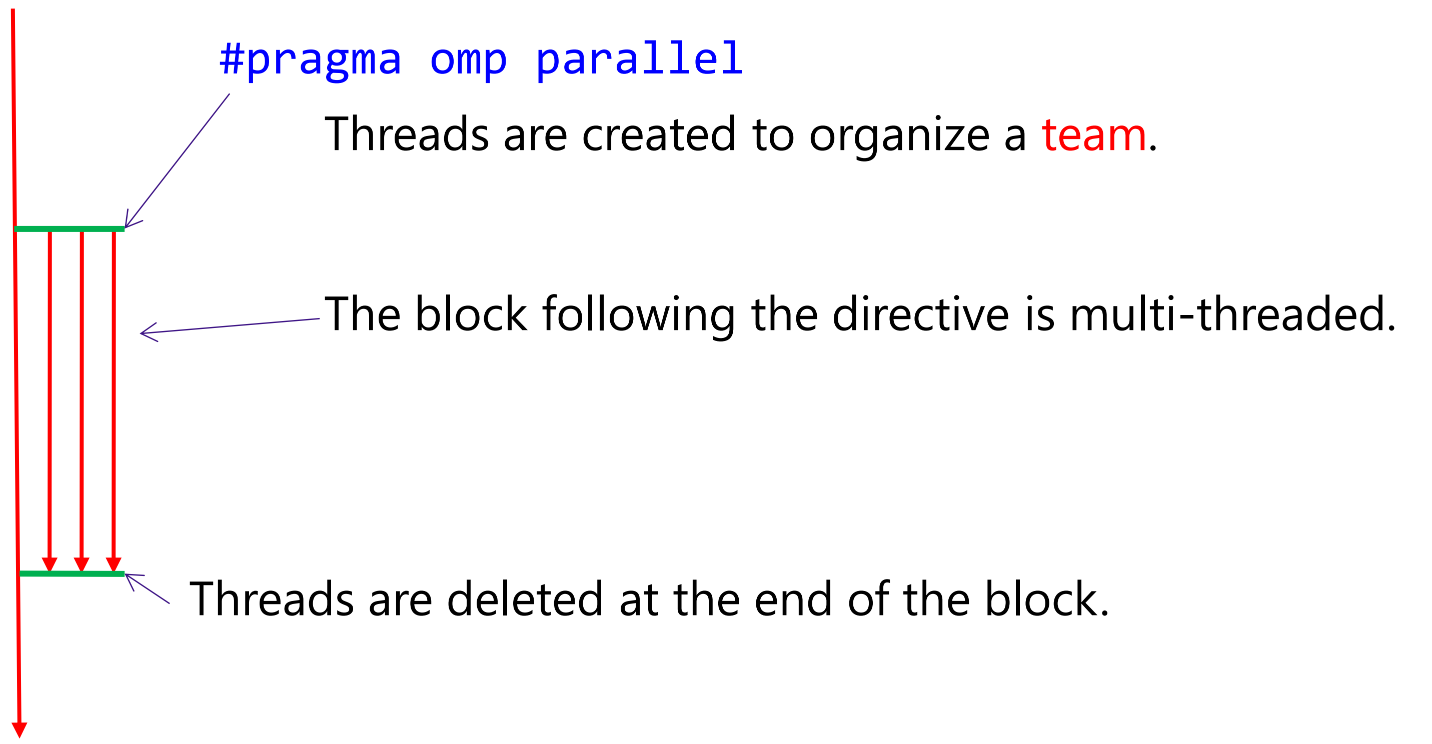


What is MPI?

- **Message Passing Interface (MPI)**
  - Interface for parallel programs with message passing.
    - Multiple programs (**MPI processes**) run on a parallel computer.
    - Each MPI process has its own memory space.
    - MPI processes can pass their data to others if necessary.
    - MPI defines only the interface (not the implementation).
      - We do not need to care about how MPI processes actually communicate.
- **Major MPI Implementations**
  - MPICH (http://www.mpich.org/)
  - Open MPI (http://www.open-mpi.org/)

What’s OpenMP?

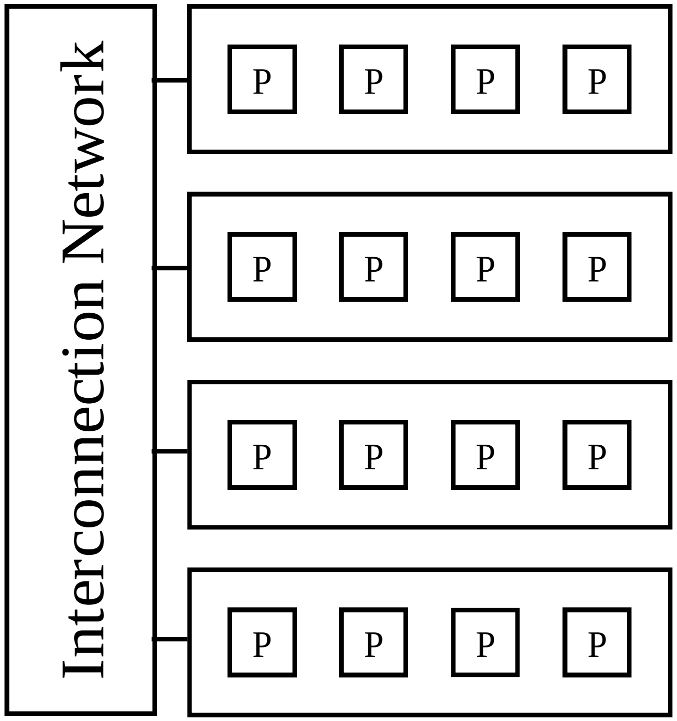
- **Threads are created/deleted on demand.**
  - Thread: an execution flow


