Basic Computational Biology

High Performance Computing Technology(2)

Introduction to parallel programming
how to program the parallel computers

M. Sato

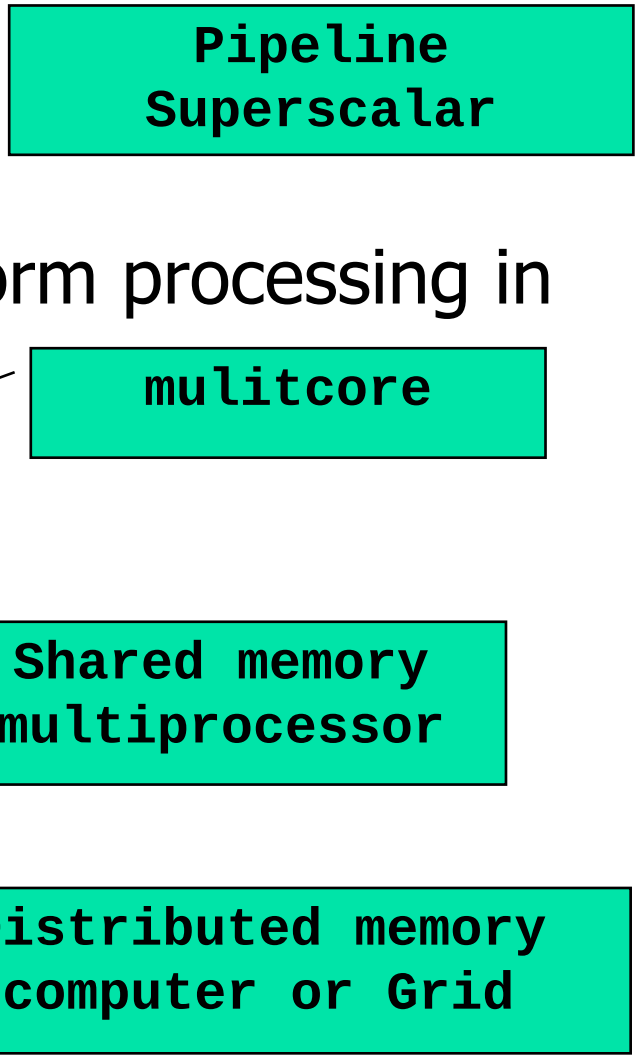
Contents

- What is parallel programming?
- Parallel Programming
  - MPI between nodes
  - OpenMP within nodes
- Programming for GPU (CUDA and OpenACC)
- Map-reduce & Cloud Computing

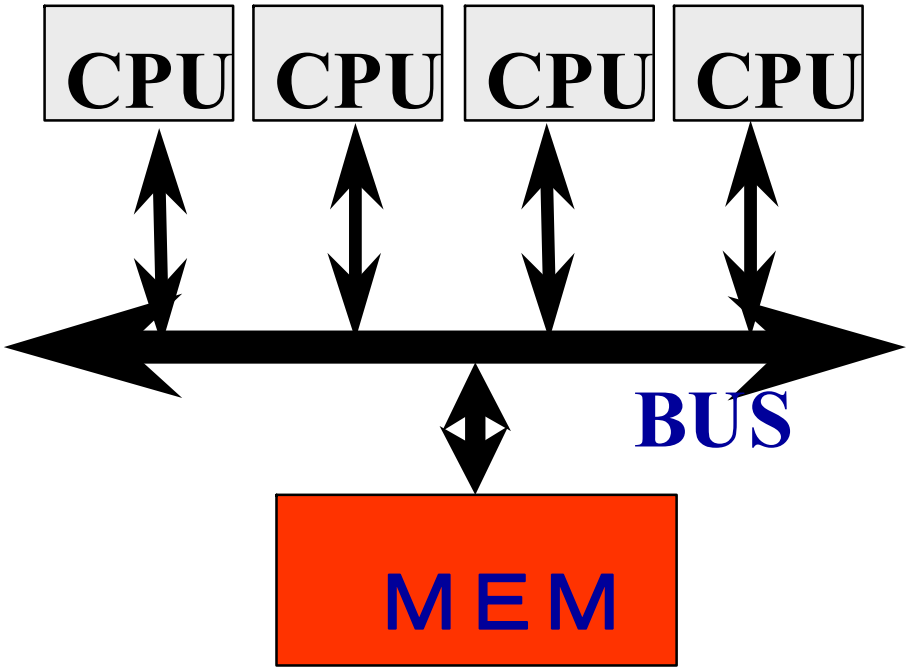
How to make computer fast?

- Computer became faster and faster by
  - Device
  - Computer architecture

- Computer architecture to perform processing in parallel at several levels:
  - Inside of CPU (core)
  - Inside of Chip
  - Between chips
  - Between computer

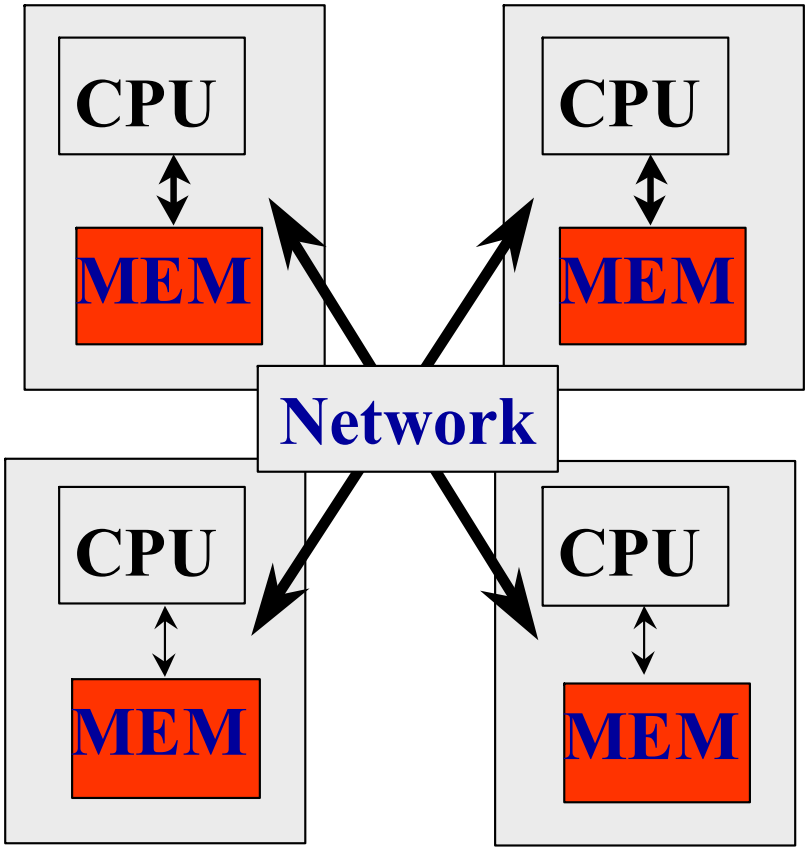


Shared memory multi-processor system



- **Multiple CPUs share main memory**
- **Threads executed in each core(CPU) communicate with each other by accessing shared data in main memory.**
- **Enterprise Server**
- **SMP Multi-core processors**

Distributed memory multi-processor



- **System with several computer of CPU and memory, connected by network.**
- **Thread executed in each computer communicate with each other by exchanging data (message) via network.タ**
- **PC Cluster**
- **AMP Multi-core processor**

Parallel computing

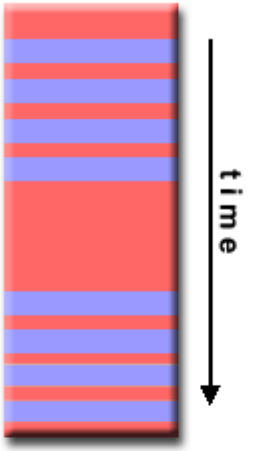
- For efficient parallel processing, certain "granularity" of parallel processing unit and enough degree of parallelisms are necessary
- Ordinary (non-scientific) applications are not sufficient to satisfy these conditions naturally
  - ex. "Word" or "Excel" applications do not have parallelism nor large amount of computation in a second
- Various scientific computations satisfy these conditions, and there are much requirement of solving these problems (especially for high-end domain science)
- Large scale parallel processing is naturally getting along with HPC
- So many numerical algorithms have been developed for scientific computation which is enable on parallel systems
- In many cases, matrix computation is essential, but direct solution is more effective in some cases

Some terminologies

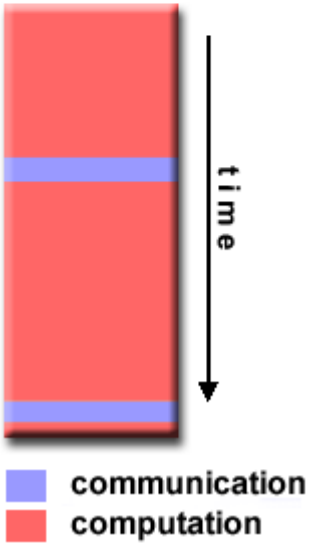
- Node – A standalone "computer in a box". Usually comprised of multiple CPUs/processors/cores. Nodes are networked together to comprise a parallel system.
- Task – A logically discrete section of computational work. A parallel program consists of multiple tasks running on multiple processors.
- Communications – Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network.
- Synchronization – The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point with an applications where a task may not proceed further until another task(s) reaches the same or logically equivalent point.

Some terminologies

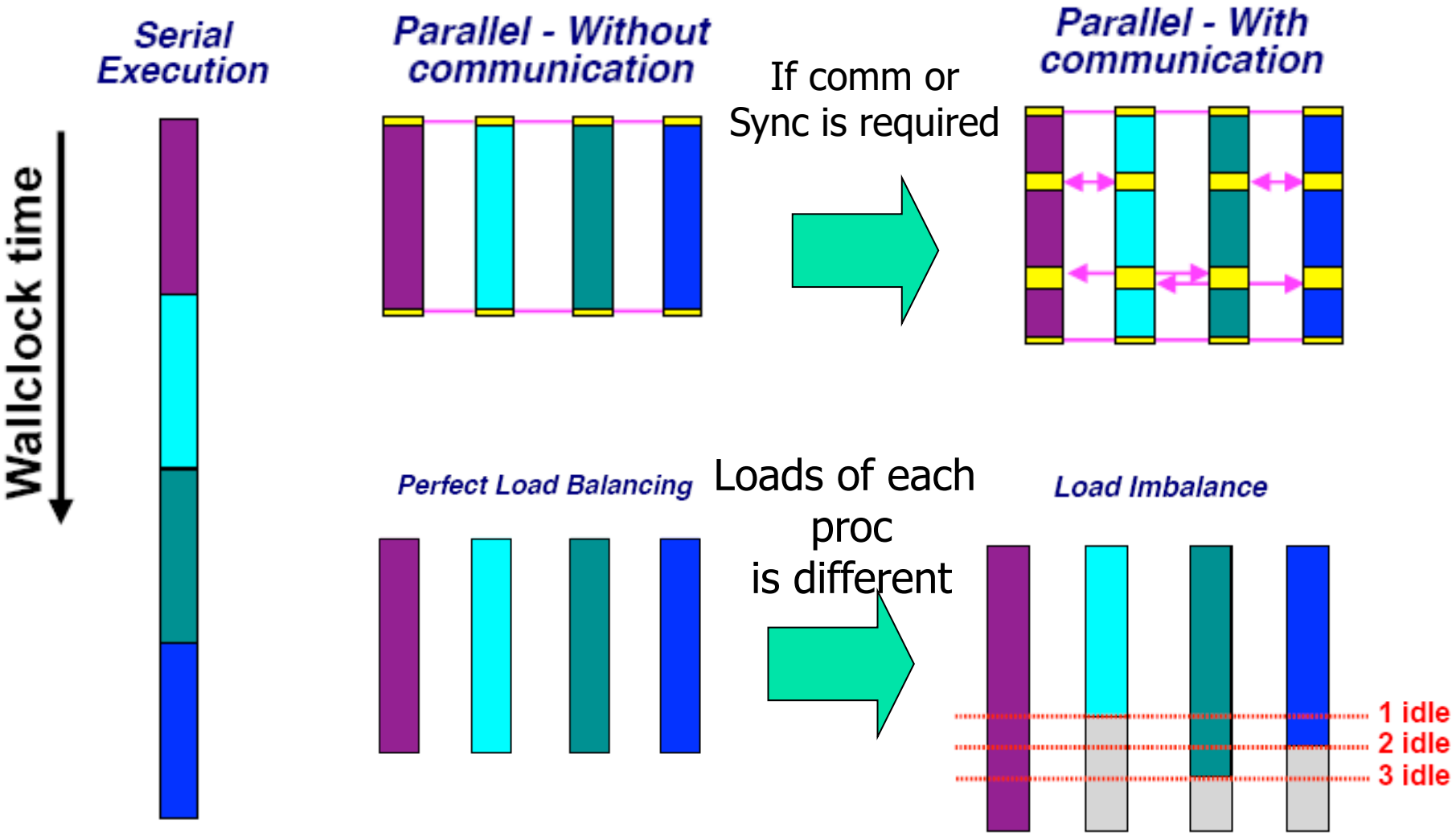
- Granularity – in parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
  - Coarse : relatively large amount of computational work are done between communication events
  - Fine: relatively samll amount of computational work are done between communication events



- Parallel overhead – The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:
  - Task start-up time
  - Synchronization
  - Data communications
  - Software overhead imposed by parallel compiler, libs, tools, ...
  - Task terminations



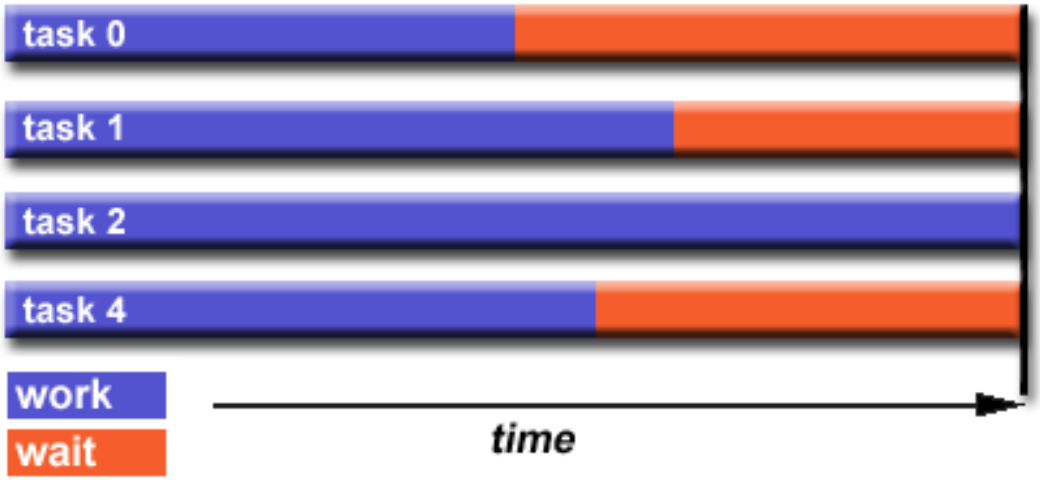
Overhead of parallel execution



Load Balancing

- Load Balancing refers to the practice of distributing work among tasks so that all tasks kept busy all of the time. It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance. For example, if all tasks are subject to a barrier sync point, the slowest task will determine the overall performance.
- How to achieve load balance:
  - Equally partition the work

    each tasks receive.
  - Use dynamic work assignment
    - Master-Worker



Some terminologies

- Scalability – Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:
  - Hardware – particularly memory-cpu bandwidth and network communications
  - Application algorithm
  - Parallel overhead related
  - Characteristics of your coding and apps.

