OpenMP

Directive-based Parallel Programming Model for Multi/Many-core Architecture

Jinpil Lee
RIKEN AICS
Programming Environment Research Team

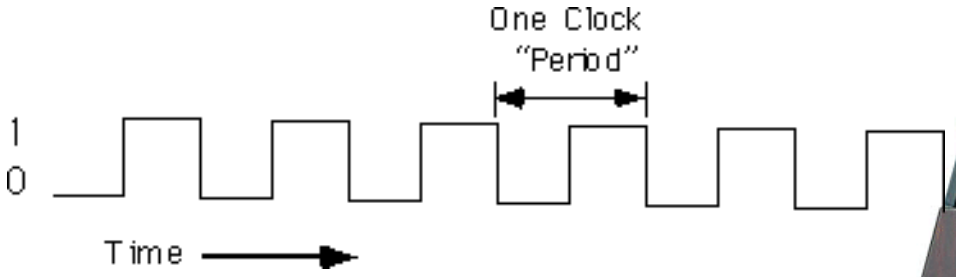
Agenda

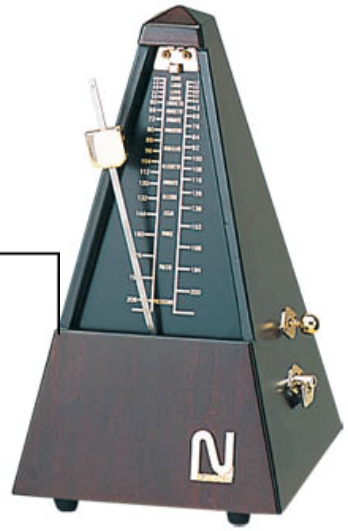
- Current processor trends
- Why we need Multithreading
- OpenMP basic
- Code example
- Advanced topics

  (task parallelism, SIMD, accelerators)

Theoretical Performance

- CPU clock

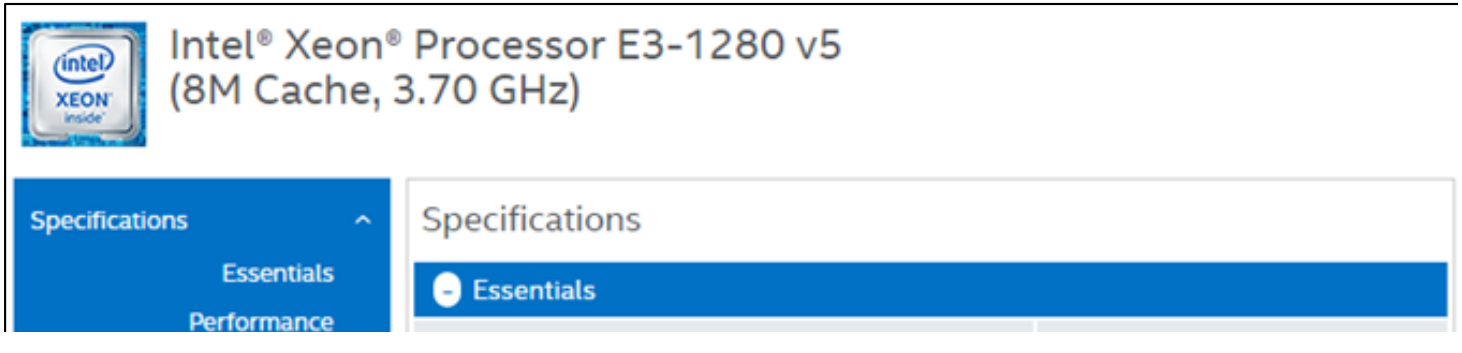




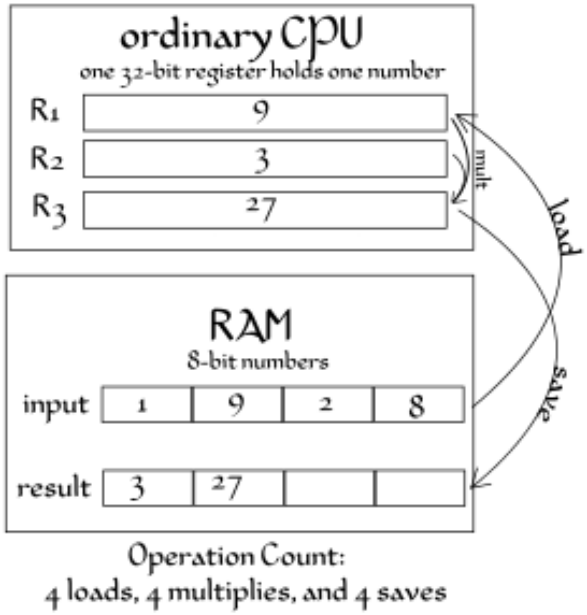
- CPU Frequency: CPU Clocks per second
- IPC: Instructions (Operations) Per Clock
- FLOPS: Floating Operations per Second