#pragma omp parallel
Threads are created to organize a team.
The block following the directive is multi-threaded.
Threads are deleted at the end of the block.

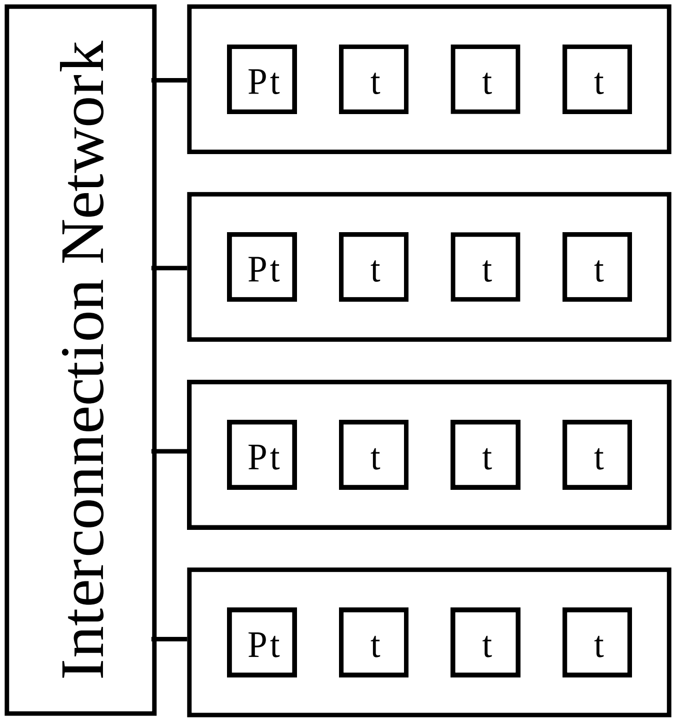
Today's Topic

- **Advantages of using both MPI and OpenMP**
  - Case Study: Conjugate gradient method
- **Performance Prediction and Analysis**
  - Roofline model (memory bandwidth)
  - Amdahl's Law (strong scaling ability)
  - Gustafson-Barsis's Law (weak scaling ability)
  - Karp-Flatt Metric (parallel performance evaluation)

C+MPI vs. C+MPI+OpenMP



C + MPI



C + MPI + OpenMP

C + MPI + OpenMP

- Lower communication overhead
- More portions of program may be practical to parallelize
- May allow more overlap of communications with computations

Case Study: Conjugate Gradient

- *A* is positive definite if for every nonzero vector x and its transpose x^T, the product $x^TAx > 0$
- If *A* is symmetric and positive definite, then the function

$$q(x) = \tfrac{1}{2} x^T A x - x^T b + c$$

  has a unique minimizer that is solution to $Ax = b$
- Conjugate gradient is an iterative method that solves $Ax = b$ by minimizing $q(x)$