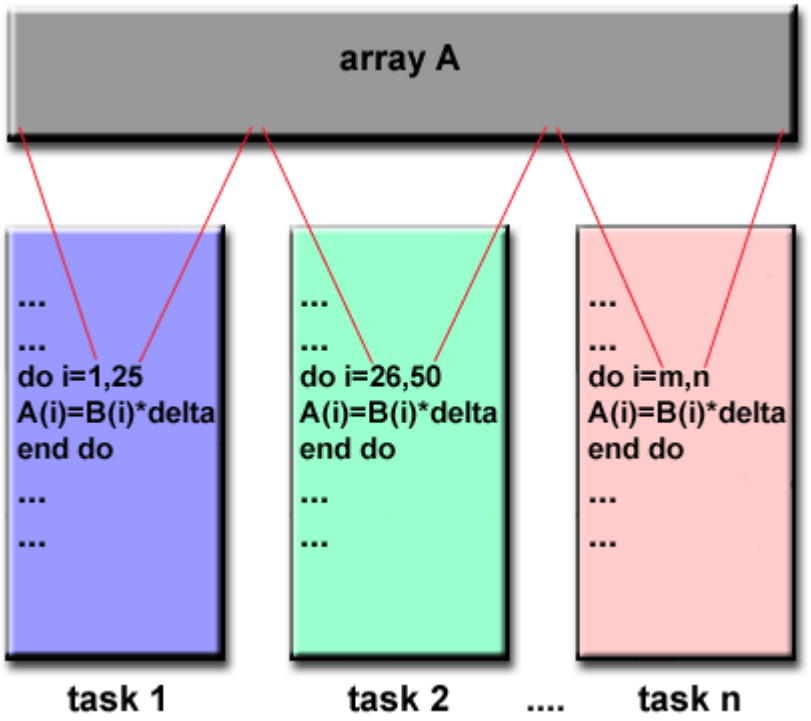
Data Parallel Model

- The data parallel models demonstrates the followings:
  - Most of the parallel work focuses on performing operations on a data sets. The data set is typically organized into common structure, such as an array or cube.
  - A set of task work collectively

on the same data structure, however, each task works on different partition of the same data structure.

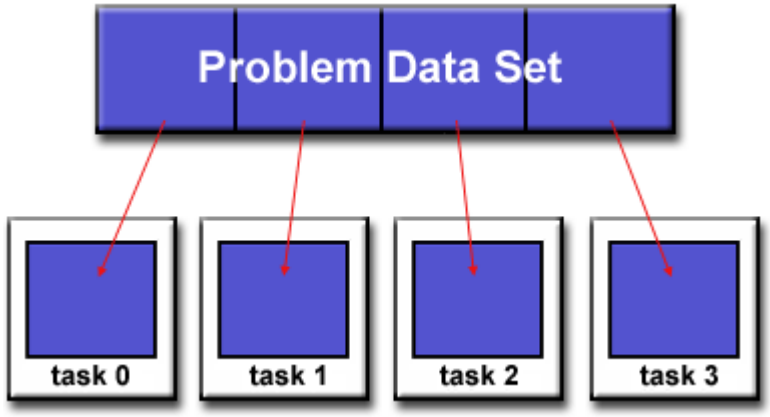
  - Tasks perform the same

operation on their partition of work

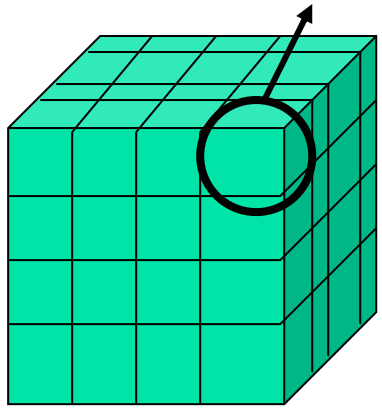


Example of data parallel model

- domain decomposition
  - Divide the space of simulation into uniform grids
  - Perform the same computation on each gird, sometimes with interaction of neighbor
  - example:



grid for computational unit



simulation space

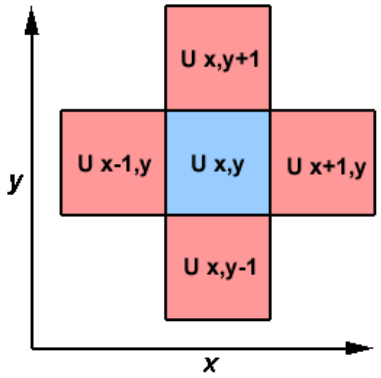
```
for(t=0; t < T; t++){
    for(i=0; i < N; i++)
        a[i] = b[i-1] + 2*b[i] + b[i+1];
    for(i=0; i < N; i++)
        b[i] = a[i];
}
```

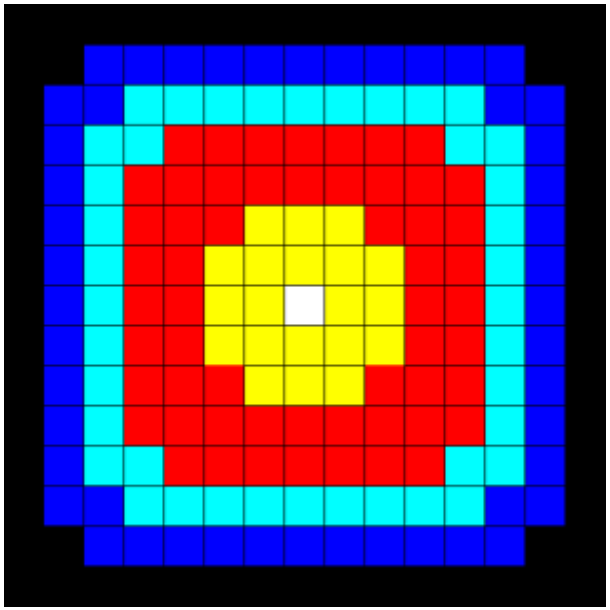
b[...] の部分で自分 (i) 以外のインデックスが出てくる

Simple Heat Equation

- Most problems in parallel computing require communication among the tasks. A number of common problem require communications "neibhbor" task. (stencil computations)
- A finite difference scheme is employed to solve the heat equations numerically on a square regions.
- For the fully explicit problem, a time stepping algorithm is used. The element of a 2-dimensional array represent the temperature at the point on the square.

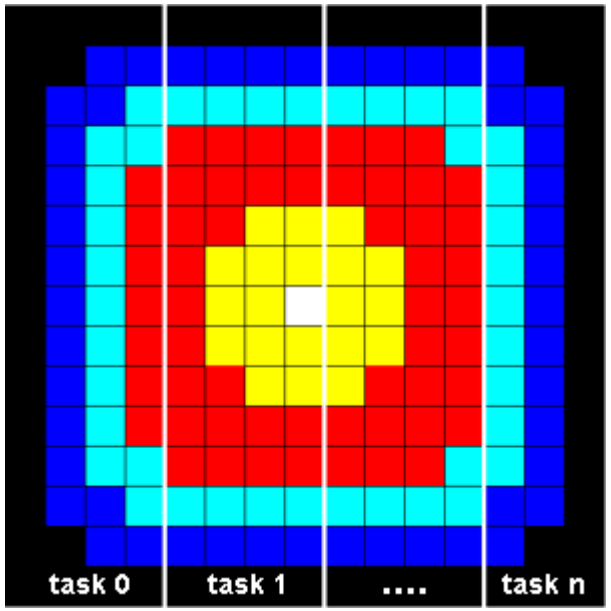
$$U_{x,y} = U_{x,y} + C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy}) + C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})$$





Simple Heat Equation

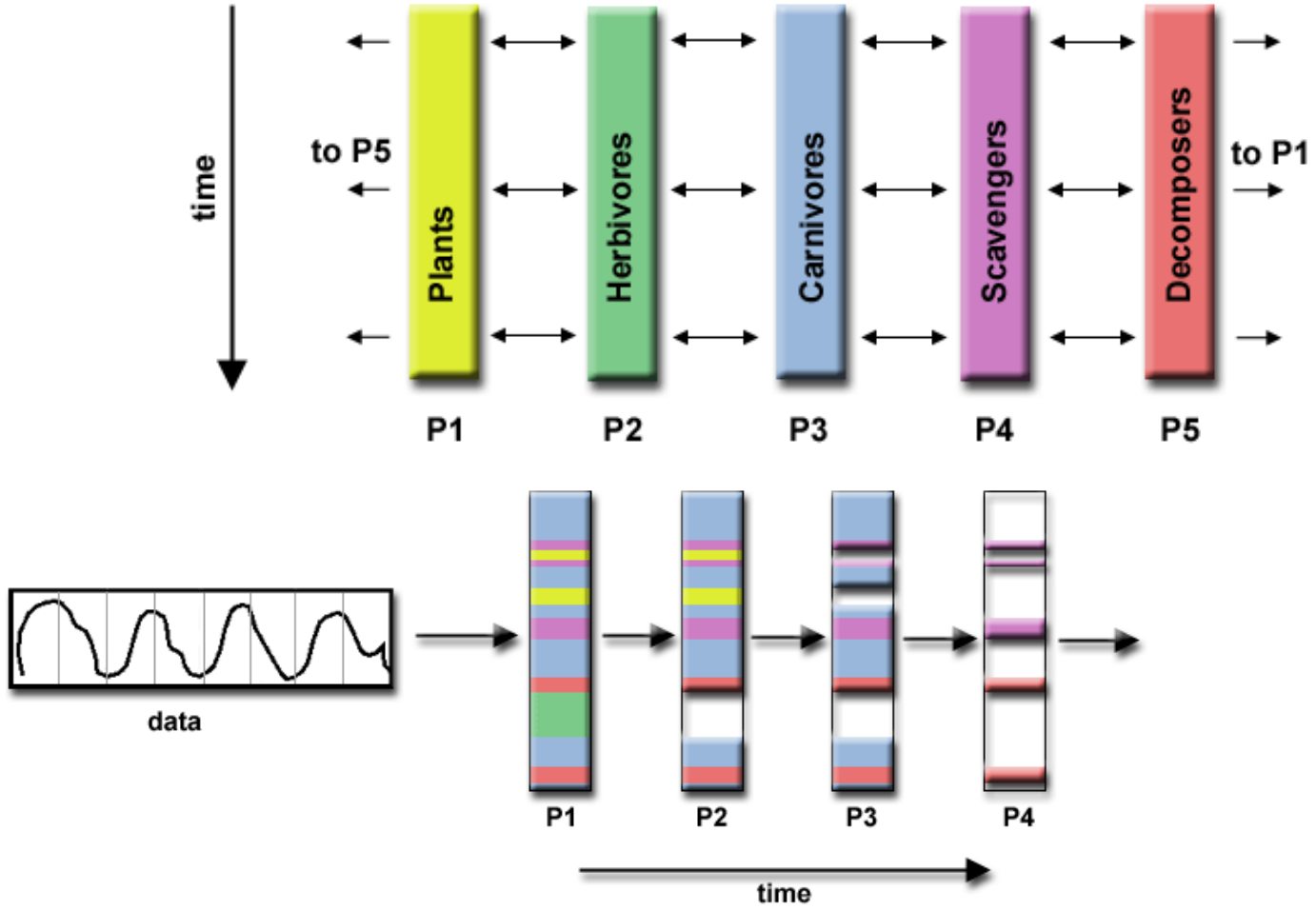
- The entire array is partitioned and distributed as subarray to all task. Each task owns a portion of the total array.
  - send slave read of u1 to neighbor processor
  - receive u1
  - compute u2 at each processor
  - update u1 with u2
  - repeat the above computation until the condition is satisfied.




```
do iy = 2, ny-1
do ix = 2, nx-1
  u2(ix,iy) = u1(ix,iy)+
      cx*(u1(ix+1),y)+u1(ix+1,iy)-2*u1(ix,iy))+
       cy*(u1(ix,iy+1)+u1(ix,iy-1)-2*(ix,iy))
end do
end do
```