3.0 GHz * 1 IPC * 1 core = 3 GFLOPS
2.0 GHz * 4 IPC * 1 core = 8 GFLOPS
1.0 GHz * 2 IPC * 10 cores = 20 GFLOPS

Theoretical Peak Performance

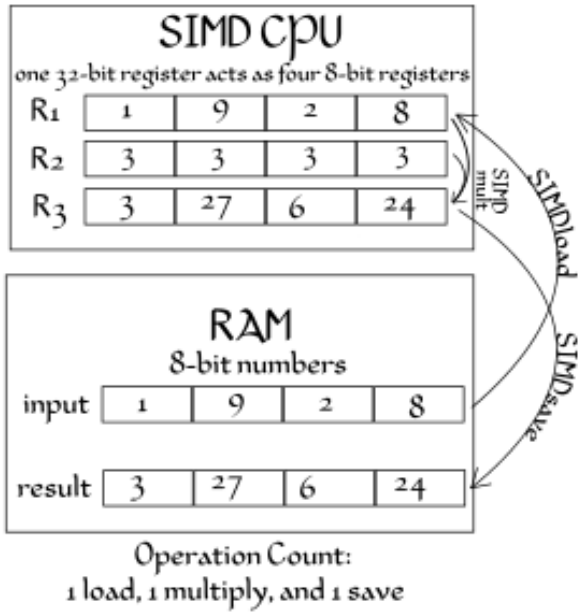


Single Core



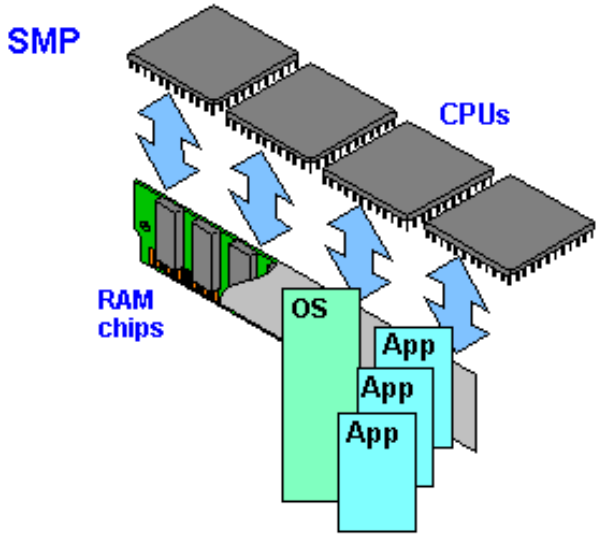
1.0 GHz * 1 IPC
= 1 GFLOPS

SIMD



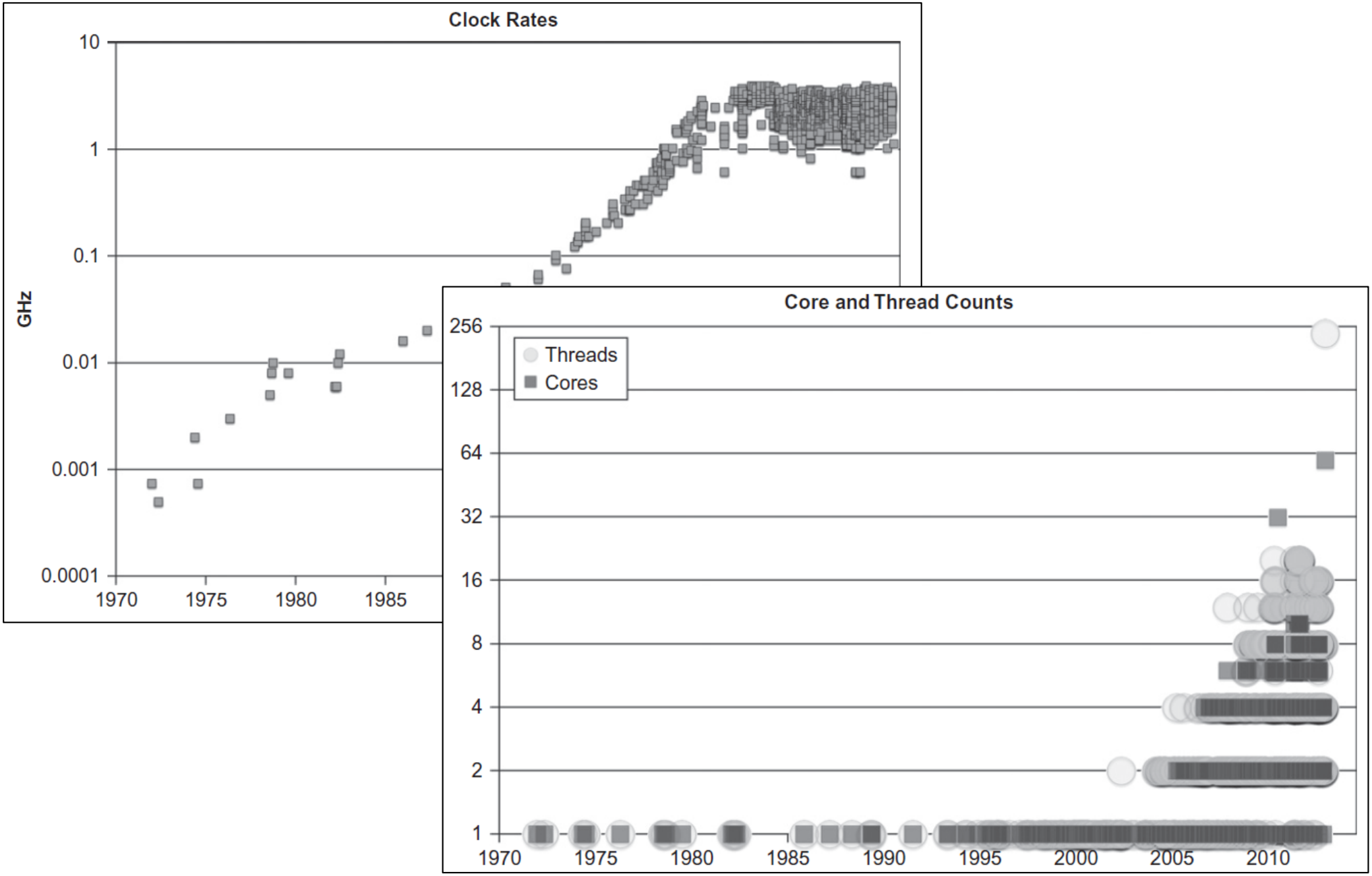
1.0 GHz * 4 IPC
= 4 GFLOPS

SMP



1.0 GHz * 4 IPC
* 4 cores = 16 GFLOPS

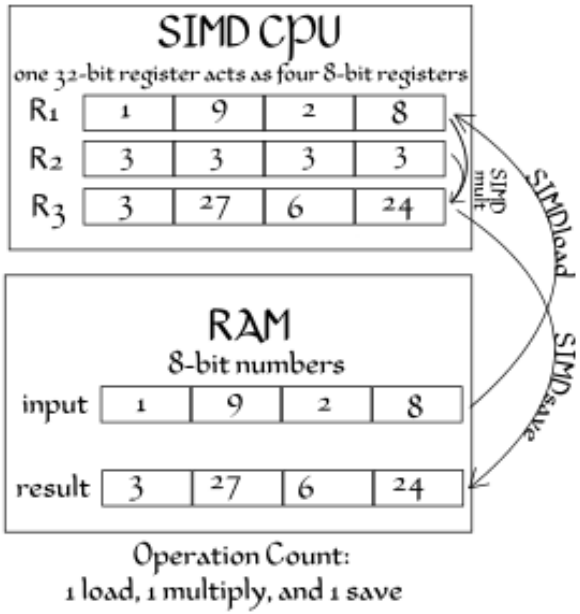
Processor Trends



Processor Trends (cont'd)

- SIMD vector length is getting large
- Intel Advanced Vector eXtension (AVX)
  - most Intel processors in the market
  - 256-bit
  - 8 32-bit values (int, float)
  - 4 64-bit values (long, double)
- Intel AVX-512
  - Knights Landing Architecture
  - 512-bit
  - 16 32-bit values
  - 8 64-bit values
- ARM Scalable Vector Extension (SVE)
  - vector length is not fixed
  - larger than 512-bit

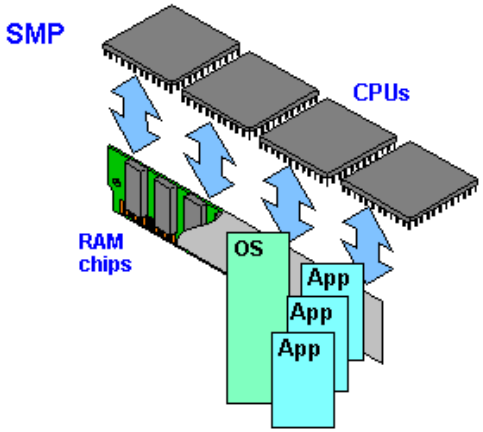
SIMD



1.0 GHz * 4 IPC
= 4 GFLOPS

Why Parallel Programming?

- clock frequency won't increase, we will get more cores, larger SIMD instructions
- normal programs use a single core (and SIMD instructions)
- "Free Lunch is over"



Single core

0.5 GHz * 1 SIMD * 1 cores
= 0.5 GFLOPS

2.0 GHz * 1 SIMD * 1 cores
= 2 GFLOPS

4.0 GHz * 4 SIMD * 1 cores
= 16 GFLOPS

Multi-core

3.0 GHz * 1 SIMD * 1 cores
= 3 GFLOPS

3.0 GHz * 16 SIMD * 1 cores
= 48 GFLOPS

3.0 GHz * 16 SIMD * 16 cores
= 768 GFLOPS

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)*

What is 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 4.5

- http://www.openmp.org/

OpenMP API

- It is not a new language!
  - Base languages are extended by compiler directives/pragma, runtime library, environment variable.
  - Base languages : Fortran 90, C, C++
    - Fortran : directive line starting with !$OMP
    - C: directive by #pragma omp

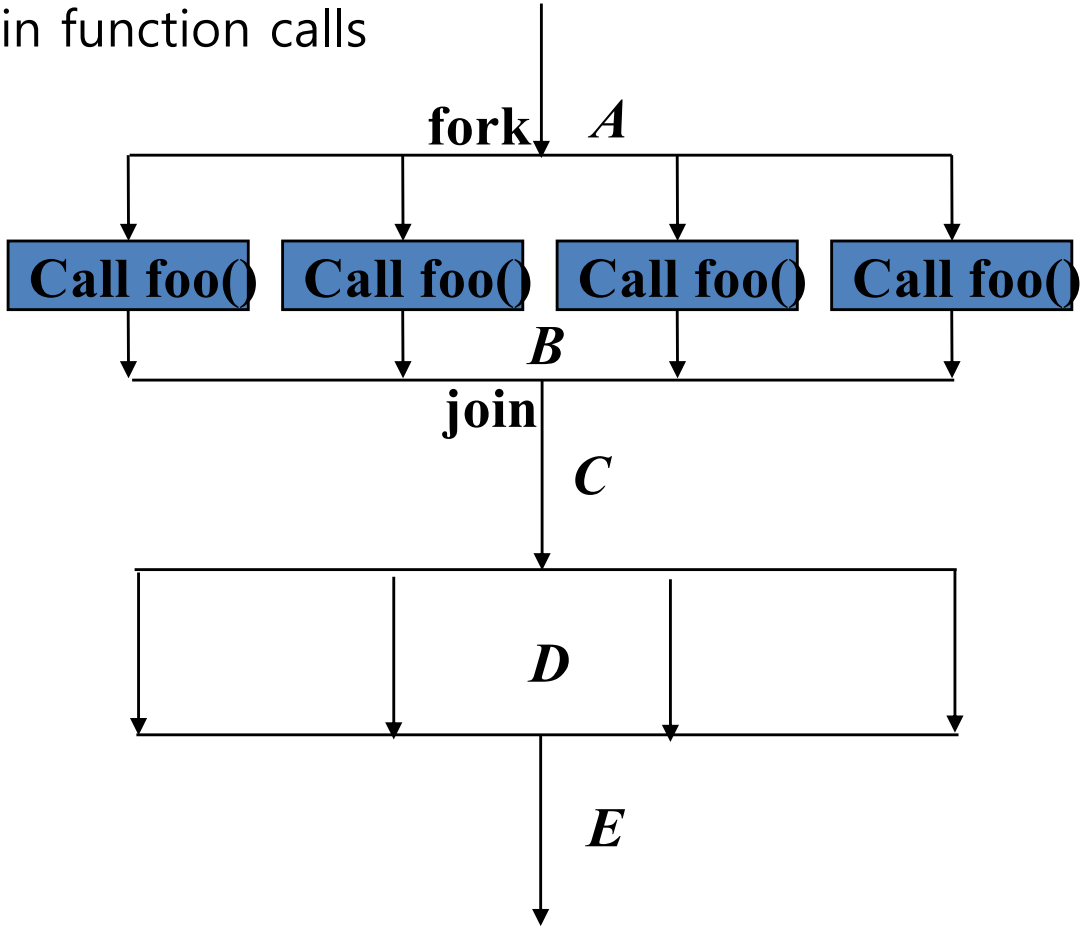
- Different from automatic parallelization
  - OpenMP parallel execution model is defined explicitly by a programmer.