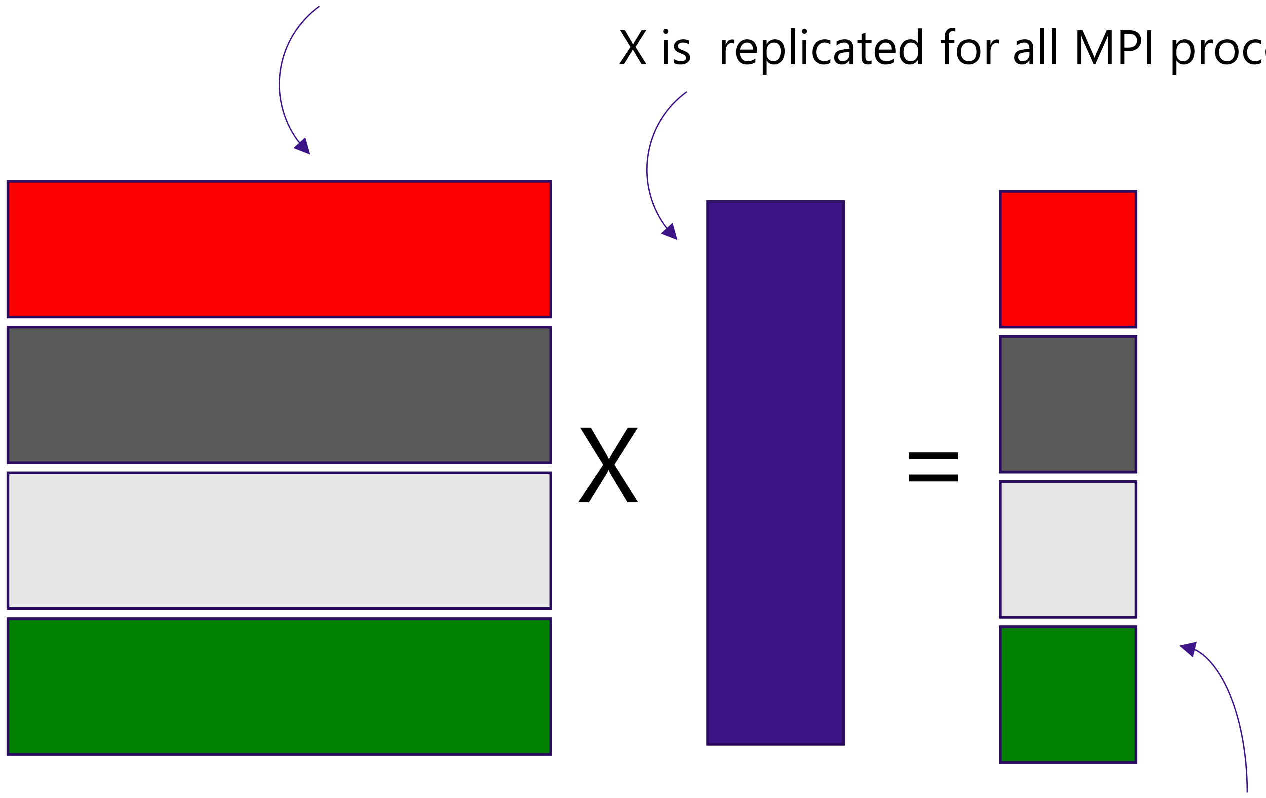
Case Study: Conjugate Gradient

- **Conjugate gradient method solves $Ax = b$**
- **In our program we assume A is dense**
  - Matrix-vector multiplication
  - Inner product (dot product)
    - Matrix-vector multiplication has higher time complexity
- **Methodology**
  - Start with a sequential program
  - Profile functions to determine where most execution time spent
  - Tackle most time-intensive function first

MPI Parallelization

A is decomposed and distributed over MPI processes

X is replicated for all MPI processes



Allgather communication is needed for all MPI processes to have the entire resultant vector (=new X).

Performance Profiling

- Compiler with –pg option
- Execute the program
- Use the gprof command to see the performance profile

```
$ mpicc -pg cg.c main.c MyMPI.c –o cg
$ mpirun –np 1 ./cg a-huge b-huge
$ ls
MyMPI.c a-huge b-huge cg cg.c gmon.out main.c
$ gprof ./cg | less

… Performance Profile Information …
```

Result of Profiling MPI Program

Clearly our focus needs to be on function **`matrix_vector_product`**

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
99.97    110.41   110.41      530     0.21     0.21  matrix_vector_product
 0.05    110.46     0.05     1060     0.00     0.00  dot_product
 0.02    110.48     0.02       10     0.00    11.05  cg
 0.00    110.48     0.00     1063     0.00     0.00  my_malloc
 0.00    110.48     0.00      530     0.00     0.00  create_mixed_xfer_arrays
 0.00    110.48     0.00      530     0.00     0.00  replicate_block_vector
 0.00    110.48     0.00        2     0.00     0.00  get_size
 0.00    110.48     0.00        1     0.00     0.00  print_replicated_vector
 0.00    110.48     0.00        1     0.00     0.00  print_subvector
 0.00    110.48     0.00        1     0.00     0.00  read_replicated_vector
 0.00    110.48     0.00        1     0.00     0.00  read_row_striped_matrix
```

Code for matrix_vector_product

Lines 90-103 in cg.c

```
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);
}
```

Adding OpenMP directives

- Want to minimize fork/join overhead by paralleling 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++) {
```

We transformed a C+MPI program to a C+MPI+OpenMP program by **adding only one line** to our program!

Excercise

- Download the zip file from the Google Classroom page

```
$ scp cg-mpi.zip front:
$ ssh front
$ unzip cg-mpi.zip
$ cd cg-mpi/
$ make
$ ./gen-cg 20480 a-mat b-vec
$ qsub run.sh
$ make prof
$ qsub run.sh
$ gprof ./cg | less
$ make omp
$ qsub run.sh
```

On your PC, copy the file to front.

Login to front

Unzip the copied file on front

Compiler for default MPI execution

Generate input files

Compile for profiling

Check the profiling result

Compile with OpenMP

You can change the number of threads, and the number of processes. See run.sh.