Pipeline

- Breaking a task into steps performed by different processors unit, with inputs streams through, much like assemble lines
- Example: signal processing



master/worker parallel processing

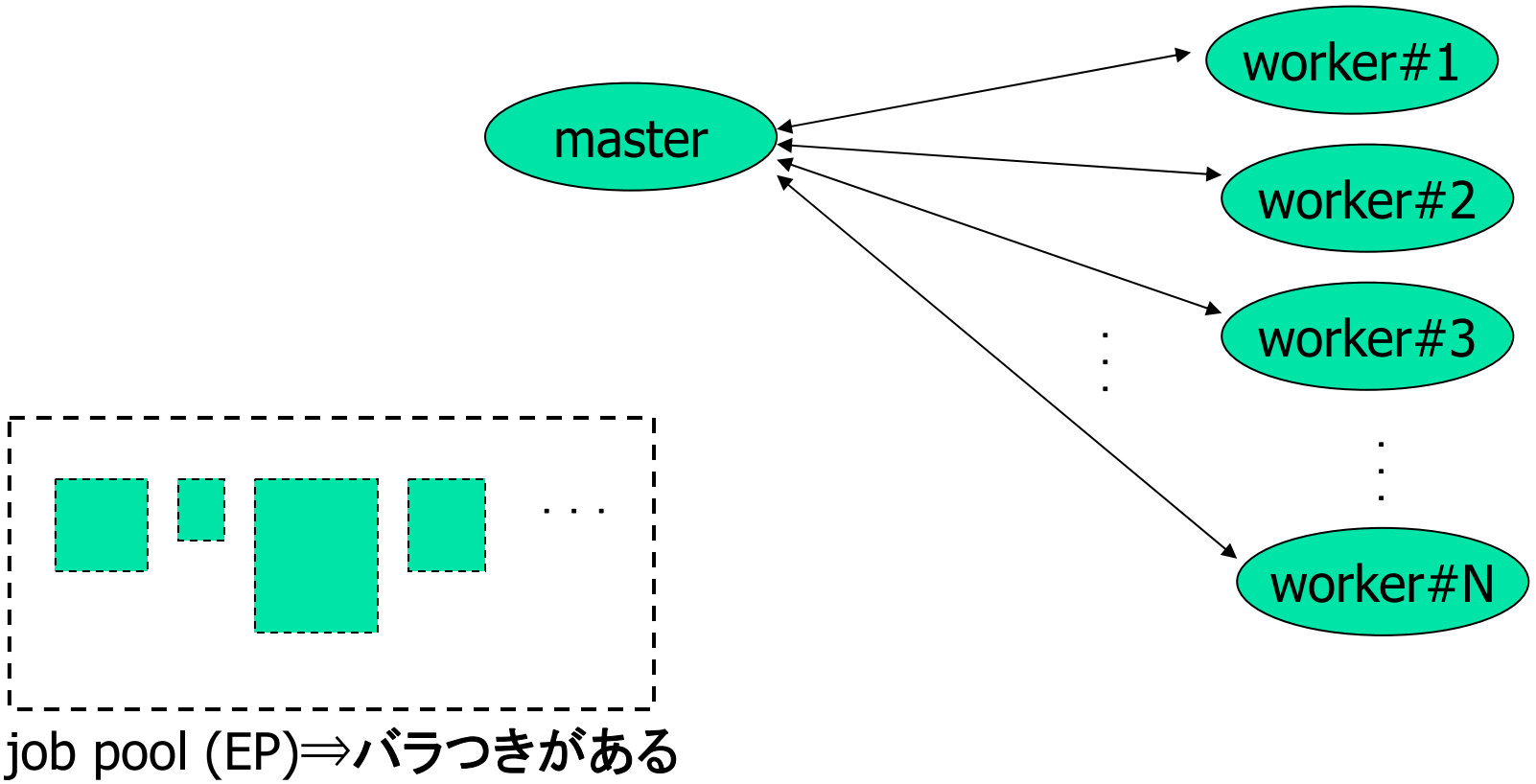
- one master processor and several worker processors
- A pool of work in master processor.
- master pick up one work to send the work to a worker.
- When worker finish the given work, then it return the result and receive next work

```
master::
// give a job to each worker
while(1){
    // receive a worker's result
    // give the next job to that worker
}
```

```
worker::
while(1){
        // receive a job from master
    // process the job
    // send the result to master
        }
```

master/worker parallel processing

- It is effective parallel processing when each work have different load --> load balancing

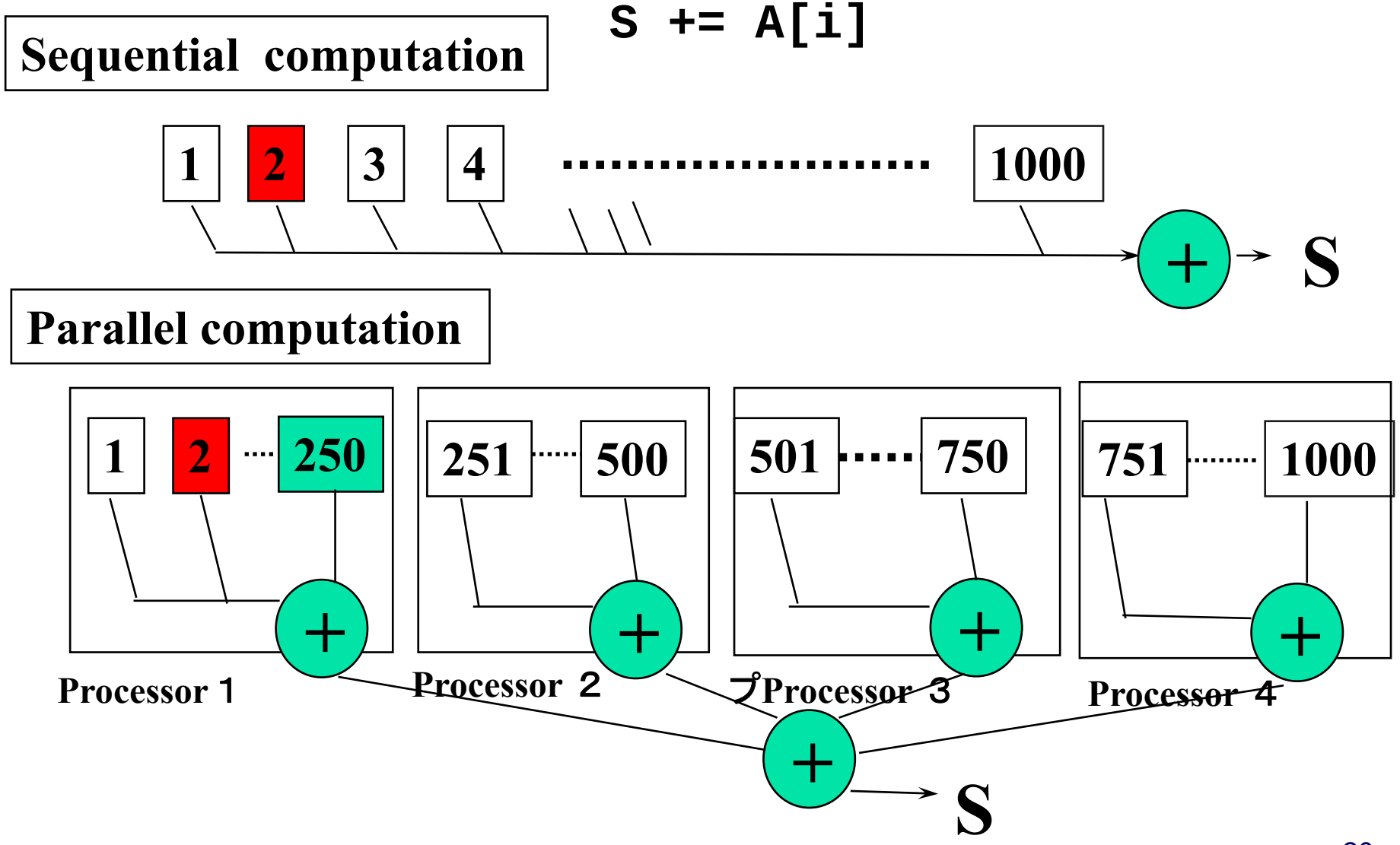


Parallel Programming

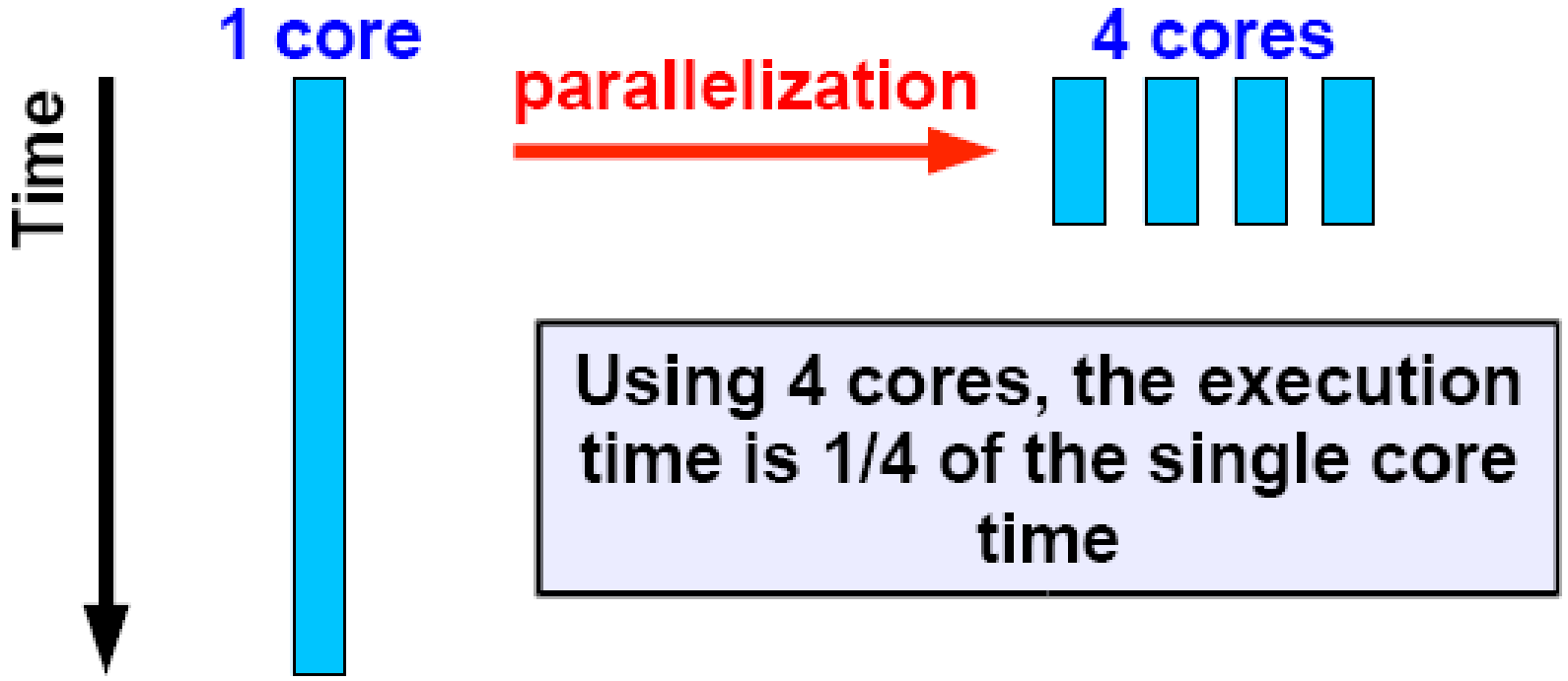
MPI & OPENMP

Very simple example of parallel computing for high performance

```
for(i=0;i<1000; i++)
    S += A[i]
```



Why parallelization needs?
4 times speedup by using 4 cores!