- If directives are ignored (removed), the OpenMP program can be executed as a sequential program
  - Can be parallelized in incrementally
  - Practical approach with respect to program development and debugging.
  - Can be maintained as a same source program for both sequential and parallel version.

OpenMP Execution Model

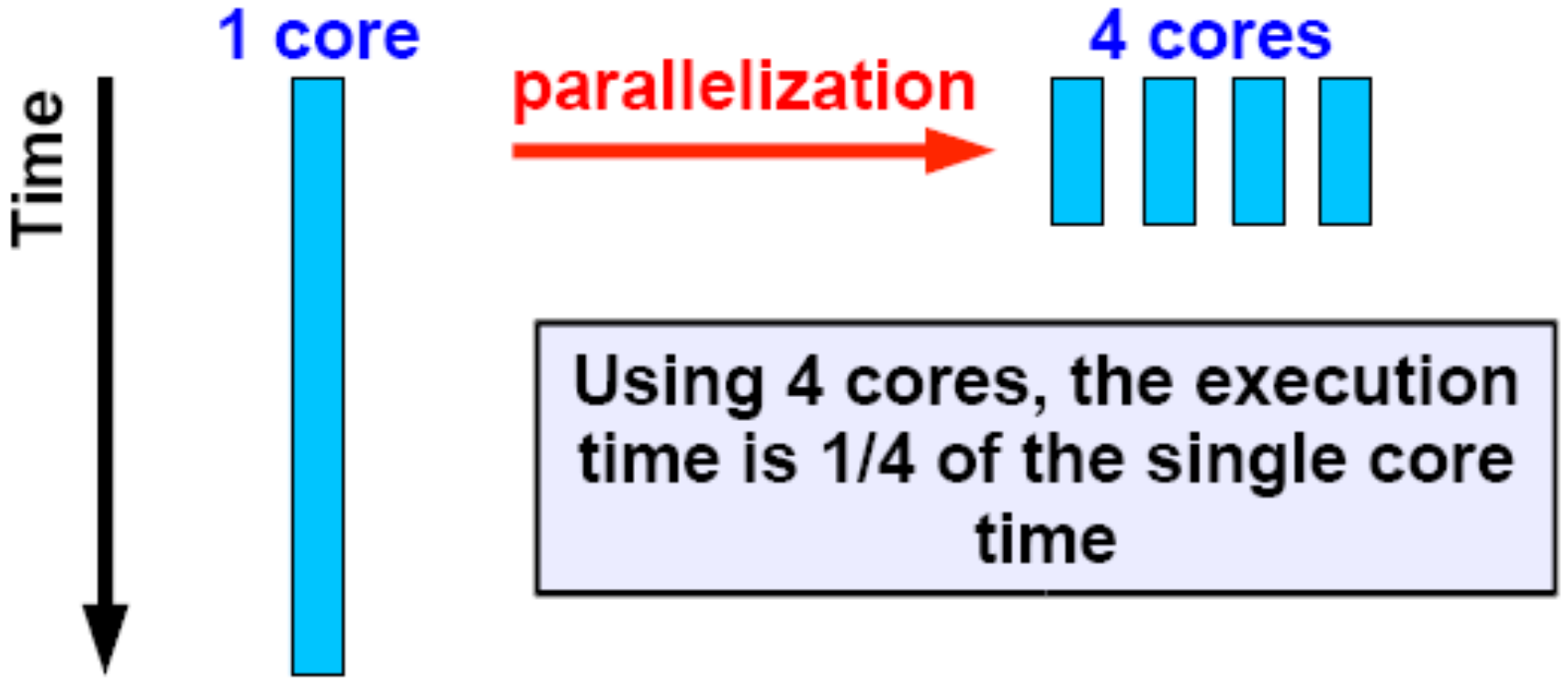
- Start from sequential execution
- Fork-join Model
- parallel region
  - Duplicated execution even in function calls

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

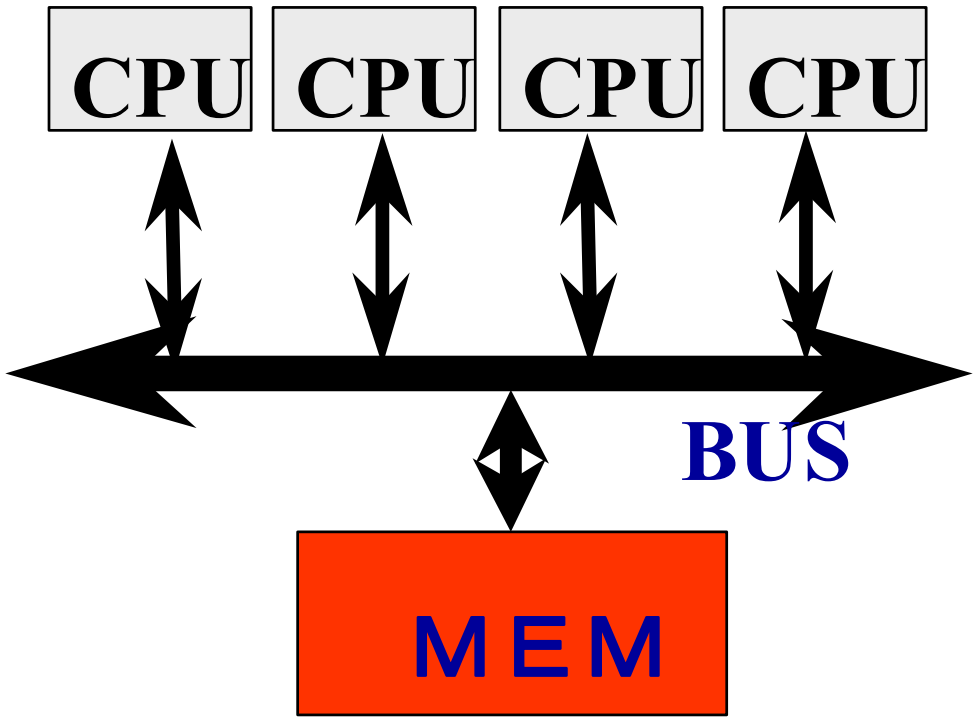


Parallelism Improves Performance

4 times speedup by using 4 cores!