Today's Topic

- **Advantages of using both MPI and OpenMP**
  - Case Study: Conjugate gradient method
- **Performance Prediction and Analysis**
  - Roofline model (memory bandwidth)
  - Amdahl's Law (strong scaling ability)
  - Gustafson-Barsis's Law (weak scaling ability)
  - Karp-Flatt Metric (parallel performance evaluation)

Performance Prediction and Analysis

- **Why do we need performance prediction of a parallel algorithm?**
  - Because it helps us confirm whether an expected performance has been achieved or not.
- **The purposes of performance analysis**
  - To understand the barriers to high performance
  - To predict how much improvement can be achieved by increasing the number of processors/cores.

Finding Performance Bottleneck (1/2)

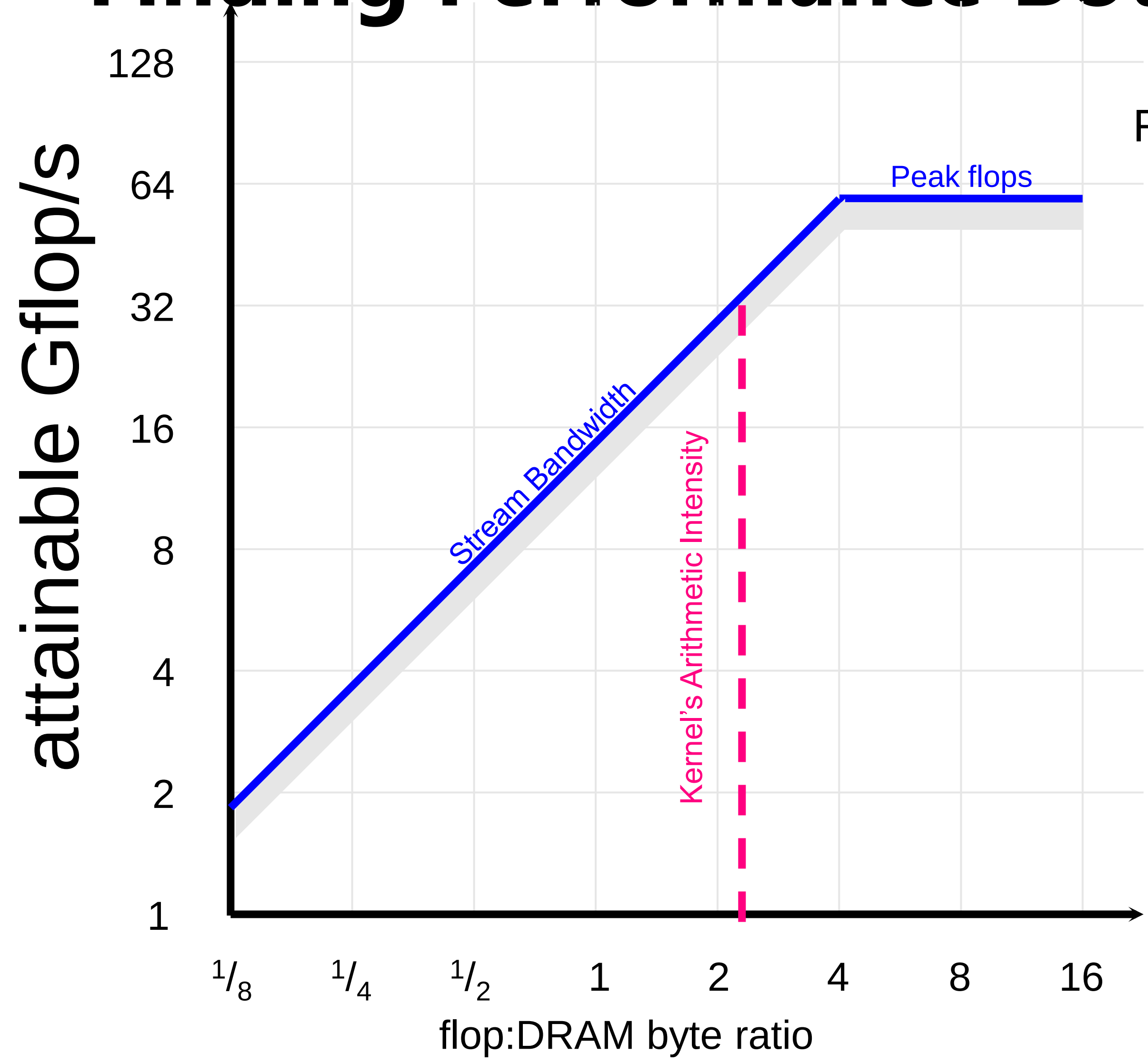
Roofline model (Williams, 2009)

- Suppose a computer of **B** [bytes/s] and **F** [flop/s]. Then, what is the expected flop/s rate on simple vector addition?
  - Assume each vector element is a 64-bit floating point value.
  - Two operands are needed for one addition, and one result is produced (Assume each is a 64-bit floating point value).
    → 24 bytes for one addition (2 inputs and 1 output).