Parallel programming models

- *There are numerous parallel programming models*
- *The ones most well-known are:*
  - *Distributed Memory*
    - ✔ *Sockets (standardized, low level)*
    - ✔ *PVM - Parallel Virtual Machine (obsolete)*
    - ✔ *MPI - Message Passing Interface (de-facto std)*
  - *Shared Memory*
    - ✔ *Posix Threads (standardized, low level)*
    - ✔ *OpenMP (de-facto standard)*
    - ✔ *Automatic Parallelization (compiler does it for you)*

Simple example of Message Passing Programming

- Sum up 1000 element in array

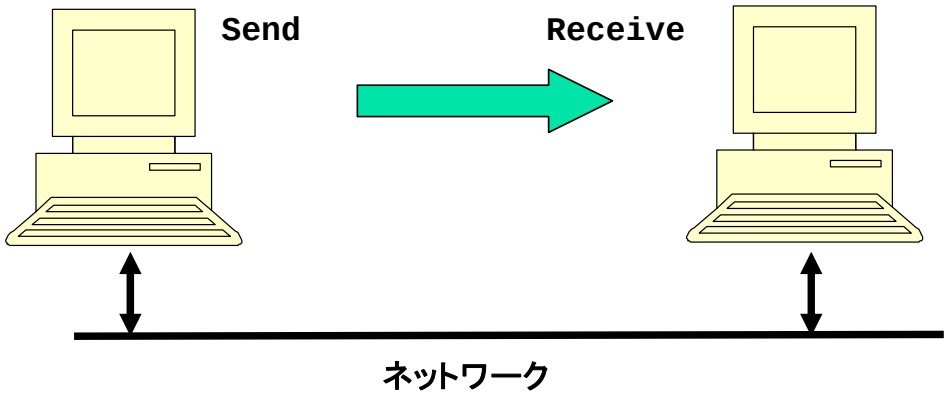
```
int a[250]; /* 250 elements are allocated in each node */

main(){    /* start main in each node */
   int i,s,ss;
   s=0;
   for(i=0; i<250;i++) s+= a[i]; /*compute local sum*/
   if(myid == 0){   /* if processor 0 */
      for(proc=1;proc<4; proc++){
         recv(&ss,proc);  /* receive data from others*/
         s+=ss;   /*add local sum to sum*/
      }
   } else {  /* if processor 1,2,3 */
      send(s,0);  /* send local sum to processor 0 */
   }
}
```

Parallel programming using MPI

- MPI (Message Passing Interface)
- Mainly, for High performance scientific computing
- Standard library for message passing parallel programming in high-end distributed memory systems.
  - Required in case of system with more than 100 nodes.
  - Not easy and time-consuming work
    - "assembly programming" in distributed programming

- Communication with message
  - point-to –point : Send/Receive
- Collective operations
  - Reduce/Bcast
  - Gather/Scatter



Communicator and rank of MPI

- A communicator specifies the process group that can send and receive messages to each other.
- Rank is a ID number within a group "communicator".
- The endpoint of communication specified by communicator and rank.
- A predefined communicator MPI_COMM_WORLD is provided by MPI.
  - It allows communication with all processes that are accessible after MPI initialization and processes are identified by their rank in it. Usually using only MPI_COMM_WORLD is enough.
- Users may define new communicators if necessary

point-to-point Comm. functions

- int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - blocking send/receive operation
  - buf: initial address of send buffer
  - count: number of elements in send buffer
  - datatype: datatype of each send buffer element
  - dest: rank of destination
  - source: rank of source
  - tag: message tag
  - comm: communicator
  - status: status object (structure MPI_Status)

Programming in MPI

```
#include "mpi.h"
#include <stdio.h>
#define MY_TAG 100
double A[1000/N_PE];
int main( int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double sum, x;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d on %s¥n", myid, processor_name);

    ....
```

Programming in MPI

```
    sum = 0.0;
    for (i = 0; i < 1000/N_PE; i++){
       sum+ = A[i];
    }

    if(myid == 0){
       for(i = 1; i < numprocs; i++){
          MPI_Recv(&t,1,MPI_DOUBLE,i,MY_TAG,MPI_COMM_WORLD,&status
            sum += t;
        }
    } else
            MPI_Send(&t,1,MPI_DOUBLE,0,MY_TAG,MPI_COMM_WORLD);
    /* MPI_Reduce(&sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM
     MPI_Barrier(MPI_COMM_WORLD);
    ...
    MPI_Finalize();
    return 0;
}
```

Collective communication

- Collective communication is defined as communication that involves a group of processes.

  - Broadcast

  - Gather

  - Allgather = Gather + Broadcast

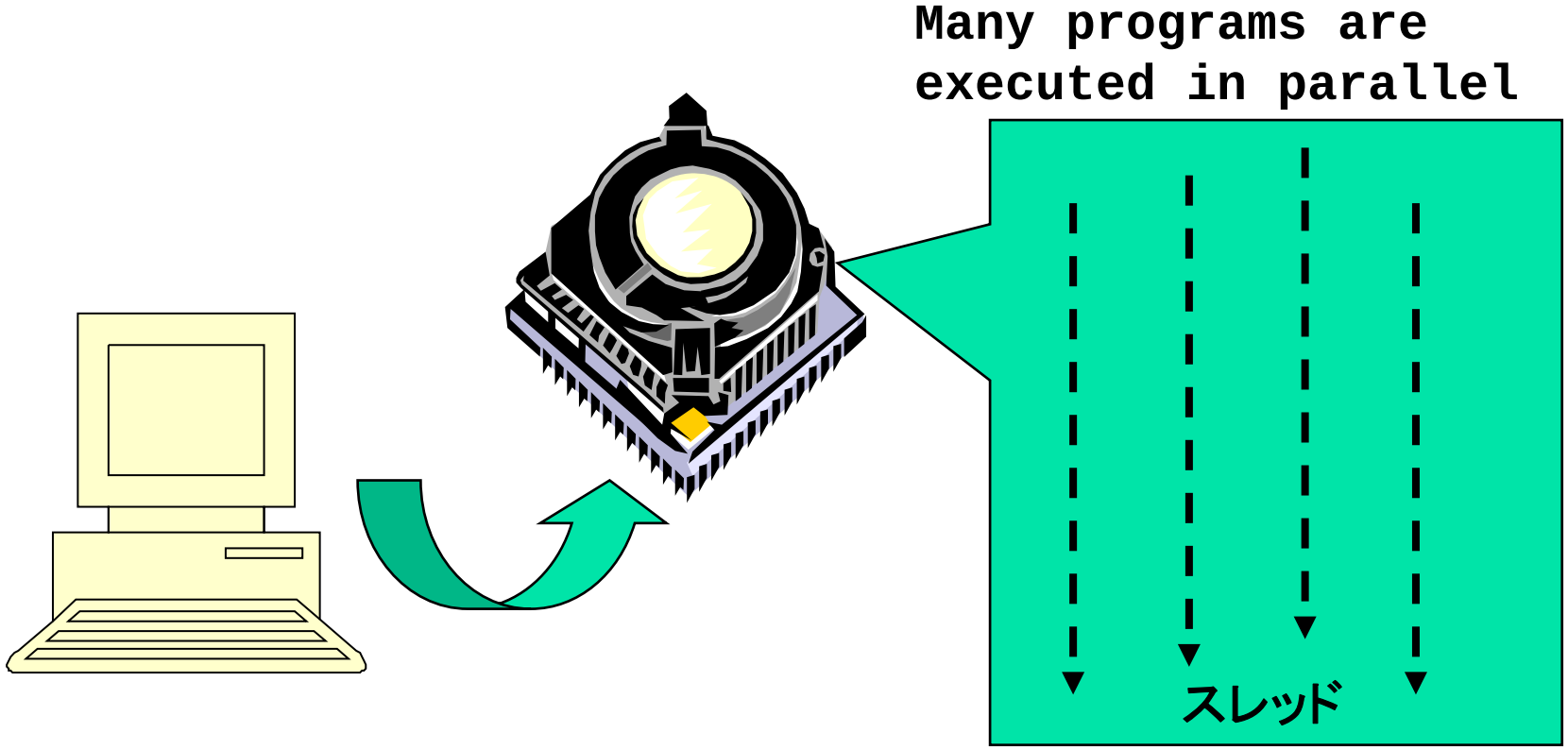
  - Scatter

Parallel programming models

- *There are numerous parallel programming models*
- *The ones most well-known are:*
  - *Distributed Memory*
    - ✔ *Sockets (standardized, low level)*
    - ✔ *PVM - Parallel Virtual Machine (obsolete)*
    - ✔ *MPI - Message Passing Interface (de-facto std)*
  - *Shared Memory*
    - ✔ *Posix Threads (standardized, low level)*
    - ✔ *OpenMP (de-facto standard)*
    - ✔ *Automatic Parallelization (compiler does it for you)*

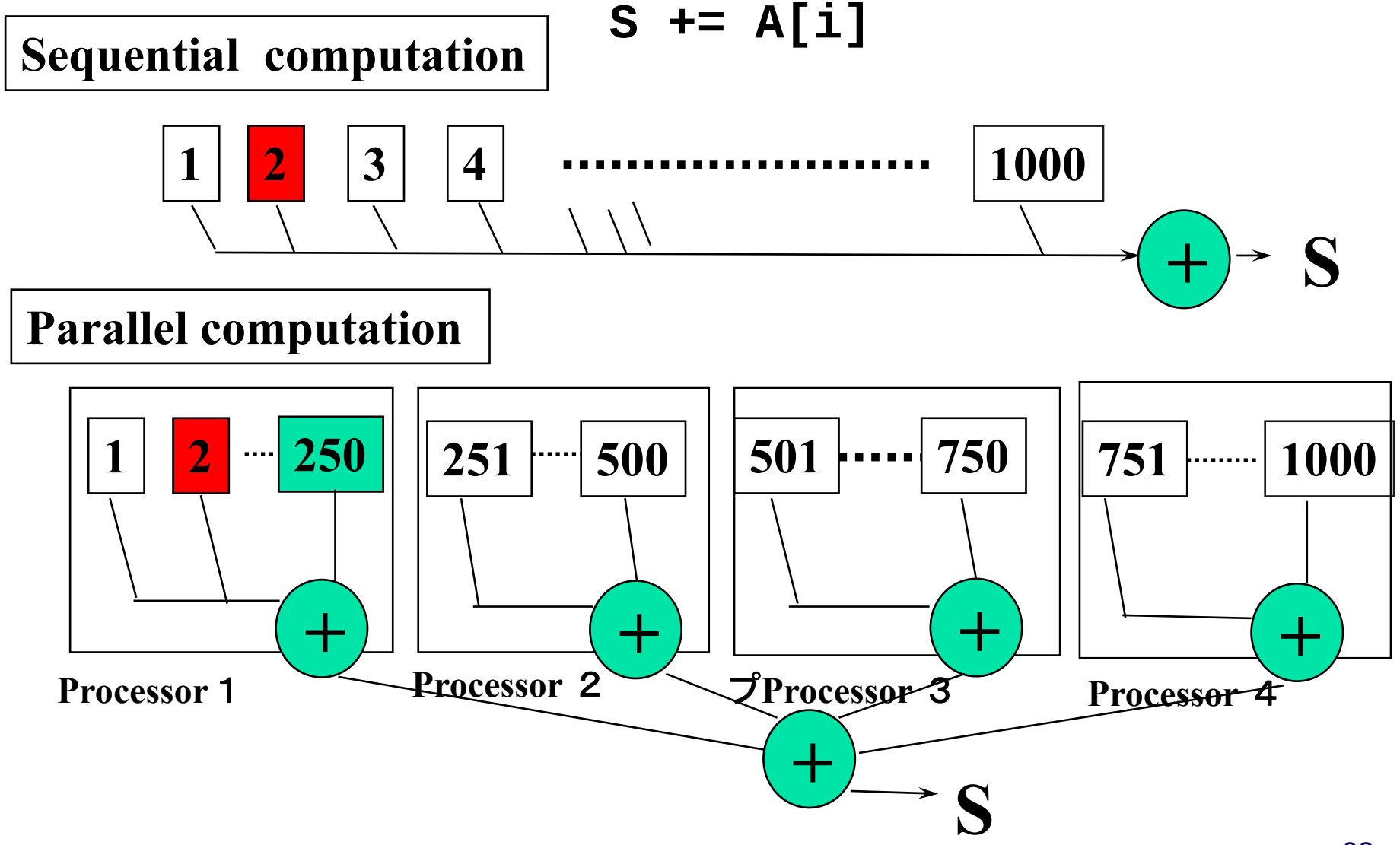
Multithread(ed) programming

- Basic model for shared memory
- Thread of execution = abstraction of execution in processors.
  - Different from process
    - Procss = thread + memory space
  - POSIX thread library = pthread



Very simple example of parallel computing

```
for(i=0;i<1000; i++)
    S += A[i]
```



Programming using POSIX thread

- Create threads

Pthread, Solaris thread

```
for(t=1;t<n_thd;t++){
  r=pthread_create(thd_main,t)
}
thd_main(0);
for(t=1; t<n_thd;t++)
       pthread_join();
```

Thread =
Execution of program

- Divide and assign iterations of loop
- Synchronization for sum

```
int s;  /* global */
int n_thd; /* number of threads */
int thd_main(int id)
{ int c,b,e,i,ss;
  c=1000/n_thd;
  b=c*id;
  e=s+c;
  ss=0;
  for(i=b; i<e; i++) ss += a[i];
  pthread_lock();
  s += ss;
  pthread_unlock();
  return s;
}
```

Programming in OpenMP

これだけで、OK!

```
#pragma omp parallel for reduction(+:s)
   for(i=0; i<1000;i++) s+= a[i];
```

What's OpenMP?

- Programming model and API for shared memory parallel programming
  - It is not a brand-new language.
  - Base-languages(Fortran/C/C++) are extended for parallel programming by directives.
  - Main target area is scientific application.
  - Getting popular as a programming model for shared memory processors as multi-processor and multi-core processor appears.