Shared-memory Multicore Processor



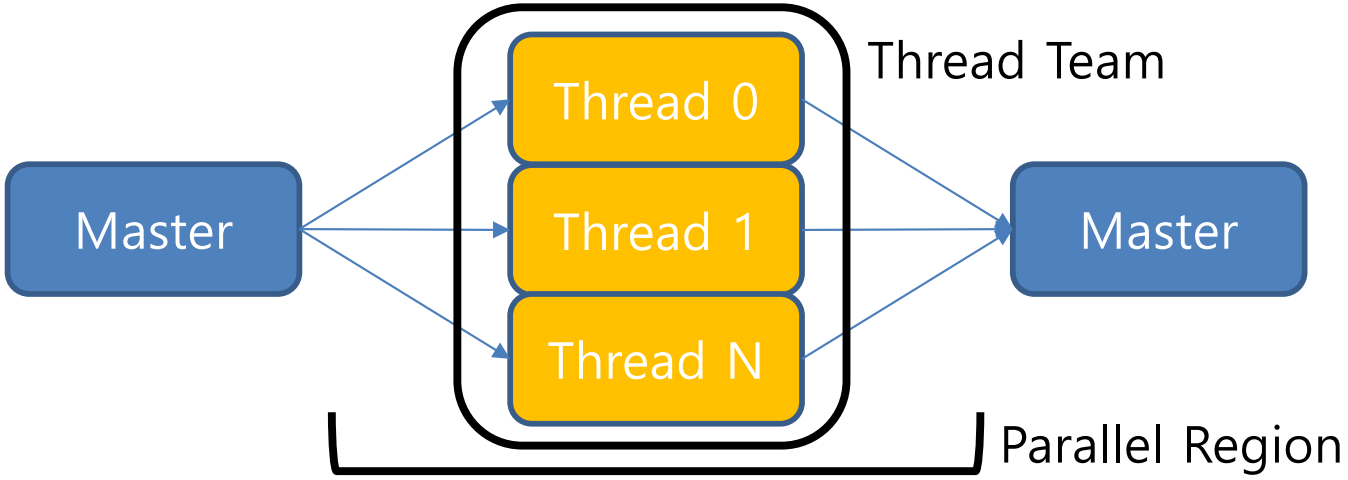
- Multiple cores share main memory
- Threads executed in each core(CPU) communicate with each other by accessing shared data in main memory.

parallel Directive

- starts parallel execution
- creates a **thread team**
- by default, all variables are shared among threads

```
  int x = 100;
#pragma omp parallel
{
  int my_thread_id = omp_get_thread_num();
  printf("[%d] x is %d¥n", my_thread_id, x);
  if (my_thread_id == 0) x = 0;
}
  printf("x is %d¥n", x);
```

Parallel Region



Compiling OpenMP Code

```
#include <stdio.h>
#include <omp.h>

int main(void) {
  int x = 100;
#pragma omp parallel
{
  int my_thread_id = omp_get_thread_num();
  printf("[%d] x is %d¥n", my_thread_id, x);
  if (my_thread_id == 0) x = 0;
}
  printf("x is %d¥n", x);
  return 0;
}
```

```
[jinpil]$ gcc -O3 -fopenmp test.c -o test
[jinpil]$ icc -O3 -qopenmp test.c -o test
[jinpil]$ clang -O3 -fopenmp -I$(HEADER) -L(LIB) -lomp test.c -o test
```

※ LLVM Clang provides the OpenMP library as an external module

Running OpenMP Code

```
[jinpil]$ gcc -O3 -fopenmp test.c -o test
[jinpil]$ OMP_NUM_THREADS=1
[jinpil]$ ./test
[0] x is 100
x is 0
[jinpil]$ OMP_NUM_THREADS=4
[jinpil]$ ./test
[0] x is 100
[2] x is 0
[3] x is 0
[1] x is 0
x is 0
[jinpil]$ ./test
[0] x is 100
[2] x is 0
[3] x is 100
[1] x is 100
x is 0
```

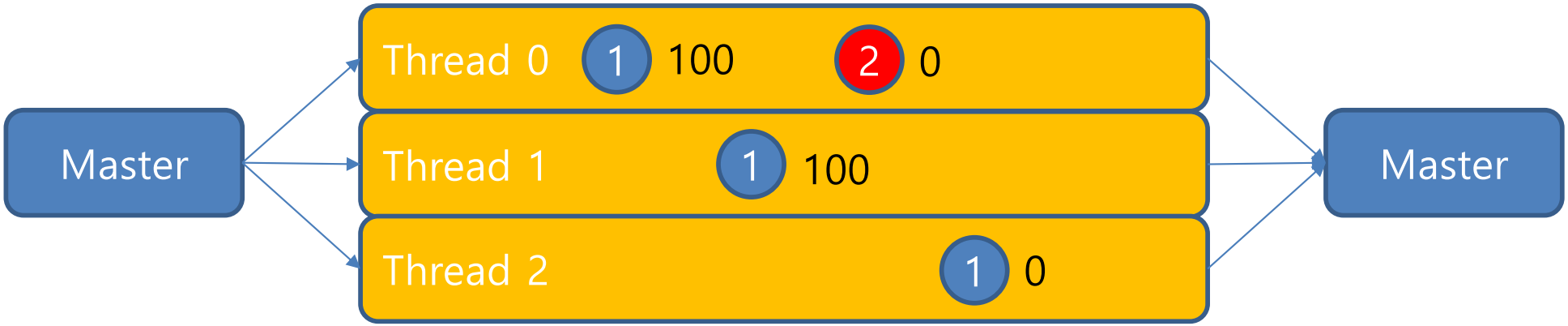
```
 int x = 100;
#pragma omp parallel
{
  int my_thread_id = omp_get_thread_num();
  printf("[%d] x is %d¥n", my_thread_id, x);
  if (my_thread_id == 0) x = 0;
}
  printf("x is %d¥n", x);
```

Data Race in Parallel Execution

- all threads do their workloads concurrently
- it can cause some unexpectable results

```
  int x = 100;
#pragma omp parallel
{
  int my_thread_id = omp_get_thread_num();
  printf("[%d] x is %d¥n", my_thread_id, x);
  if (my_thread_id == 0) x = 0;
}
  printf("x is %d¥n", x);
```

```
[jinpil]$ ./test
[0] x is 100
[1] x is 100
[2] x is 0
x is 0
```

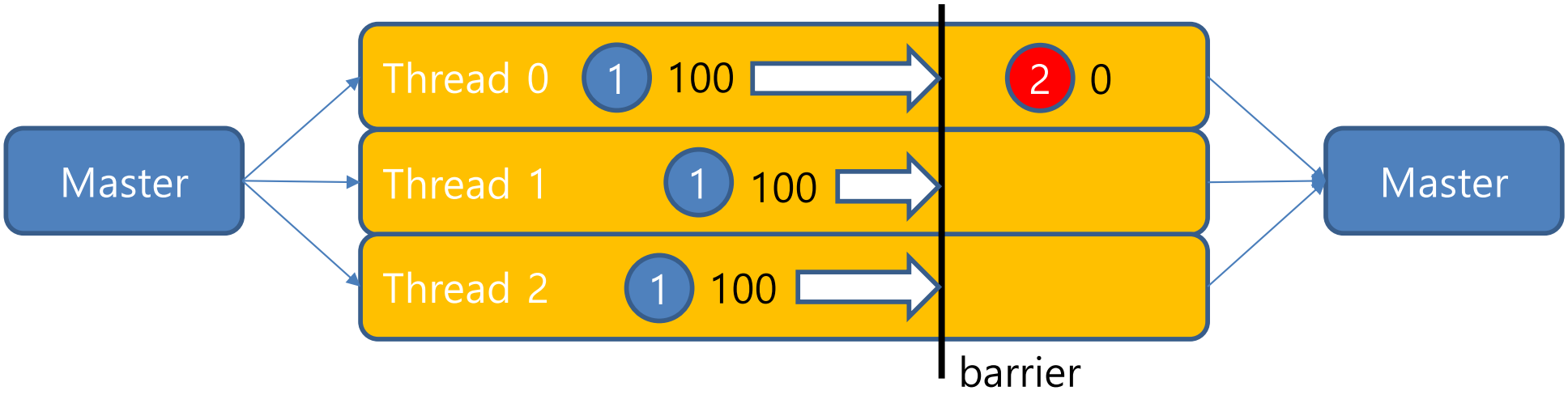


barrier Directive

- thread stops when they reach a barrier
- waits until all thread reach the barrier

```
  int x = 100;
#pragma omp parallel
{
  int my_thread_id = omp_get_thread_num();
  printf("[%d] x is %d¥n", my_thread_id, x);
#pragma omp barrier
  if (my_thread_id == 0) x = 0;
}
  printf("x is %d¥n", x);
```

```
[jinpil]$ ./test
[0] x is 100
[2] x is 100
[1] x is 100
x is 0
```