= **Arithmetic intensity** of vector addition is 1/24

- The computer can achieve its peak flop/s rate, F, only if B is 24 times larger than F. Otherwise, the memory bandwidth limits the theoretically achievable flop/s rate (MB = performance bottleneck).

Finding Performance Bottleneck (2/2)



Roofline model (Williams, 2008)

- Machines have finite memory bandwidth
- Apply a Bound and Bottleneck Analysis
- Still Unrealistically optimistic model

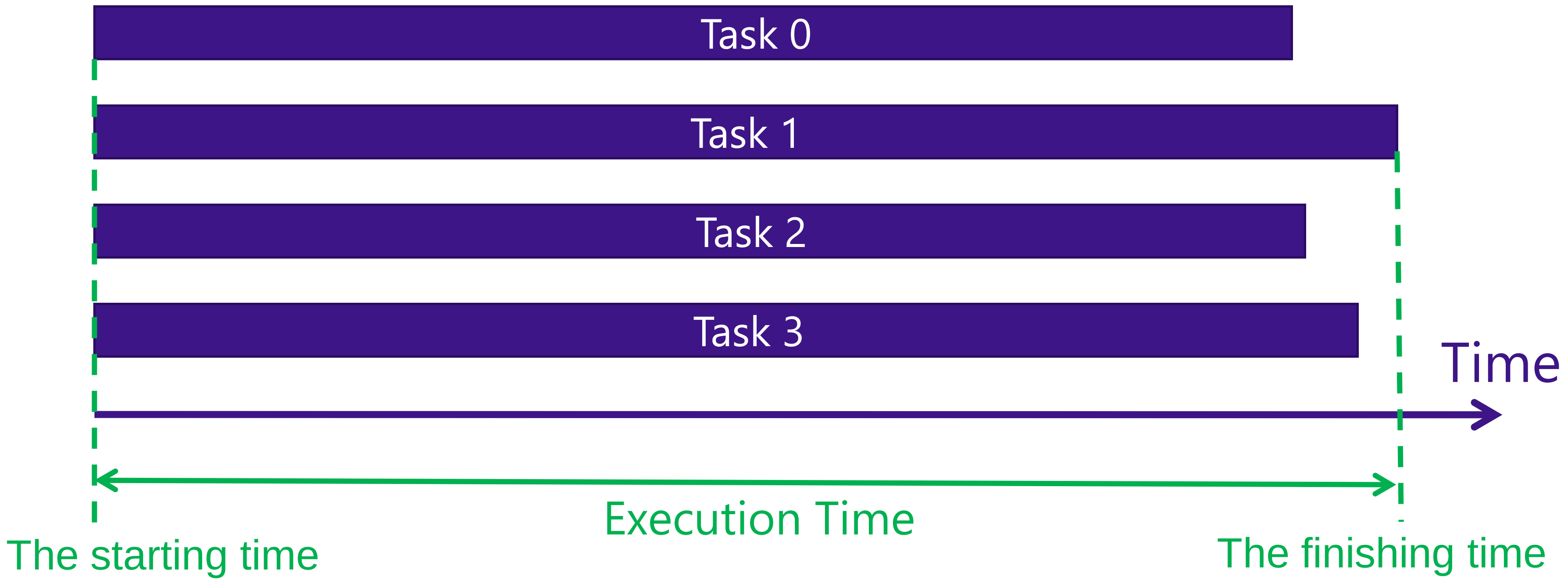
$$\textbf{Gflop/s(AI)} = \min \begin{cases} \textbf{Peak Gflop/s} \\ \textbf{StreamBW * AI} \end{cases}$$

Figure by William 2008

Execution Time

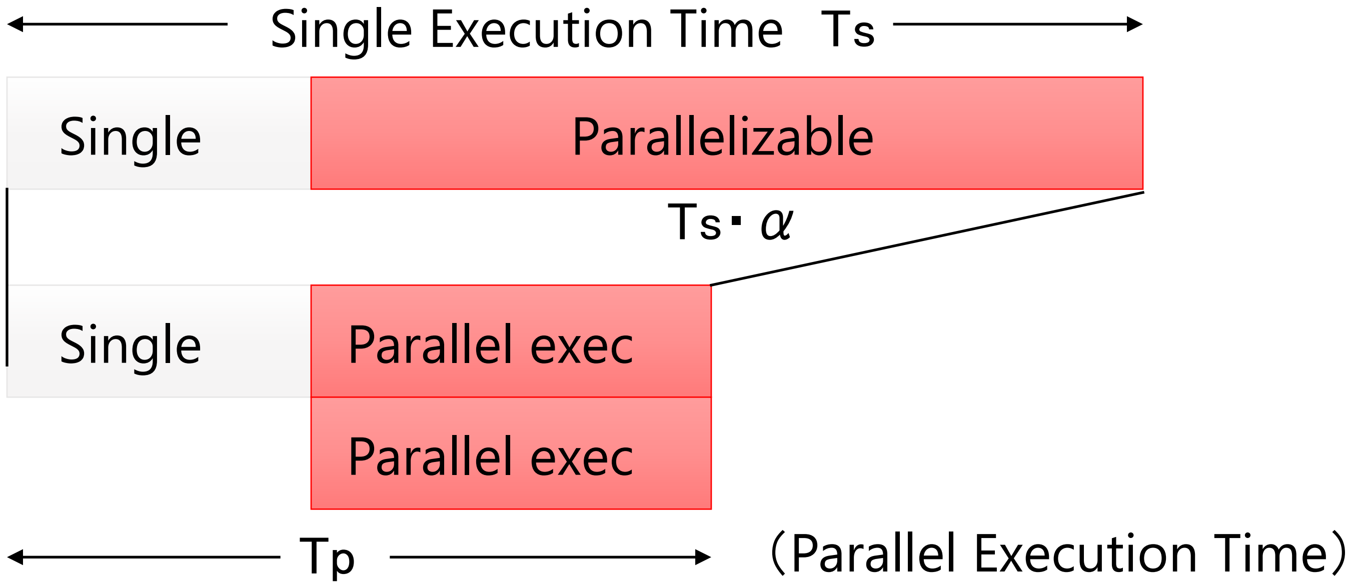
- **Execution time of a parallel algorithm:**
  - The period of time during any task is active.