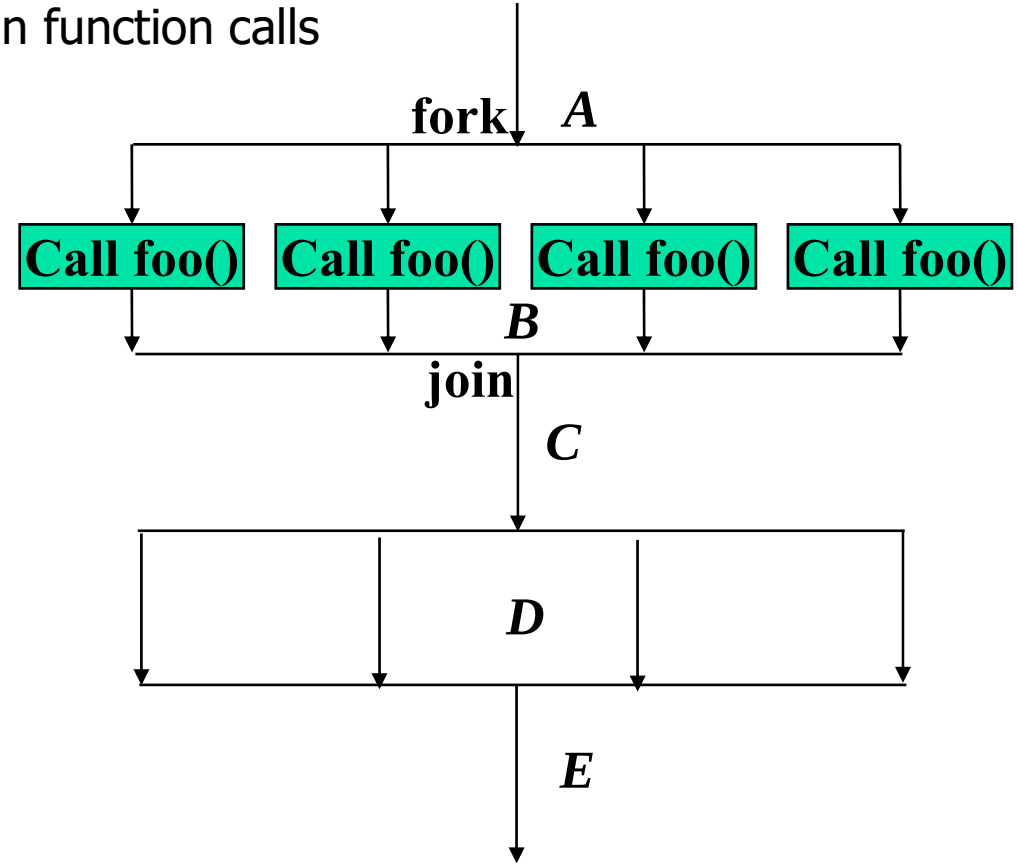
- OpenMP Architecture Review Board (ARB) decides spec.
  - Initial members were from ISV compiler venders in US.
  - Oct. 1997 Fortran ver.1.0 API
  - Oct. 1998 C/C++ ver.1.0 API
  - Latest version, OpenMP 3.0

- http://www.openmp.org/

OpenMP Execution model

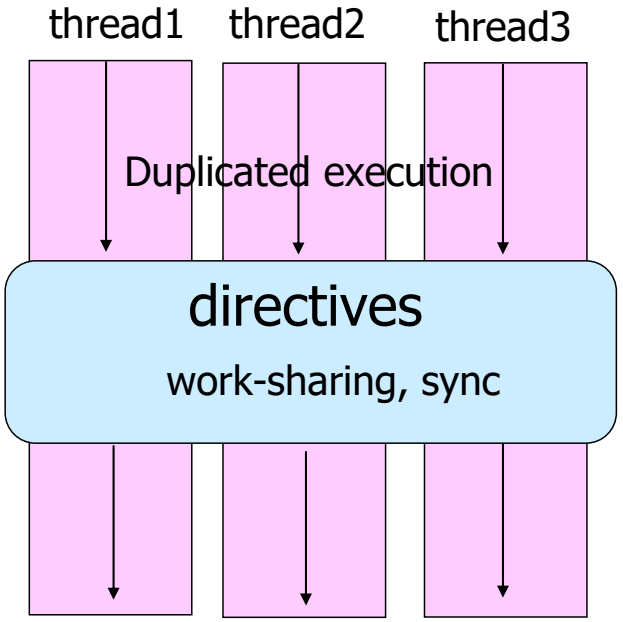
- Start from sequential execution
- Fork-join Model
- parallel region
  - Duplicated execution even in function calls

```
… A …
#pragma omp parallel
{
    foo(); /* ..B… */
}
… C ….
#pragma omp parallel
{
… D …
}
… E …
```



Work sharing Constructs

- Specify how to share the execution within a team
  - Used in parallel region
  - `for` Construct
    - Assign iterations for each threads
    - For data parallel program
  - `Sections` Construct
    - Execute each section by different threads
    - For task-parallelism
  - `Single` Construct
    - Execute statements by only one thread
  - Combined Construct with parallel directive
    - `parallel for` Construct
    - `parallel sections` Construct



For Construct

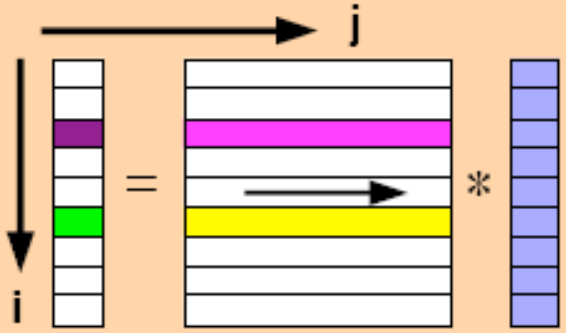
- Execute iterations specified For-loop in parallel
- For-loop specified by the directive must be in *canonical shape*

```
#pragma omp for [clause...]
 for(var=lb; var logical-op ub; incr-expr)
    body
```

  - *Var* must be loop variable of integer or pointer(automatically private)
  - *incr-expr*
    - ++*var*, *var*++, --*var*, *var*--, *var*+=*incr*, *var*-=*incr*
  - *logical-op*
    - <、<=、>、>=
  - Jump to ouside loop or break are not allows
  - Scheduling method and data attributes are specified in *clause*

Example: matrix-vector product

```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```



TID = 0

```
for (i=0,1,2,3,4)
 i = 0
   sum = Σ b[i=0][j]*c[j]
   a[0] = sum
 i = 1
   sum = Σ b[i=1][j]*c[j]
   a[1] = sum
```

TID = 1

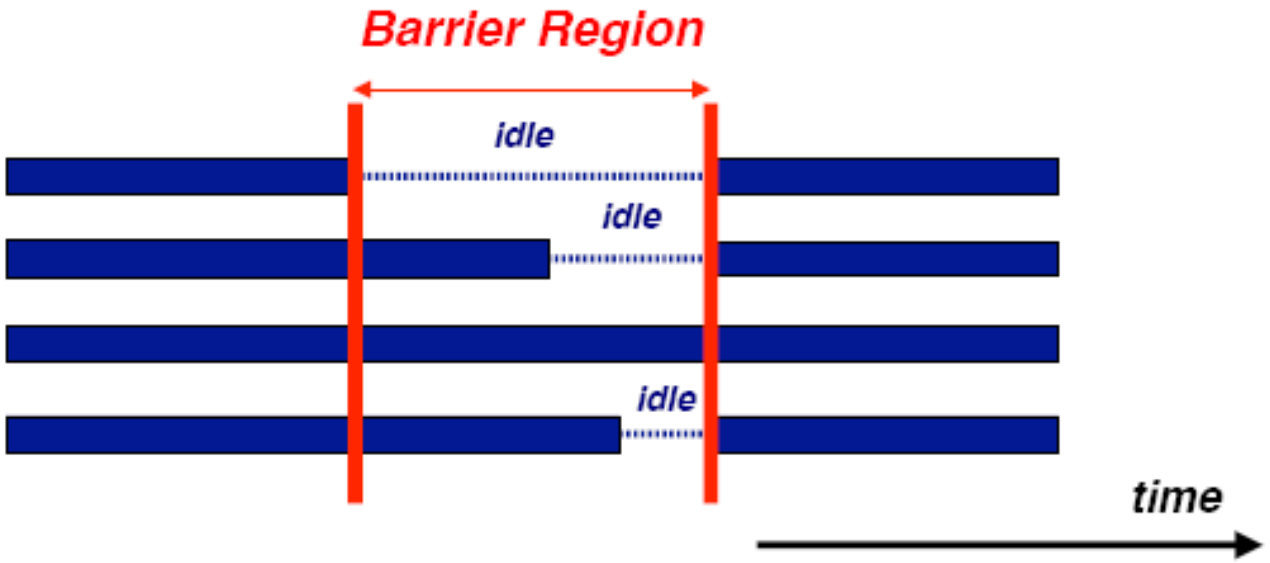
```
for (i=5,6,7,8,9)
 i = 5
   sum = Σ b[i=5][j]*c[j]
   a[5] = sum
 i = 6
   sum = Σ b[i=6][j]*c[j]
   a[6] = sum
```

... etc ...

Barrier directive

- Sync team by barrier synchronization
  - Wait until all threads in the team reached to the barrier point.
  - Memory write operation to shared memory is completed (flush) at the barrier point.
  - Implicit barrier operation is performed at the end of parallel region, work sharing construct without `nowait` clause