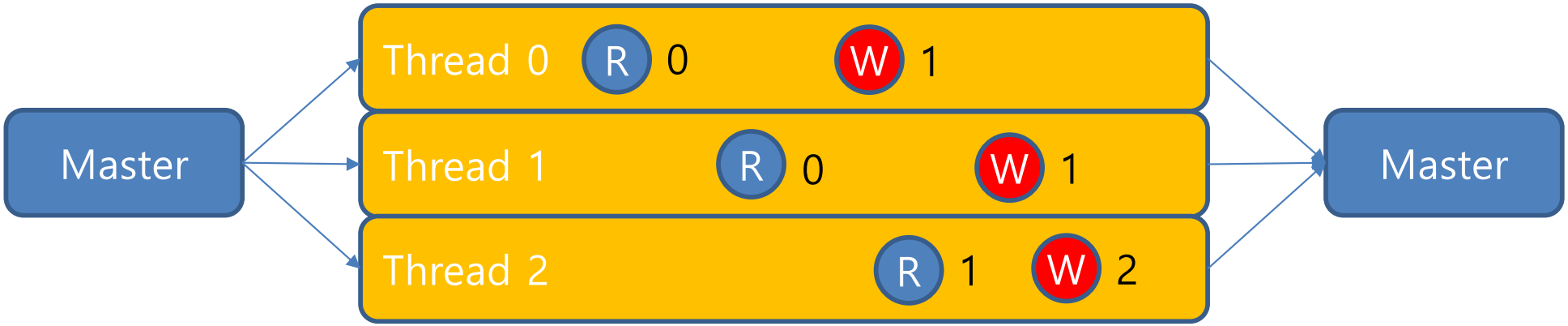


Another Data Race Problem

- be careful when writing shares variables
- because mostly statements are not **atomic**
  - each statement is divided into several instructions

```
  int sum = 0;
#pragma omp parallel
{
  sum++; // sum = sum + 1
}
  printf("sum is %d¥n", sum);
```

```
[jinpil]$ OMP_NUM_THREADS=128
[jinpil]$ ./test
sum is 121
[jinpil]$ ./test
sum is 120
```

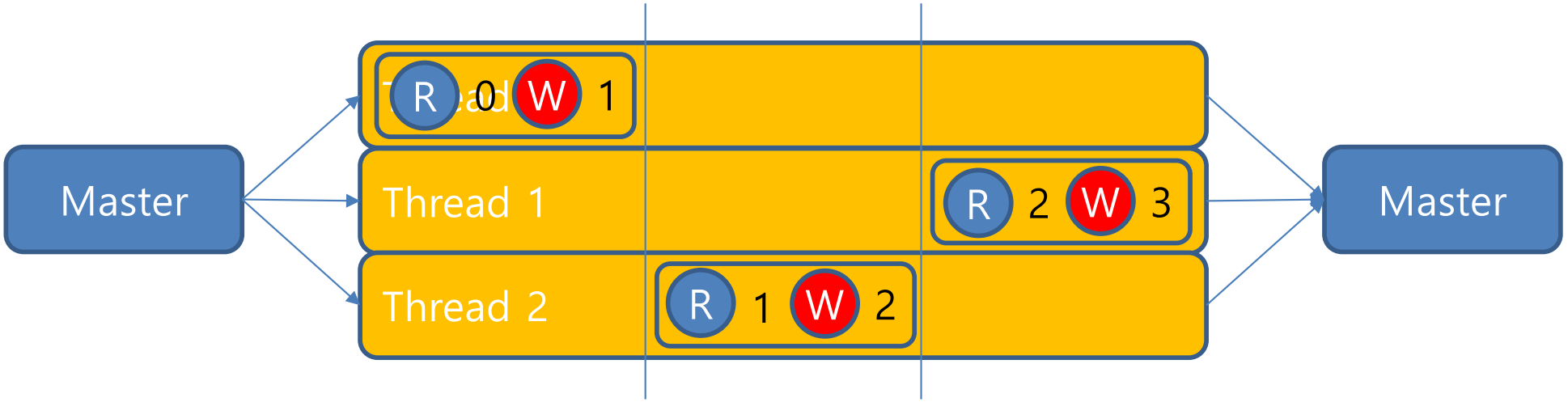


critical Directive

- make target structured block a **critical section**
- executed by a single thread at a time

```
  int sum = 0;
#pragma omp parallel
{
#pragma omp critical
{
  sum++;
}
}
  printf("sum is %d¥n", sum);
```

```
[jinpil]$ OMP_NUM_THREADS=128
[jinpil]$ ./test
sum is 128
[jinpil]$ ./test
sum is 128
```



Data Attribute Clauses

- by default, all variables are shared among threads
- some clauses create thread private variables

```
  int x = 100;
  int t_private, t_shared;
#pragma omp parallel private(t_private) firstprivate(x)
{
  int my_thread_id = omp_get_thread_num();
  t_private = my_thread_id;
  t_shared = my_thread_id;
#pragma omp barrier
  printf("[%d] private: %d | shared: %d | x: %d¥n",
         my_thread_id, t_private, t_shared, x);
}
```

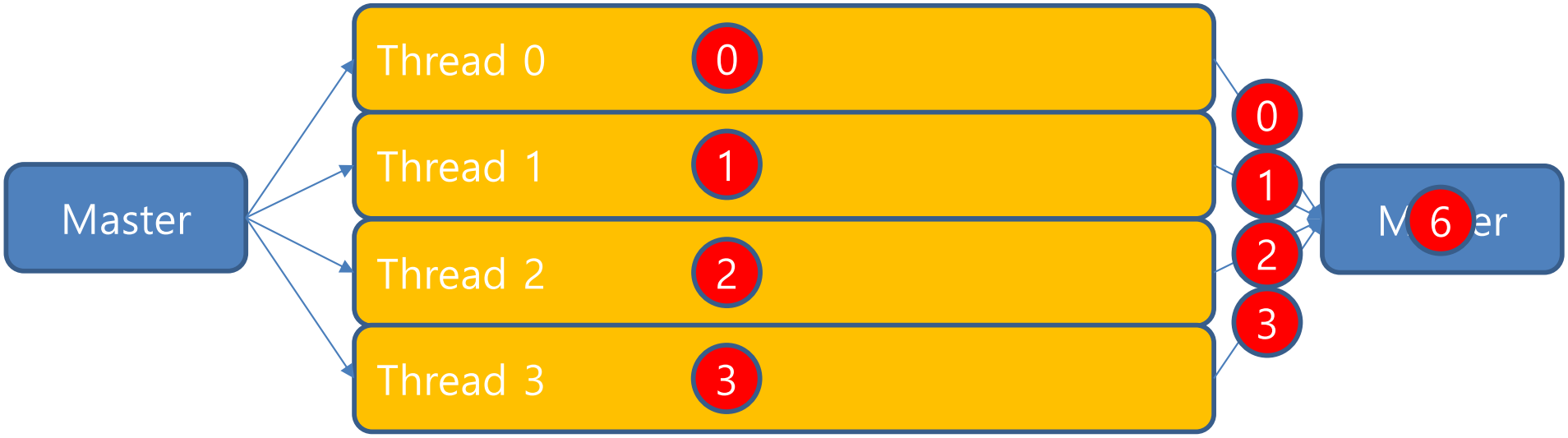
```
[jinpil]$ ./test
[0] private: 0 | shared: 1 | x: 100
[3] private: 3 | shared: 1 | x: 100
[2] private: 2 | shared: 1 | x: 100
[1] private: 1 | shared: 1 | x: 100
```

reduction Clause

- specified operation at the end of the parallel region
- op: +, *, -, &, |, ^, &&, ||, max, min
- reduction variables are private in the parallel region

```
  int value;
#pragma omp parallel reduction(+:value)
{
  value = omp_get_thread_num();
}
  printf("value is %d¥n", value);
```