    (Shorter is Better.)



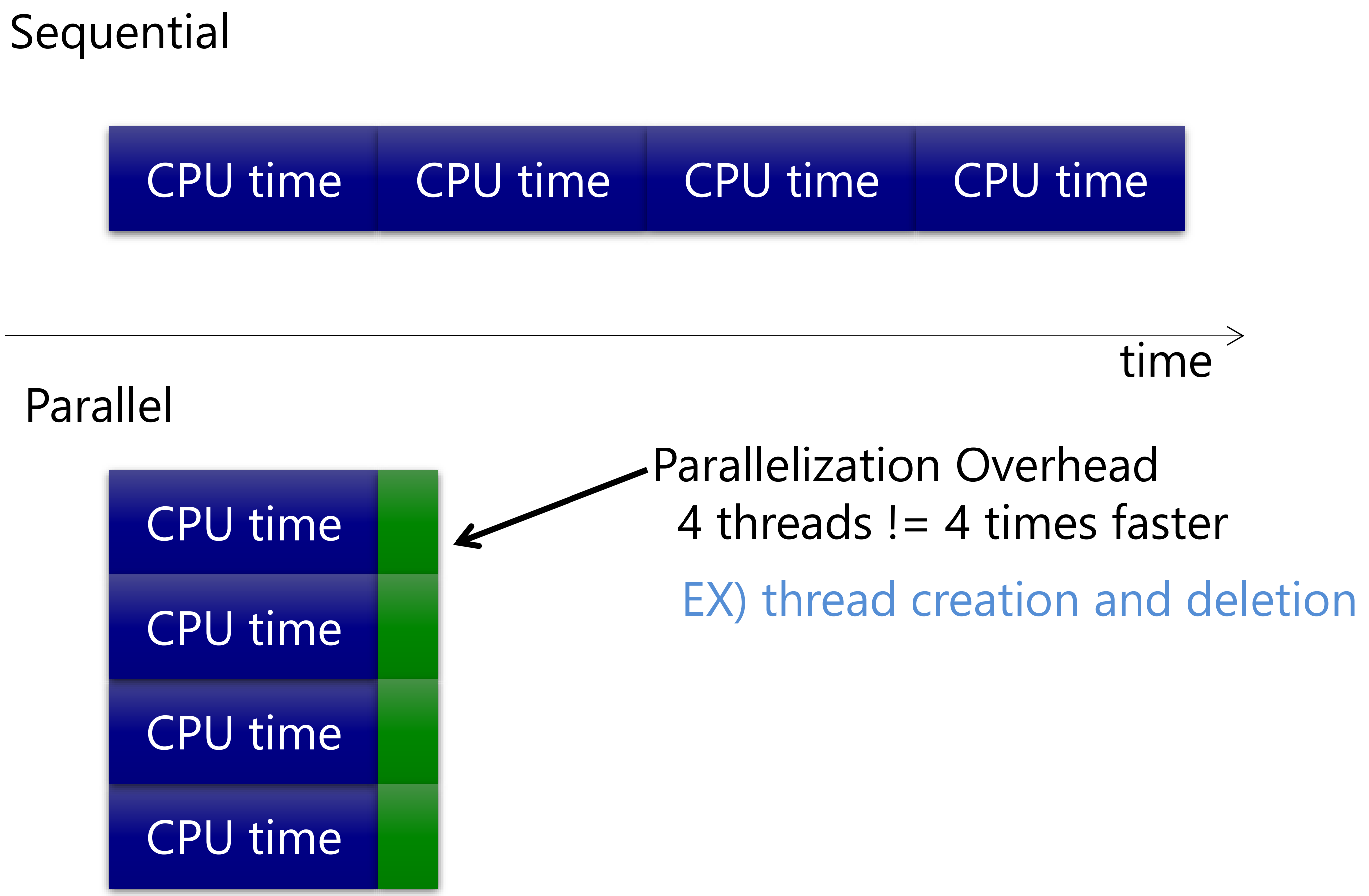
Parallelization Ratio

Parallelization Ratio α: The ratio of parallel exec.