```
#pragma omp barrier
```



Other directives

- Single construct： to specify a region executed by one thread.
- Master construct: to specify a region executed by master thread.
- Section construct: to specify regions executed by different threads (task parallelism)
- Critical construct：to specify critical region executed exclusively between threads
- Flush construct
- Threadprivate construct

GPU Computing

- GPGPU - General-Purpose Graphic Processing Unit
  - A technology to make use of GPU for general-purpose computing (scientific applications)

- CUDA（Compute Unified Device Architecture）
  - Co-designed Hardware and Software to exploit computing power of NVIDIA GPU for GP computing.
  - (In other words), at the moment, in order to obtain full performance of GPGPU, a program must be written in CUDA language.

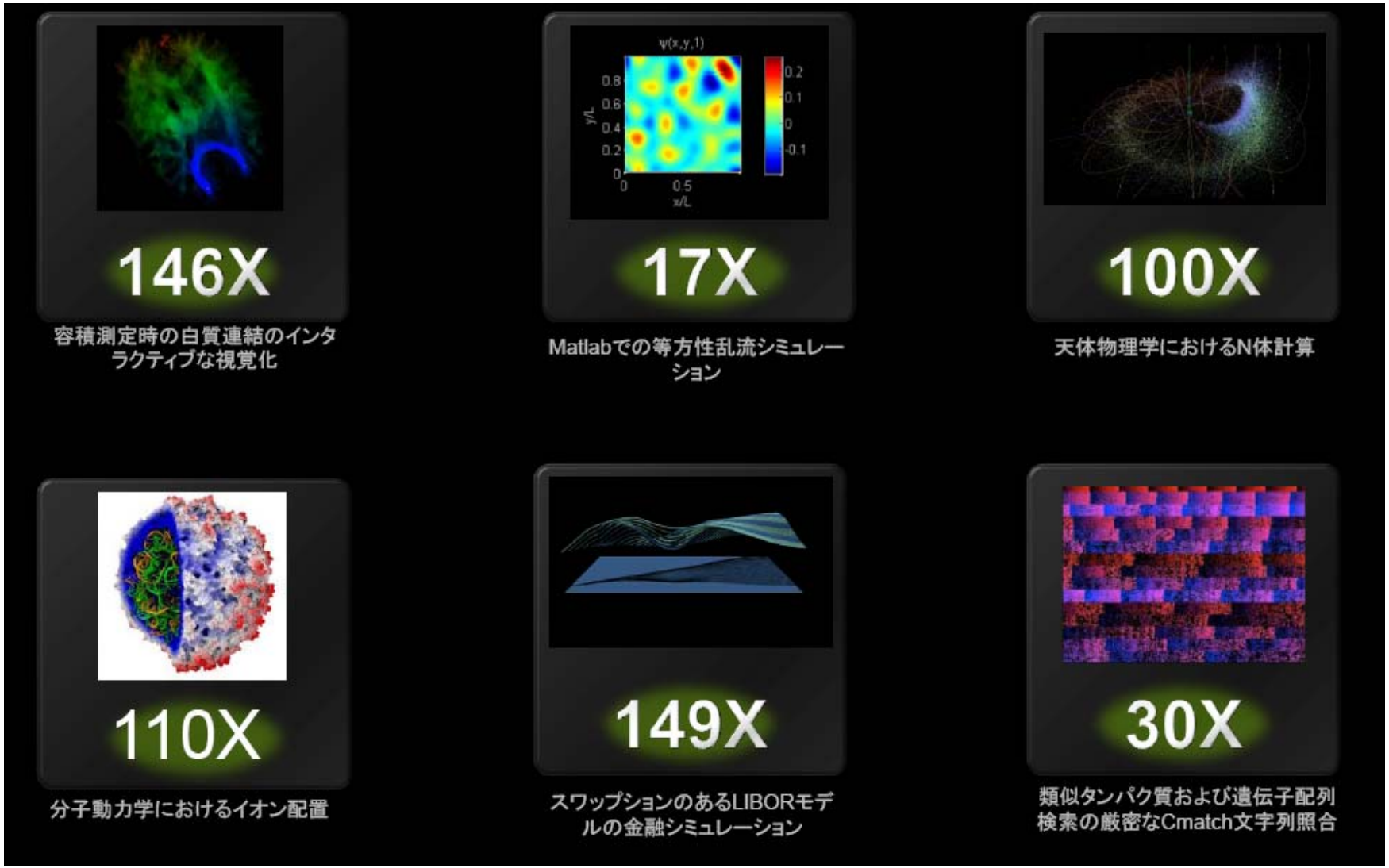
- It is attracting many people's interest since GPU enables great performance much more than that of CPU (even multi-core) in some scientific fields.

- Why GPGPU now?—— price (cost-performance)!!!

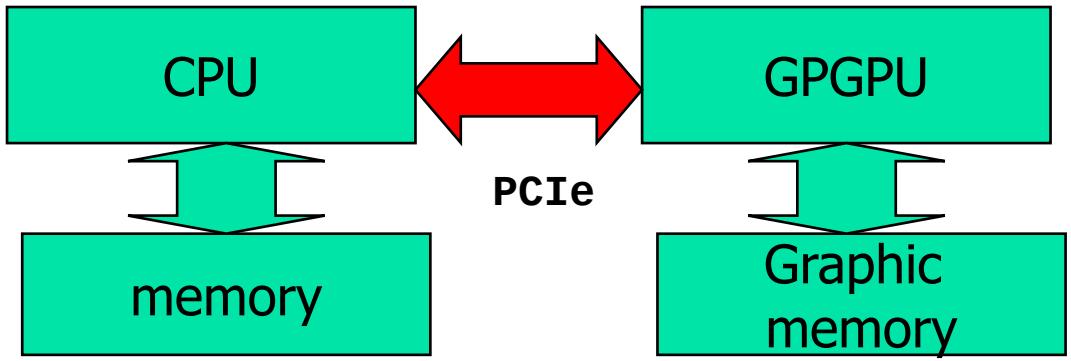
Parallel Programming for GPU

CUDA & OPENACC

Applications（From NVIDIA's slides）

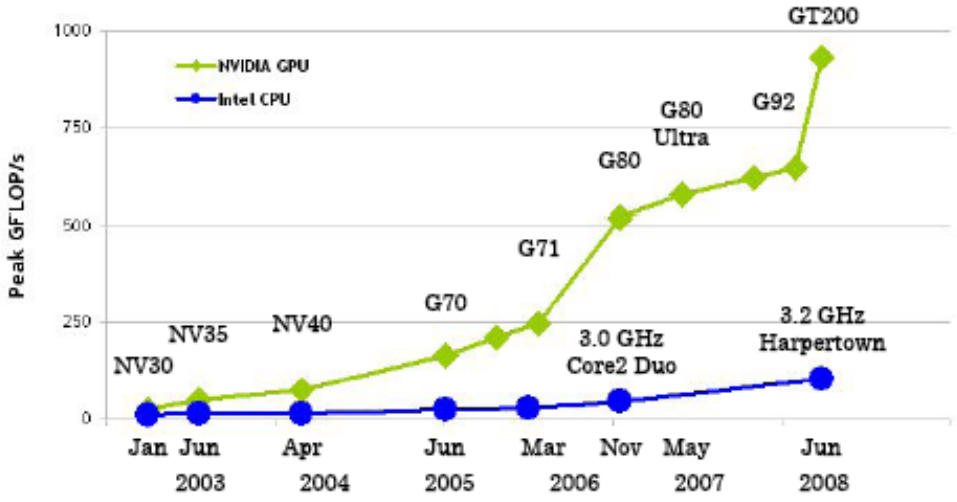


CPU vs. GPU



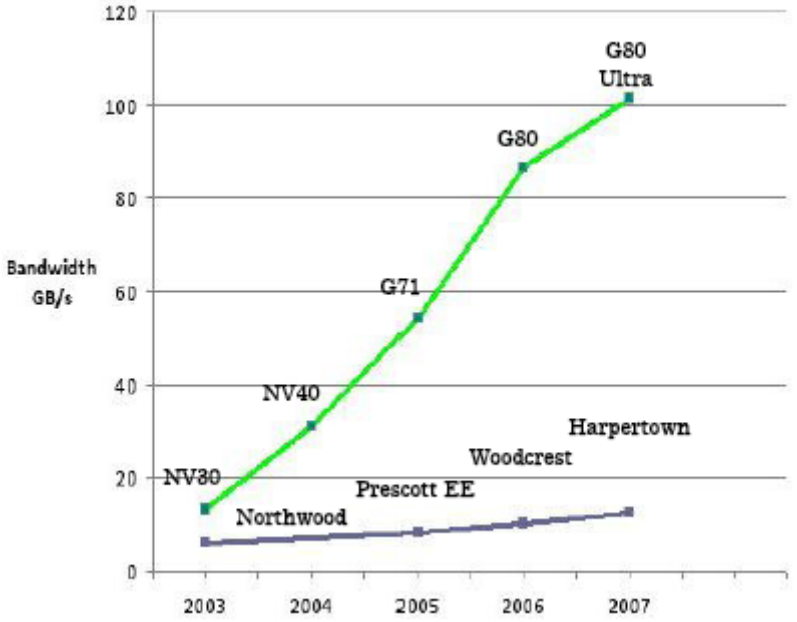
Connected via PCIexpress

Computing performance



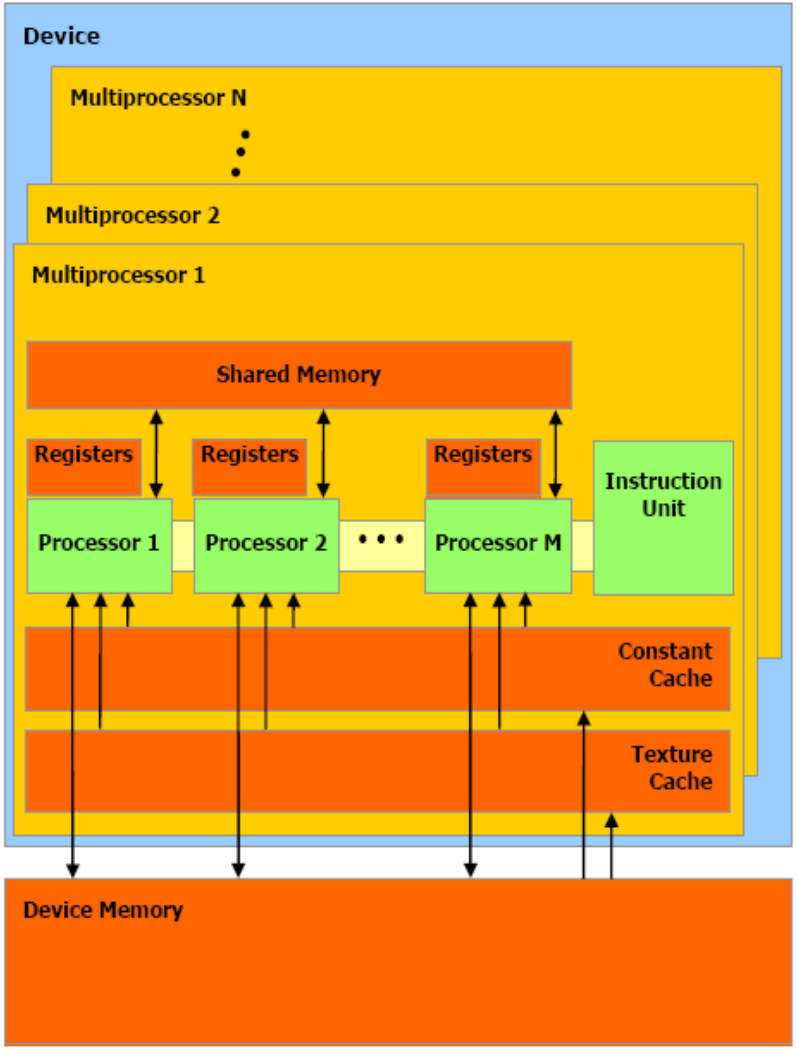
GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

Memory bandwidth



NVIDIA GPGPU's architecture

- Many multiprocessor in a chip
  - eight Scalar Processor (SP) cores,
  - two special function units for transcendentals
  - a multithreaded instruction unit
  - on-chip shared Memory
- SIMT (single-instruction, multiple-thread).
  - The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state.
  - creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.
- Complex memory hierarchy
  - Device Memory (Global Memory)
  - Shared Memory
  - Constant Cache
  - Texture Cache

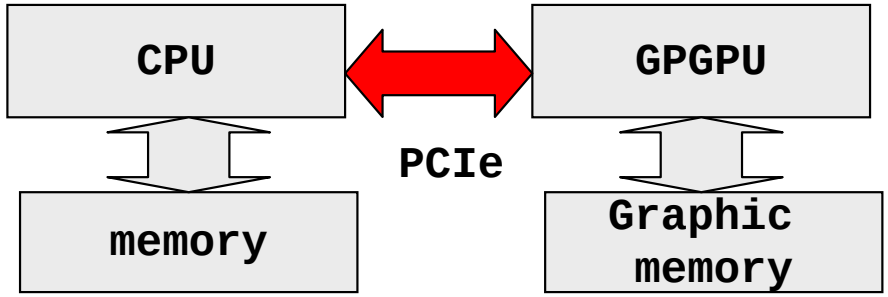


CUDA (Compute Unified Device Architecture)

- C programming language on GPUs
- Requires no knowledge of graphics APIs or GPU programming
- Access to native instructions and memory
- Easy to get started and to get real performance benefit
- Designed and developed by NVIDIA
- Requires an NVIDIA GPU (GeForce 8xxx/Tesla/Quadro)
- Stable, available (for free), documented and supported
- For both Windows and Linux

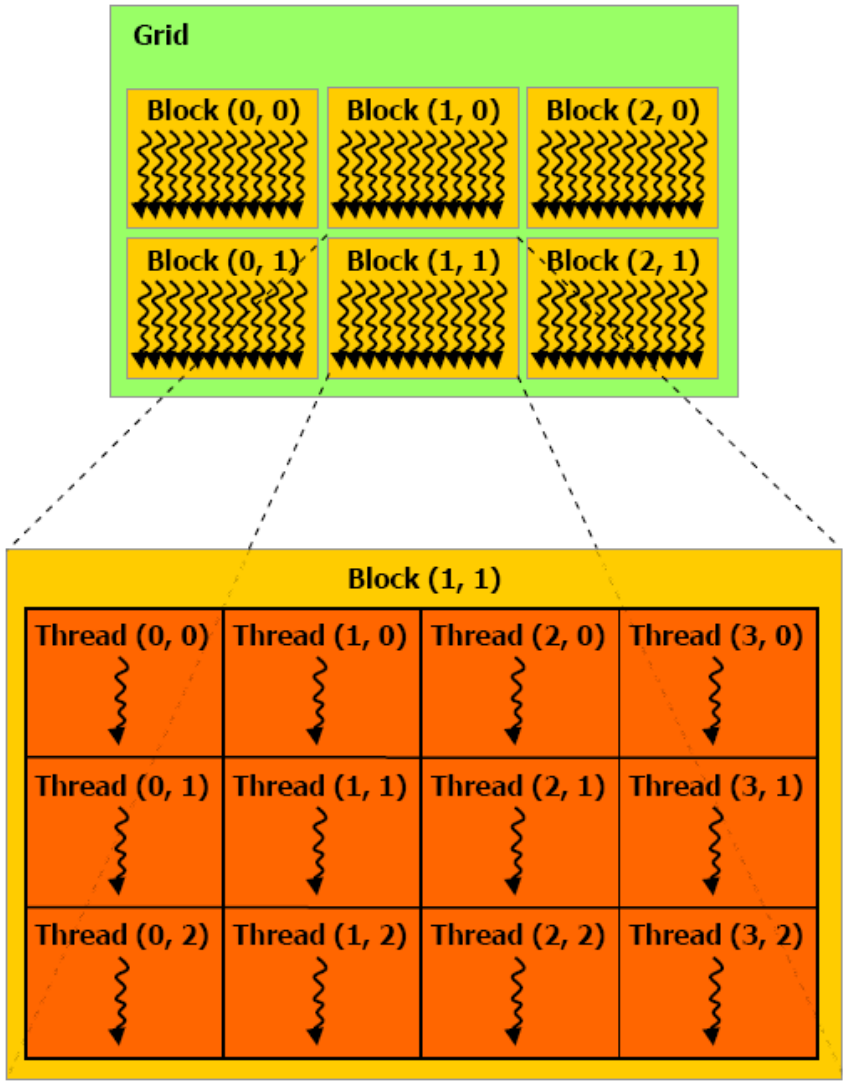
CUDA Programming model (1/2)

- GPU is programmed as a compute device working as co-processor from CPU(host).
  - Codes for data-parallel, compute intensive part are offloaded as functions to the device
  - Offload hot-spot in the program which is frequently executed on the same data
    - For example, data-parallel loop on the same data
  - Call "kernel" a code of the function compiled as a function for the device
  - Kernel is executed by multiple threads of device.
    - Only one kernel is executed on the device at a time.
  - Host (CPU) and device(GPU) has its owns memory, host memory and device memory
  - Data is copied between both memory.



CUDA Programming model (2/2)

- computational Grid is composed of multiple thread blocks
- thread block includes multiple threads
- Each thread executes kernel
  - A function executed by each thread called "kernel"
  - Kernel can be thought as one iteration in parallel loop
- computational Grid and block can have 1,2,3 dimension
- The reserved variable, blockID and threadID have ID of threads.



Example: Element-wise Matrix Add