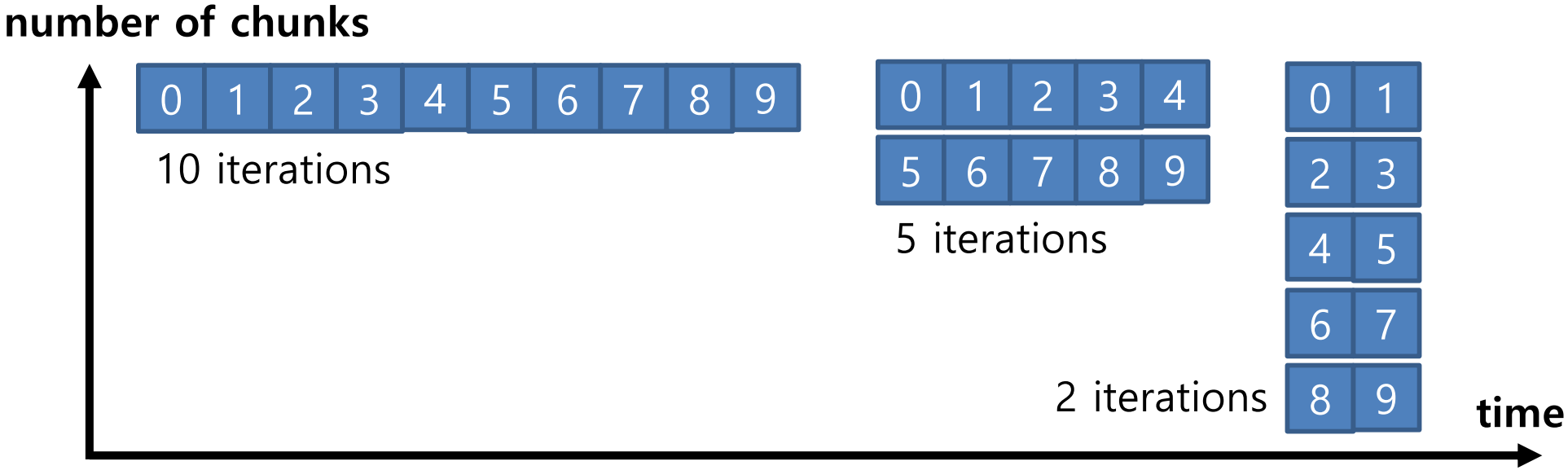
```
[jinpil]$ ./test
value is 6
```



Data Parallelism

- typical in computational science
- array references in a loop statement
- if each iteration can be executed independently, it can be **parallelized**

```
double A[10];
double B[10];
double C[10];
for (int i = 0; i < 10; i++) {
  C[i] = A[i] + B[i];
}
```



loop Directive

- targeting loop statement
  - **for** in C, **do** in Fortran
- work sharing in the current thread team
  - distribute iterations among threads
  - should be specified in a parallel region

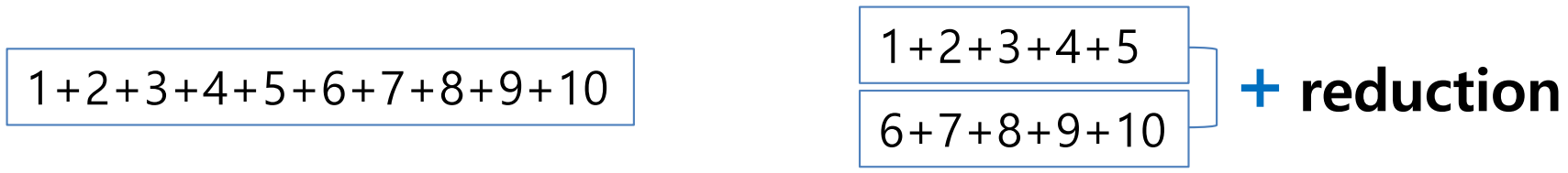
```
double A[10];
double B[10];
double C[10];
#pragma omp parallel
{
#pragma omp for
  for (int i = 0; i < 10; i++) {
    C[i] = A[i] + B[i];
  }
}
```

```
#pragma omp parallel for
  for (int i = 0; i < 10; i++) {
    C[i] = A[i] + B[i];
  }
```

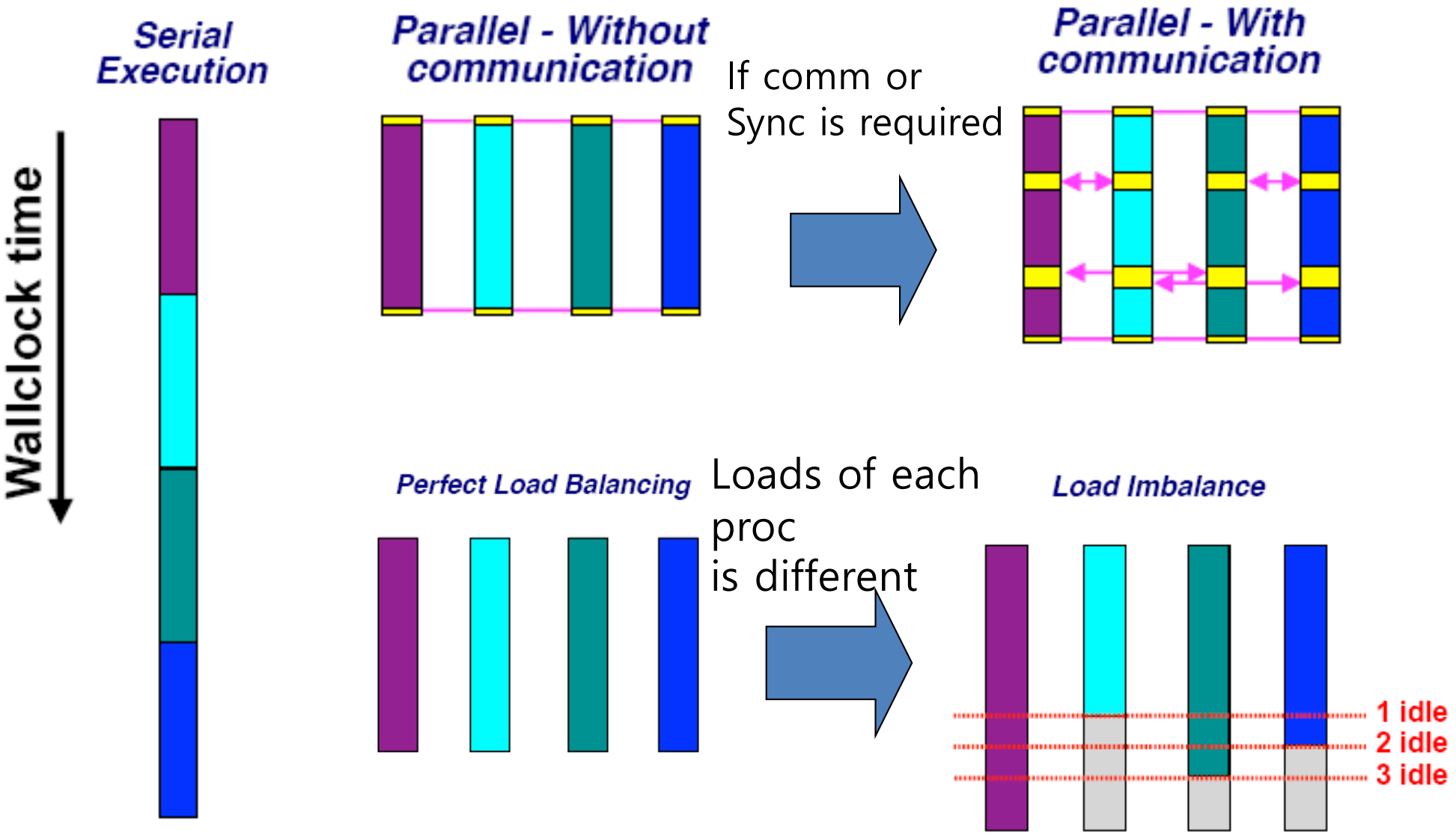
Clauses in loop Directive

- data attribute clauses
  - **private**, **firstprivate**, **lastprivate**
  - if a variable is accessed frequently, it should be a privatized
- **reduction** clause

```
  double A[N];
  double c = INIT_VALUE;
  double sum;
  int iter;
#pragma omp parallel for firstprivate(c) reduction(+:sum) lastprivate(iter)
  for (int i = 0; i < N; i++) {
    sum += (A[i] * c);
    iter = i;
  }
  // iter will be N-1 here
```



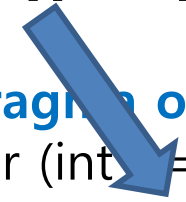
Overhead of Parallel Execution



Implicit Barrier

- implicit barrier at the end of **loop** directive
- **nowait** clause removes the implicit barrier
- barrier is needed when data dependency exists