$$\text{Speedup ratio} = \frac{Ts}{Tp} = \frac{1}{1-\alpha+\alpha/n}$$ (Amdahl's law)

Parallelization Overhead



4 threads != 4 times faster

EX) thread creation and deletion

Speedup and Efficiency

- **Speedup**
  - The ratio between sequential execution time and parallel execution time

Speedup = (Sequential Exec. Time) / (Parallel Exec. Time)

- **Efficiency**
  - A measure of processor utilization

    Efficiency = Speedup / (Number of Processors)

- **Scalability**
  - A measure of parallel system’s ability to increase the performance as the number of processor increases.

Why Ideal Speedup so difficult?

- **linear speedup is prevented by …**
  - Serial operations
  - Communication operations
  - Process start-up
  - Imbalanced workloads
  - Architectural limitations

Amdahl's Law

- **Operations performed by parallel algorithm**
  - Computations that must be performed sequentially
  - Computations that can be performed in parallel
  - Parallelization overhead
    - Communication operations and redundant computations
- **Upper bound of speedup ratio for a fixed problem size (Amdahl's law)**

$$\Psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)} \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p} = \frac{1}{f+(1-f)/p}$$

$\Psi(n,p)$	: speedup for a problem of size *n* on *p* processors
$\sigma(n)$	: sequential portion of the computation
$\varphi(n)$	: parallel portion of the computation
$\kappa(n,p)$	: parallelization overhead
p	: the number of processors
n	: the problem size
f	: the fraction of sequential computation $=\sigma(n)/(\sigma(n)+\varphi(n))$

Example 1

- 95% of a program’s execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?

Example 2

- 20% of a program's execution time is spent within inherently sequential code. What is the limit to the speedup achievable by a parallel version of the program?

Considering the overhead

Limitation of Amdahl's law

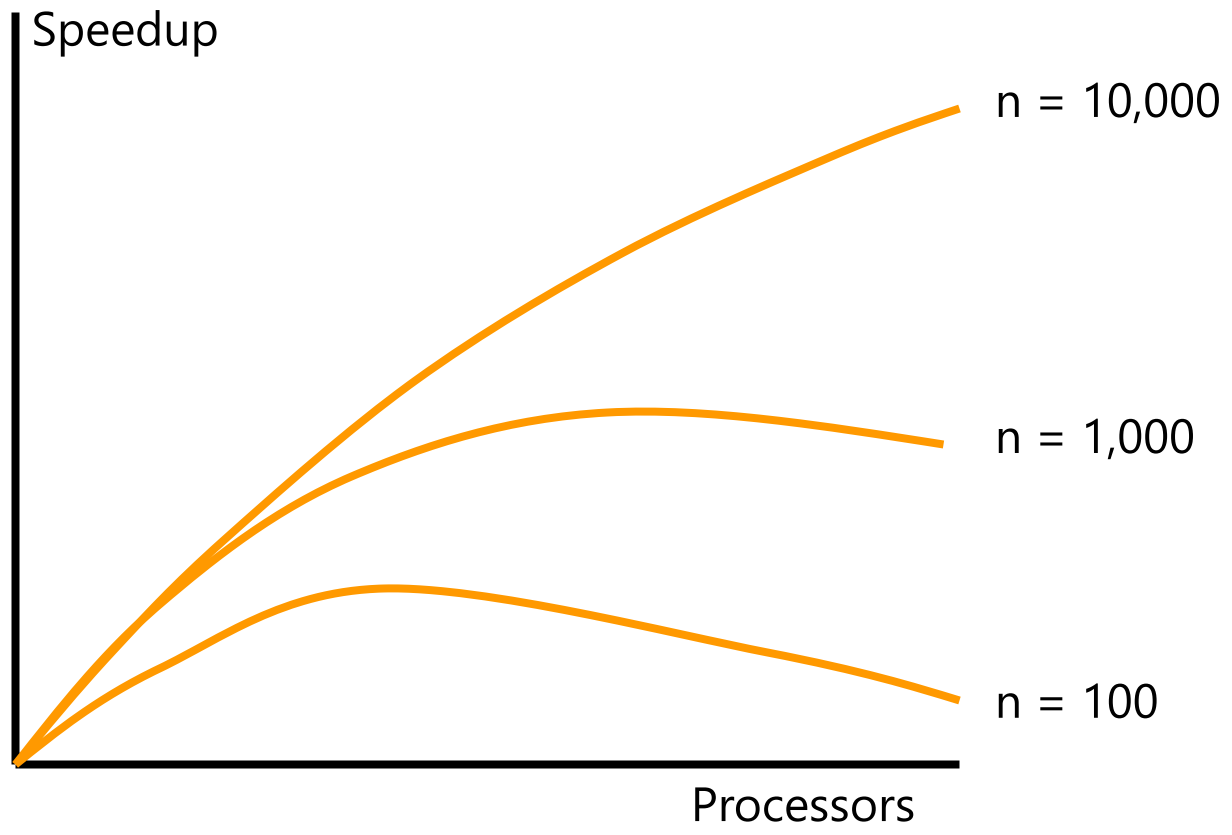
- The parallelization overhead such as communication time is ignored.
- Typically, the overhead is lower complexity than parallel portion of the computation
  - The time of parallel computation grows faster than the overhead as the problem size increases.