```
void add_matrix
( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
  for ( int j = 0; j < N; ++j ) {
    index = i + j*N;
    c[index] = a[index] + b[index];
  }
}
int main() {
 add_matrix( a, b, c, N );
}
```

CPU program

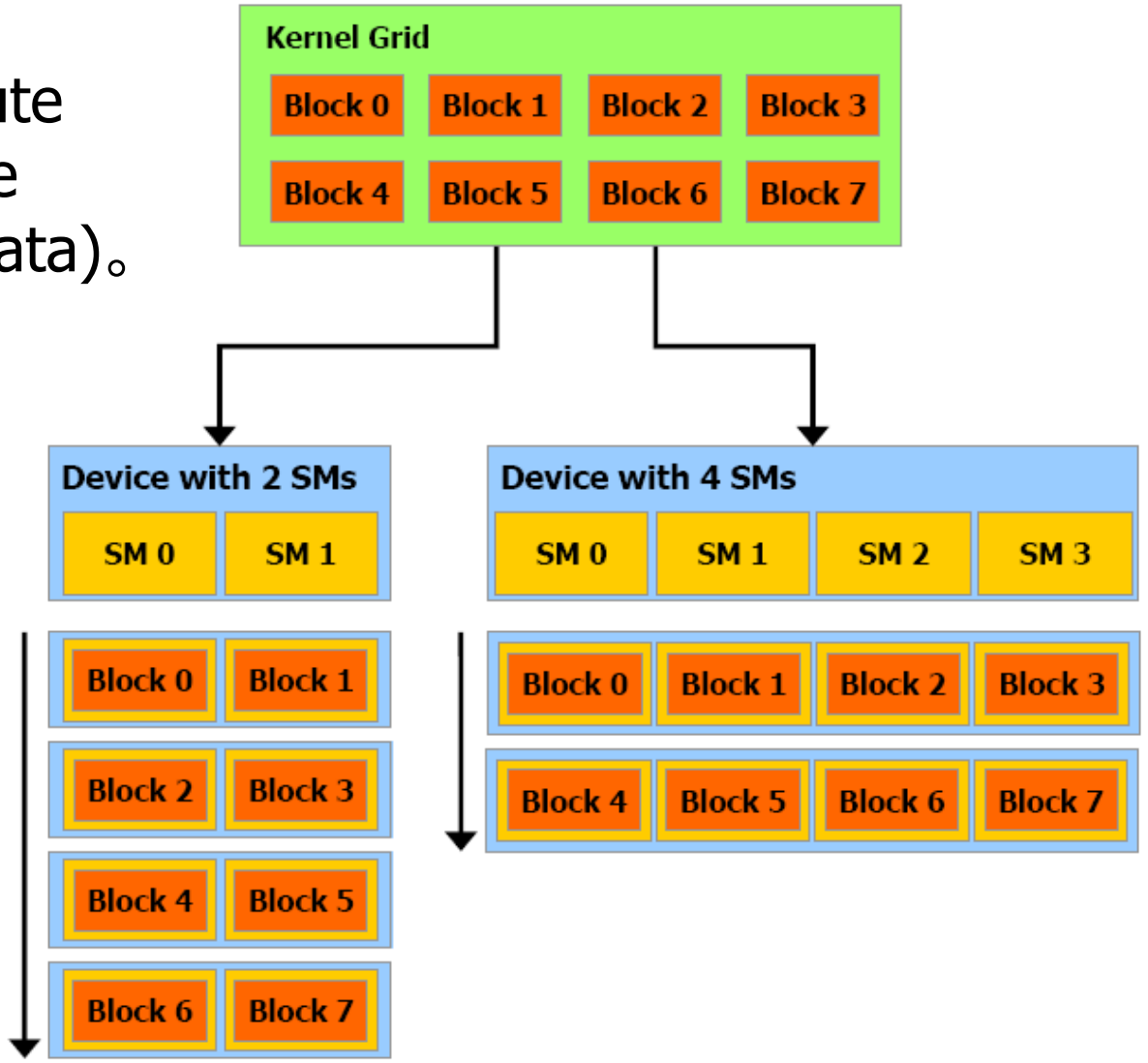
The nested for-loops are replaced with an implicit grid

CUDA program

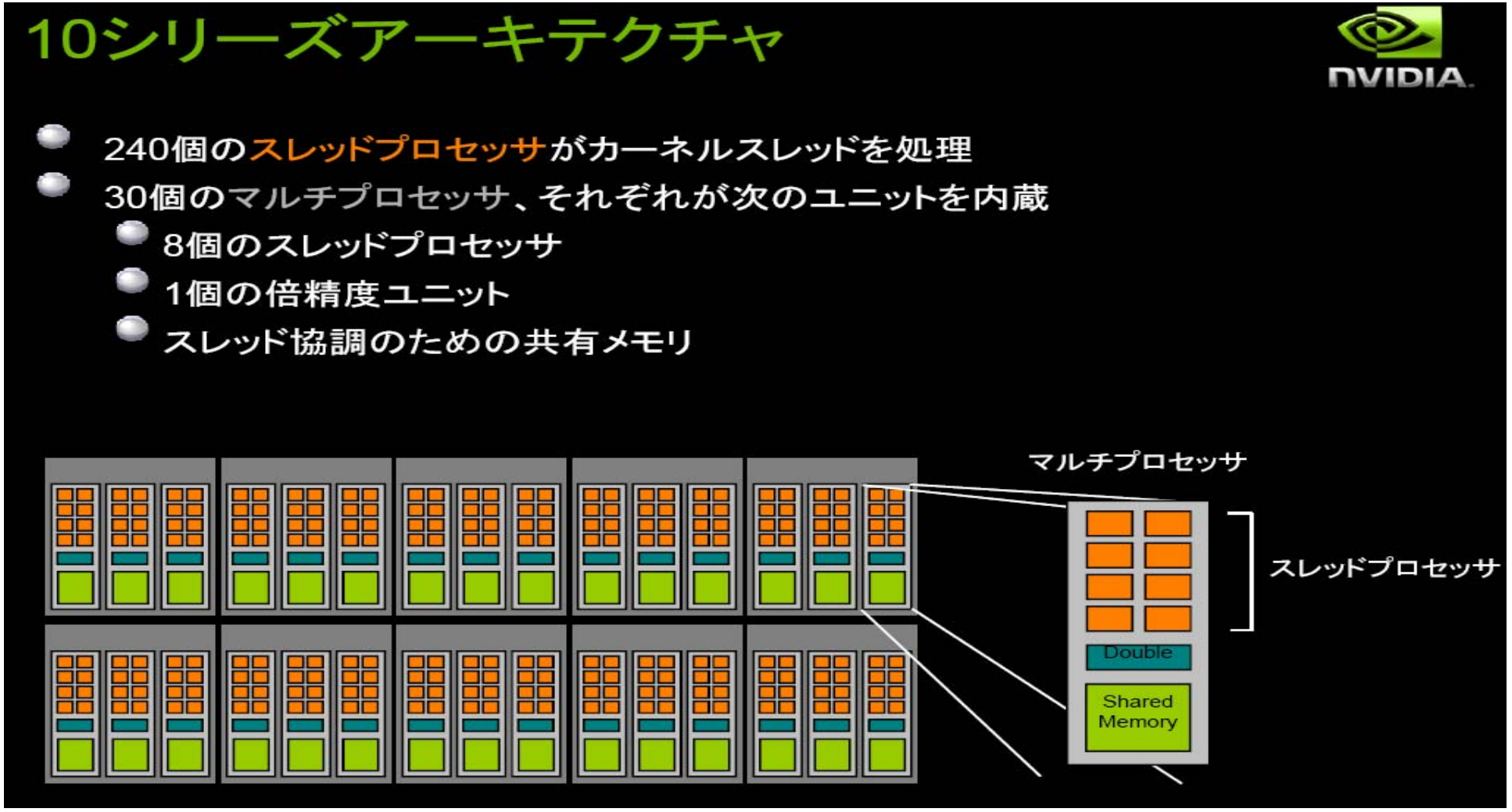
```
__global__ add_matrix
( float* a, float* b, float* c, int N ) {
 int i = blockIdx.x * blockDim.x + threadIdx.x;
 int j = blockIdx.y * blockDim.y + threadIdx.y;
 int index = i + j*N;
 if ( i < N && j < N )
  c[index] = a[index] + b[index];
}
int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

How to be executed

- SM (Streaming Multiprocessor) execute blocks in SIMD (single instruction/multiple data)。
- SM consists of 8 processors



An example of GPGPU configuration



	Number of Multiprocessors (1 Multiprocessor = 8 Processors)	Compute Capability
GeForce GTX 295	2x30	1.3
GeForce GTX 285, GTX 280	30	1.3
GeForce GTX 260	24	1
GeForce 9800 GX2	2x16	1
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512	16	1
GeForce 8800 Ultra, 8800 GTX	16	1
GeForce 9800 GT, 8800 GT, GTX 280M, 9800M GTX	14	1
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTX 260M, 9800M GT	12	1
Tesla S1070		
Tesla C1060	30	1.3
Tesla S870	4x16	1.0
Tesla D870	2x16	1.0
Tesla C870	16	1.0
Quadro Plex 2200 D2	2x30	1.3
Quadro Plex 2100 D4	4x14	1.1
Quadro Plex 2100 Model S4	4x16	1.0

Tesla C1060

コア数: 240コア
プロセッサ周波数: 1.3GHz
搭載メモリ: 4GB
単精度浮動小数点演算性能: 933GFlops (ピーク)
倍精度浮動小数点演算性能: 78GFlops (ピーク)
メモリ帯域: 102GB/sec
標準電力消費量: 187.8W
浮動小数点演算: IEEE 754 単精度/倍精度
ホスト接続: PCI Express x16 (PCI-E2.0対応)

Invoke (Launching) Kernel

- Host processor invoke the execution of kernel in this form similar to function call:

```
kernel<<<dim3 grid, dim3 block, shmem_size>>>(…)
```

- Execution Configuation ("<<< >>>")
  - Dimension of computational grid : x and y
  - Dimension of thread block: x、y、z

```
dim3 grid(16 16);
dim3 block(16,16);
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

Memory management (1/2)

- CPU and GPU have different memory space.
- Hosts（CPU）manages device (GPU）memory

- Allocation and Deallocation of GPU memory
  - `cudaMalloc(void ** pointer, size_t nbytes)`
  - `cudaMemset(void * pointer, int value, size_t count)`
  - `cudaFree(void* pointer)`

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *d_a = 0;
cudaMalloc( (void**)&d_a nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Memory management (2/2)

- Data copy operation between CPU and device
  - `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
    - Direction specifies how to copy from src to dst , see below
    - Block a caller of CPU thread (execution) until the memory transfer completes.
    - Copy operation starts after previous CUDA calls.
  - `enum cudaMemcpyKind`
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToHost`
    - `cudaMemcpyDeviceToDevice`

Example (host-side program)

```
// allocate host memory
int numBytes = N * sizeof(float)
float* h_A = (float*) malloc(numBytes);

// allocate device memory
// float* d_A = 0;
cudaMalloc((void**)&d_A, numbytes);

// Copy data from host to device
cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);

// Execute kernel
increment_gpu<<< N/blockSize, blockSize>>>(d_A, b);

// copy back data from device to host
cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A);
```

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
   a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;
  return EXIT_SUCCESS;
}
```

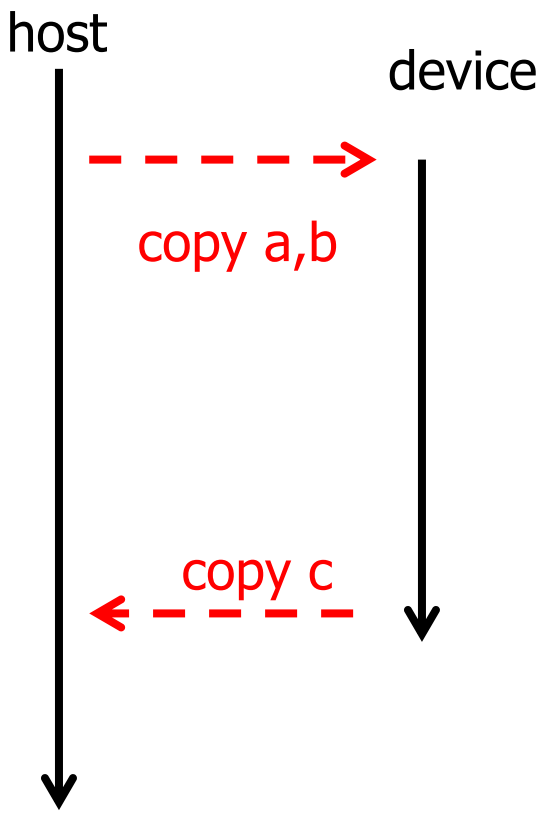
OpenACC

- A spin-off activity from OpenMP ARB for supporting accelerators such as GPGPU
- NVIDIA, Cray Inc., the Portland Group (PGI), and CAPS enterprise
- Directive to specify the code offloaded to GPU.

A simple example

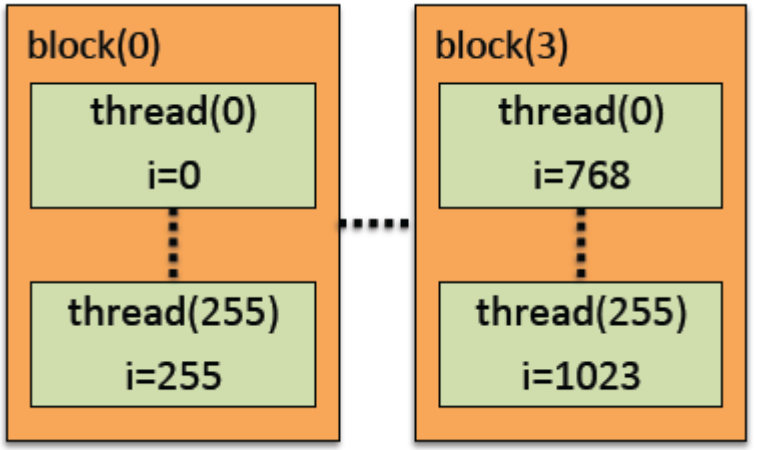
direction	copy	copyin	copyout
Host->device	○	○	
Device->Host	○		○

```
#define N 1024
int main(){
int i;
int a[N], b[N],c[N];
#pragma acc data copyin(a,b) copyout(c)
{
 #pragma acc parallel
 {
 #pragma acc loop
   for(i = 0; i < N; i++){
      c[i] = a[i] + b[i];
   }
 }
}
}
```



A simple example

```
#define N 1024
int main(){
int i;
int a[N], b[N],c[N];
#pragma acc data copyin(a,b) copyout(c)
{
 #pragma acc parallel
 {
 #pragma acc loop
   for(i = 0; i < N; i++){
      c[i] = a[i] + b[i];
   }
 }
}
}
```