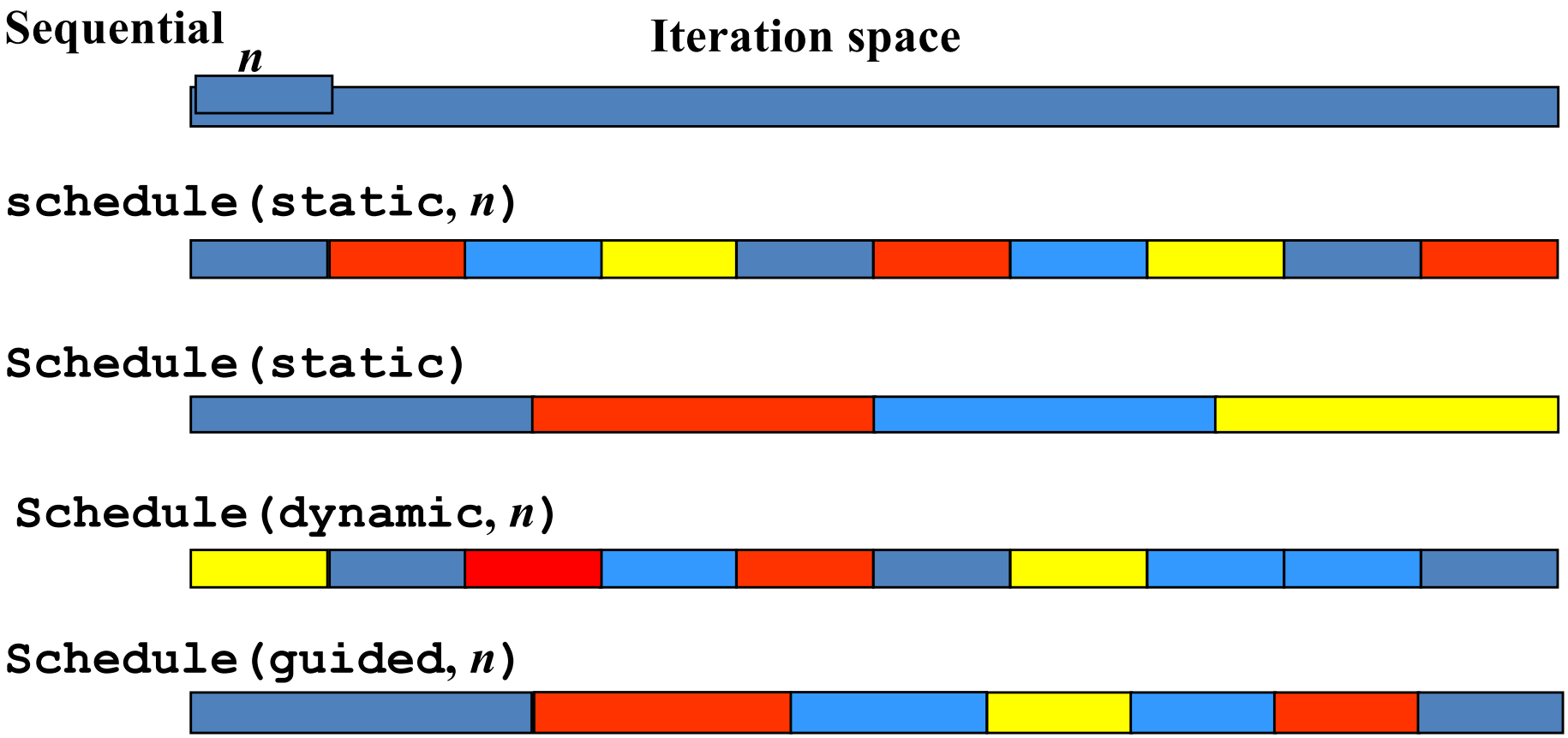
```
#pragma omp parallel
{
#pragma omp for
  for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
#pragma omp for
  for (int i = 0; i < N; i++) {
    E[i] = C[i] + D[i];
  }
}
```



```
#pragma omp parallel
{
#pragma omp for nowait
  for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
#pragma omp for
  for (int i = 0; i < N; i++) {
    F[i] = D[i] + E[i];
  }
}
```

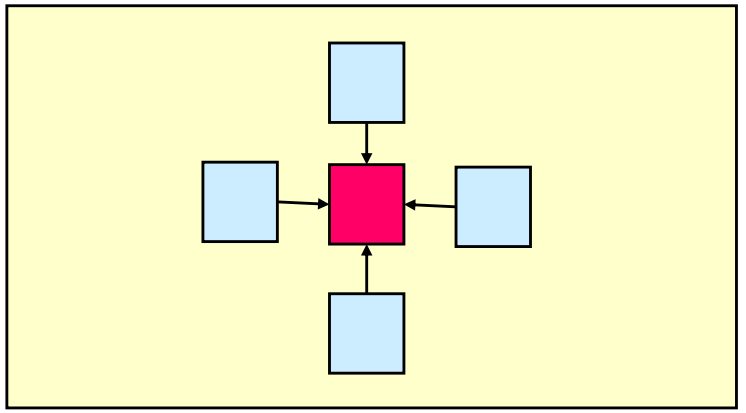
schedule Clause

- usually **static** shows good performance
- **dynamic** can be used to reduce load imbalance



Code Example: Laplace Solver

- Explicit solver of Laplace equation
  - Stencil operation: update value with 4-points of up/down/left/right.
  - Use array of “old” and “new”. Compute new by old and replace old with new.
  - Typical parallelization by domain decomposition
  - At each iteration, compute residual

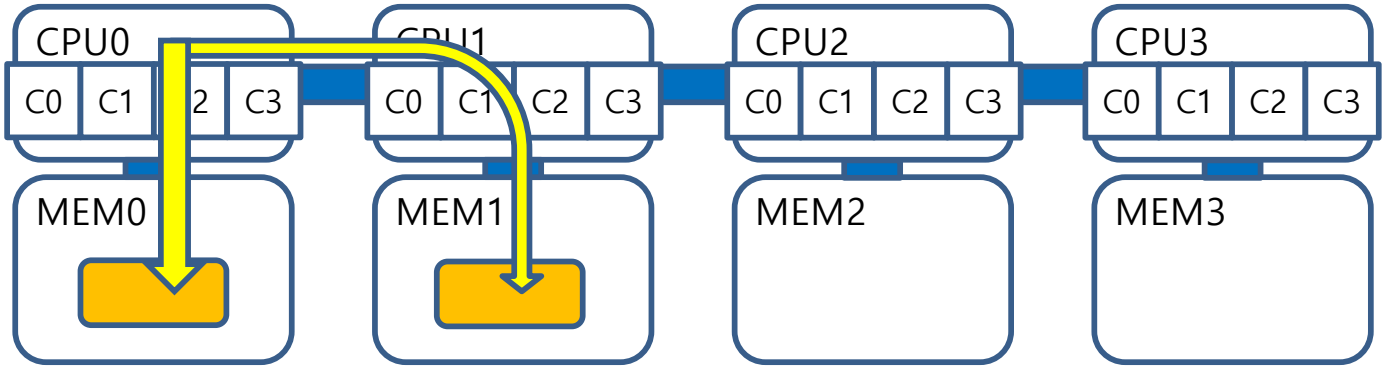


- OpenMP version
  - Parallelize 3 loops
    - OpenMP support only loop parallelization of outer loop.
  - For loop directive is orphan, in dynamic extent of parallel directive.

```
void lap_solve()
{
    int x,y,k;
    double sum;

#pragma omp parallel private(k,x,y)
{
    for(k = 0; k < NITER; k++){
        /* old <- new */
#pragma omp for
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            uu[x][y] = u[x][y];
        /* update */
#pragma omp for
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
    }
 }

/* check sum */
    sum = 0.0;
#pragma omp parallel for private(y) reduction(+:sum)
    for(x = 1; x <= XSIZE; x++)
        for(y = 1; y <= YSIZE; y++)
          sum += (uu[x][y]-u[x][y]);
    printf("sum = %g¥n",sum);
}
```