EX) Suppose a program

• sequential comp.	: $O(n)$ e.g. $18000+n$
• parallel comp.	: $O(n^2)$ e.g. $(n^2/100)/p$
• comm.	: $O(n + \log p)$ e.g. $10000 \,\mathrm{ceil}(\log p) + (n/10)$

Illustration of Amdahl Effect

Speedup is usually an increasing function of the problem size.



Amdahl Effect

- Typically $\kappa(n,p)$ has lower complexity than $\varphi(n)/p$
- As n increases, $\varphi(n)/p$ dominates $\kappa(n,p)$
- As n increases, speedup increases

Review of Amdahl's Law

- Treats problem size as a constant
- Shows how execution time decreases as number of processors increases

The salability with a fixed problem size is so-called **strong scalability**

Gustafson-Basis's Law

- **What happens if the time is limited and the problem size increases with the number of processors?**
  - Amdahl's effect: sequential fraction of a computation typically decreases as the problem size increases.
  - Increasing the number of processors enables us to increase the problem size solved in the time.
- **Solving a problem of size n using p processors (=parallel execution), and s is the fraction of execution time spent in serial part.**
  → s = σ(n)/(σ(n)+φ(n))

$$\Psi(n,p) \leq \frac{\sigma(n)+p\phi(n)}{\sigma(n)+\frac{p\phi(n)}{p}} = p+(1-p)\frac{\sigma(n)}{\sigma(n)+\phi(n)} = p+(1-p)s$$

"Scaled speedup" improves with *p* (Gustafson-Basis's Law) = generally, too optimistic!

Karp-Flatt Metric

- **Experimentally determined serial fraction e**

$$T(n,p) = \sigma(n) + \varphi(n)/p + \kappa(n,p) \longleftarrow \text{Total parallel execution time}$$

$$T(n,1) = \sigma(n) + \varphi(n) + \kappa(n,1) \longleftarrow \text{Total serial execution time}$$

Definition: experimentally determined serial fraction

$$e = (\sigma(n) + \kappa(n,p))/T(n,1)$$

$$\Rightarrow \sigma(n) + \kappa(n,p) = T(n,1)e$$

$$T(n,p) = T(n,1)e + T(n,1)(1-e)/p$$

$$T(n,1) = T(n,p)\psi(n,p)$$

$$\Rightarrow T(n,p) = T(n,p)\psi e + T(n,p)\psi(1-e)/p$$

$$\Rightarrow 1 = \psi e + \psi(1-e)/p$$

$$\Rightarrow e = \frac{1/\psi - 1/p}{1 - 1/p}$$

(smaller *e* means better parallelization)

Karp-Flatt Metric (Cont'd)

Experimentally Determined Serial Fraction

$$e = \frac{\sigma(n) + \kappa(n,p)}{\sigma(n) + \varphi(n)}$$

Inherently serial component of parallel computation + processor communication and synchronization overhead

Single processor execution time

$$e = \frac{1/\psi - 1/p}{1 - 1/p}$$

Karp-Flatt Metric (Cont’d)

- **Experimentally determined serial fraction is useful because …**
  - Takes into account parallelization overhead
  - Detects other sources of overhead or inefficiency ignored in speedup model
    - Process startup time
    - Process synchronization time
    - Imbalanced workload
    - Architectural overhead

Example 1

p	2	3	4	5	6	7	8
ψ	1.8	2.5	3.1	3.6	4.0	4.4	4.7

What is the primary reason for speedup of only 4.7 on 8 CPUs?

e	0.1	0.1	0.1	0.1	0.1	0.1	0.1

Since *e* is constant, large serial fraction is the primary reason.

Example 2

p	2	3	4	5	6	7	8
ψ	1.9	2.6	3.2	3.7	4.1	4.5	4.7

What is the primary reason for speedup of only 4.7 on 8 CPUs?

e	0.070	0.075	0.080	0.085	0.090	0.095	0.100

Since *e* is steadily increasing, overhead is the primary reason.

Today's Topic

- **Advantages of using both MPI and OpenMP**
  - Case Study: Conjugate gradient method
- **Performance Prediction and Analysis**
  - Roofline model (memory bandwidth)
  - Amdahl's Law (strong scaling ability)
  - Gustafson-Barsis's Law (weak scaling ability)
  - Karp-Flatt Metric (parallel performance evaluation)