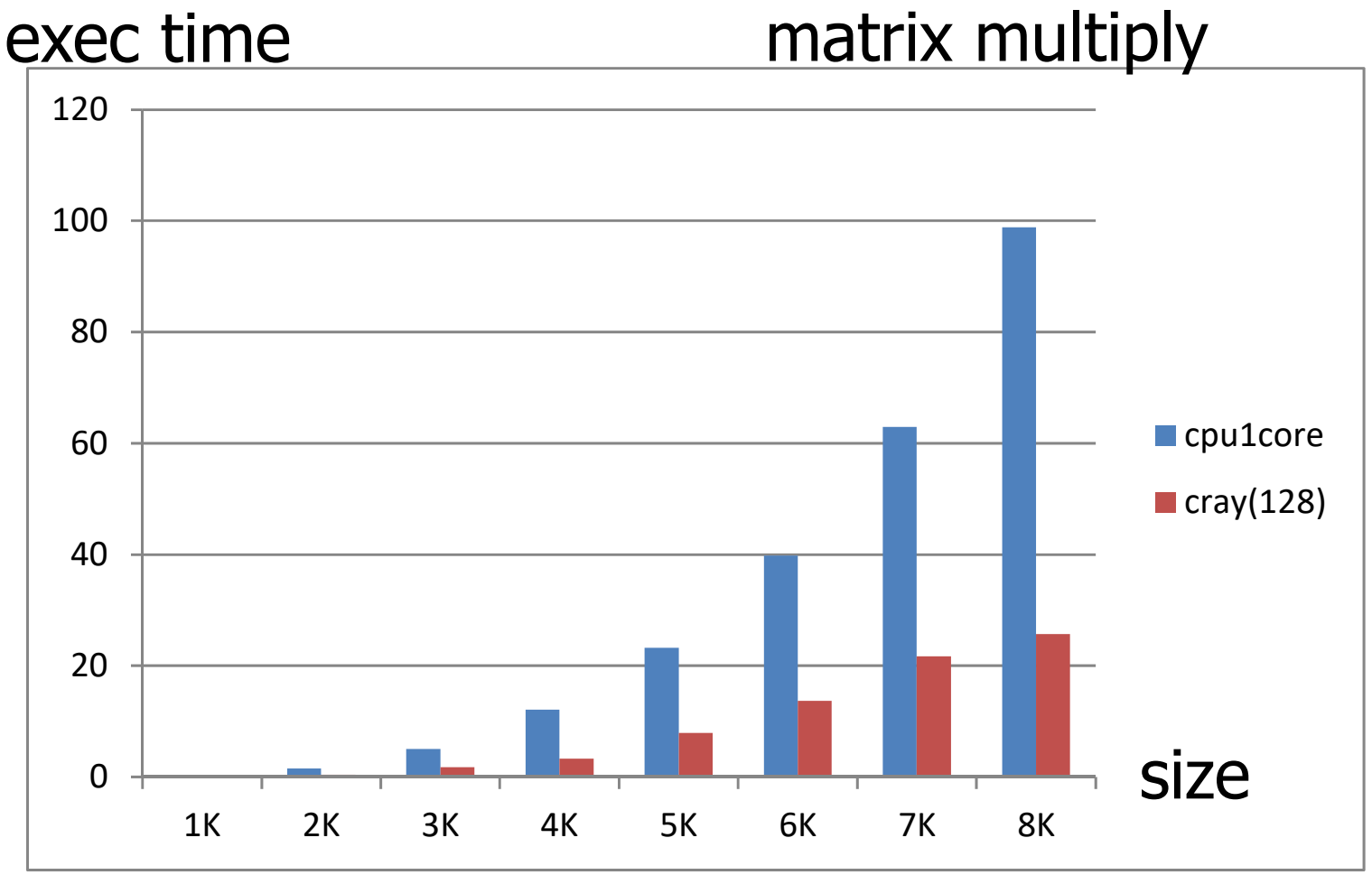
execute iterations like CUDA kernel

Matrix Multiply in OpenACC

```
#define N 1024

void main(void)
{
  double a[N][N], b[N][N], c[N][N];
  int i,j;
  // ... setup data ...
#pragma acc parallel loop copyin(a, b) copyout(c)
  for(i = 0; i < N; i++){
#pragma acc loop
    for(j = 0; j < N; j++){
      int k;
      double sum = 0.0;
      for(k = 0; k < N; k++){
            sum += a[i][k] * b[k][j];
      }
      c[i][j] = sum;
    }
  }
}
```

Performance of OpenACC code



MAPREDUCE & CLOUD

MapReduce (2004)

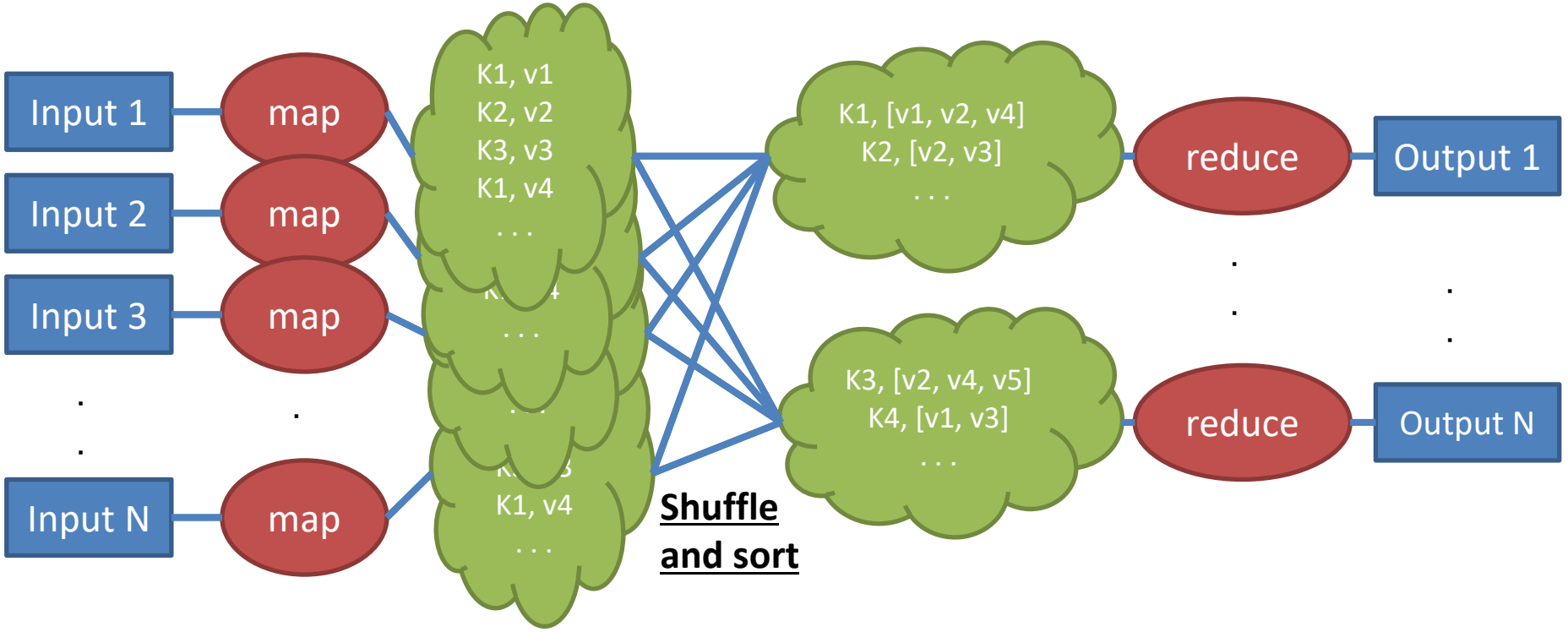
- **Programming model and runtime for data processing on large-scale cluster**
- A user specifies map and reduce functions
- Runtime system does
  - Automatically parallelize
  - Manage machine failure
  - Schedule jobs to efficiently exploit disk and network

Background

- Google requires to process
  - Inverted index
  - Various graph expression of Web documents
  - Number of pages that each host crawls
  - Set of the hottest query in a day
    - from large amount of crawled documents and Web request logs using hundreds to thousands of compute nodes
- Large amount of codes for parallelization, data distribution, error handing are required
- These hide original code for computation

New abstraction (1)

- Describes only required computation
- Runtime library hides complicated processes including parallelization, fault handling, data distribution, load balancing
- Most computation has the following same pattern



New abstraction (2)

- A functional model of user-supplied map and reduce operations enables
  - Easy parallelization of large-scale computation
  - To run failed tasks again for fault tolerance

- Simple but powerful interface
- It enables high-performance computation on large-scale cluster by auto-parallelization and auto-distribution

Comments on MapReduce

- MapReduce programming model has been successfully used at Google for many different purposes
  - Easy to use
  - It hides details of parallelization, fault tolerance, locality optimization and load balancing
  - A large variety of problems are easily expressible
  - Scales to large clusters of machines comprising thousands of machines
- It can be obtained by restricting the programing model

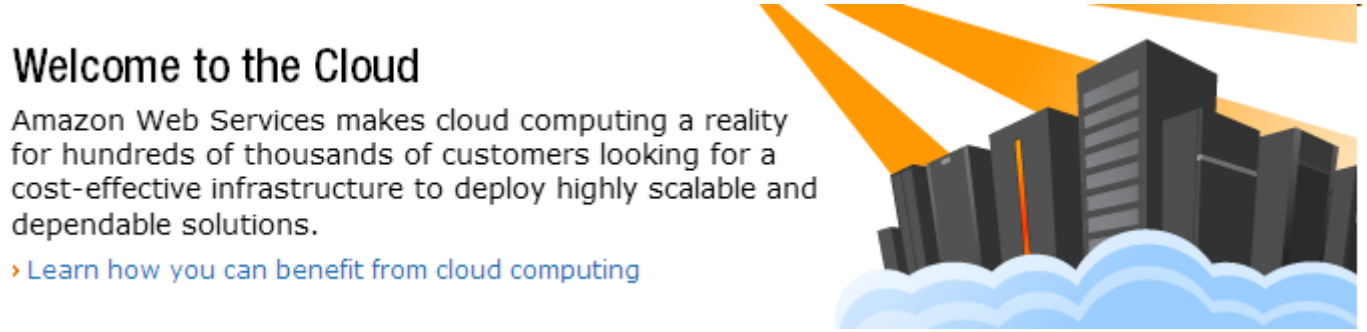
Cloud Computing

- Only required amount of CPU and storage can be used anytime from anywhere via network
  - Availability, throughput, reliability
  - Manageability
- No need to procure, maintain, and update computers

- Large-scale distributed data processing by MapReduce
  - Loosely coupled data intensive computing
  - Can be a standard parallel language other than MPI

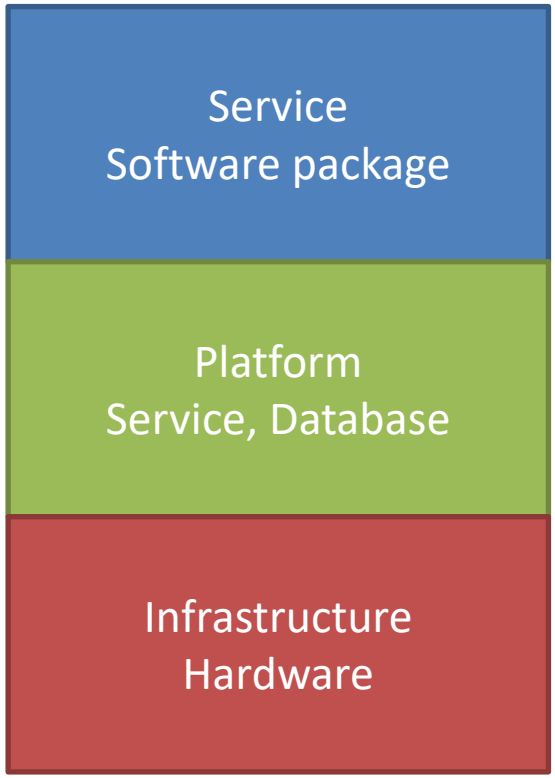
Amazon Web Services (2002)

- On-demand elastic infrastructure managed by web services
  - Elastic Compute Cloud (EC2)
    - Web service that provides resizable compute capacity
  - Simple Storage Service (S3)
    - Simple web service I/F to store and retrieve data
  - Elastic Block Store (EBS)
    - Block level storage used by EC2 in the same AZ
    - Automatically replicate within the same AZ
    - Point-in-time snapshots can be persisted to S3
- Region and Availability Zone



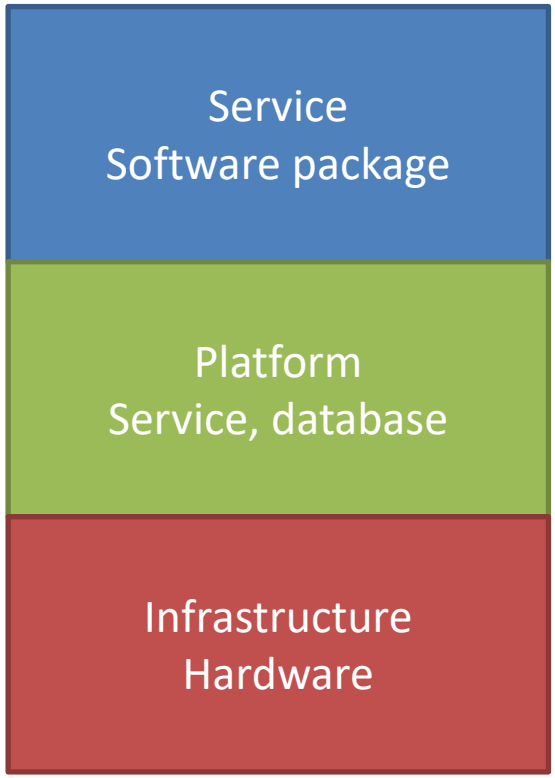
Taxonomy of Cloud

- SaaS (Software as a Service)
  - Google Apps (Gmail, …), CRM
  - Microsoft Online Services
- PaaS (Platform as a Service)
  - Development of Web apps
    - Force.com
    - Google App Engine
- IaaS (Infrastructure as a Service)
  - Amazon EC2, S3
  - Microsoft Azure



Cloud technology

- SaaS (Software as a Service)
  - Web 2.0
- PaaS (Platform as a Service)
  - Web API
  - Web Service
    - XML, WSDL, SOAP/REST
- IaaS (Infrastructure as a Service)
  - Virtual machine (Xen, KVM)
  - Virtualization of harddisk, storage and network



Storage system in cloud

- Availability, reliability
- Amazon Web Services
  - S3, EBS
  - Can construct any (file) system that uses block device
    - HDFS (using EBS) for Elastic MapReduce
  - Difficult to construct a system beyond Availability Zone and Region
- Google App Engine
  - Utilize GFS and BigTable

Summary of cloud computing

- Resources in cloud computing
  - Inexpensive, always available, reliable, high performance
  - Easy to maintain
- Realized by virtualization and web interface
- No need to procure, maintain, and update computers
- If required, more resources can be obtained by cloud