NUMA Architecture

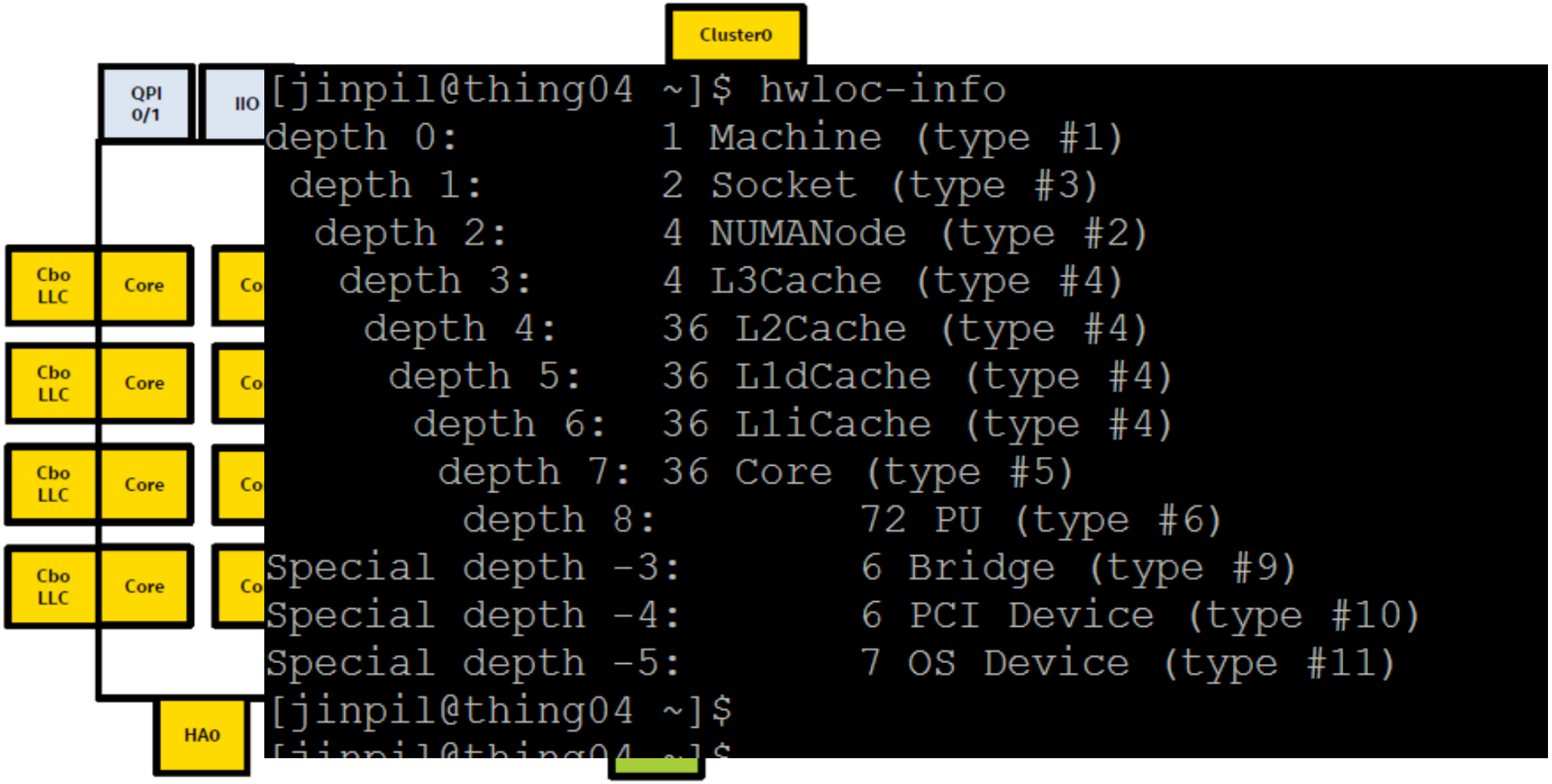
- Non-Uniform Memory Access (NUMA)
  - each NUMA node has dedicated memory
  - can access remote memory
  - access cost is not uniform
- current trends
  - improves performance by increasing NUMA nodes
  - NUMA nodes in a single processor
- parallel program should be optimized for NUMA
  - **data locality** is important on NUMA architecture



NUMA Node in a Processor

- Cluster-on-Die (COD) Mode in Xeon Processor
- Knights Landing Architecture has 4 NUMA nodes

COD Mode for 18C E5-2600 v3




```
[jinpil@thing04 ~]$ hwloc-info
depth 0:          1 Machine (type #1)
 depth 1:         2 Socket (type #3)
  depth 2:        4 NUMANode (type #2)
   depth 3:       4 L3Cache (type #4)
    depth 4:      36 L2Cache (type #4)
     depth 5:     36 L1dCache (type #4)
      depth 6:    36 L1iCache (type #4)
       depth 7:   36 Core (type #5)
        depth 8:          72 PU (type #6)
Special depth -3:         6 Bridge (type #9)
Special depth -4:         6 PCI Device (type #10)
Special depth -5:         7 OS Device (type #11)
[jinpil@thing04 ~]$
```

NUMA Optimization in OpenMP

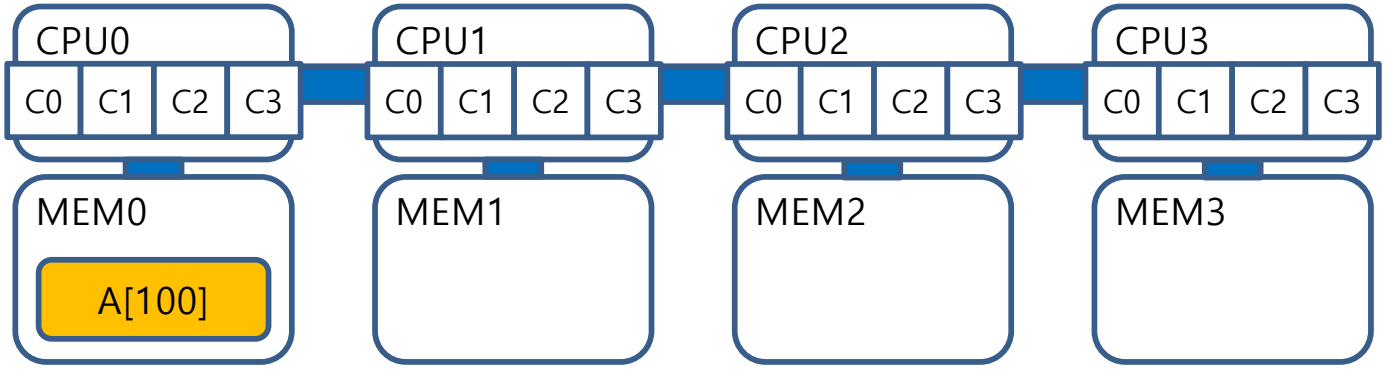
- STREAM Triad: benchmark for memory bandwidth
- key is how to initialize data
- *ser init*: no OMP directive in init()
- *par init*: using OMP directive in init()

```
void init() {
#pragma omp parallel for
  for (int j=0; j<N; j++) {
    b[j] = create_value_b(j);
    c[j] = create_value_c(j);
  }
}

void STREAM_Triad(double scalar) {
#pragma omp parallel for
  for (int j=0; j<N; j++)
    a[j] = b[j]+scalar*c[j];
}
```

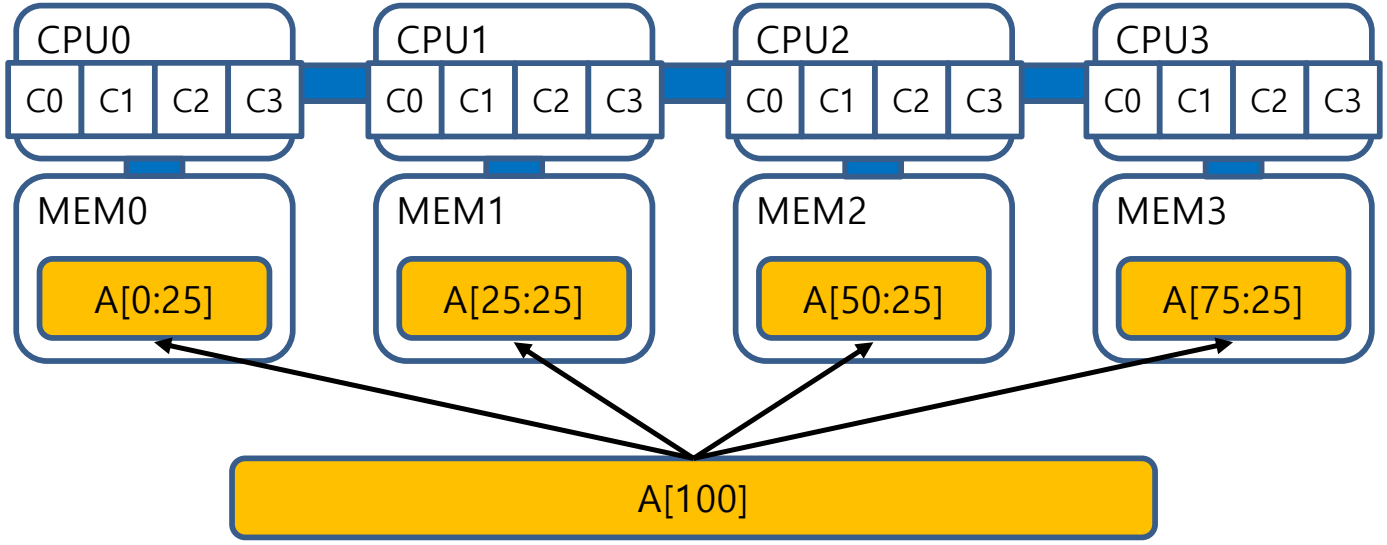
Linux Fist-touch Memory Allocation: Memory allocation happens at the first touch. The memory region will be allocated close to the accessing CPU.

Data Distribution with First-touch Allocation

- *ser init*: all data will allocated on NUMA node 0

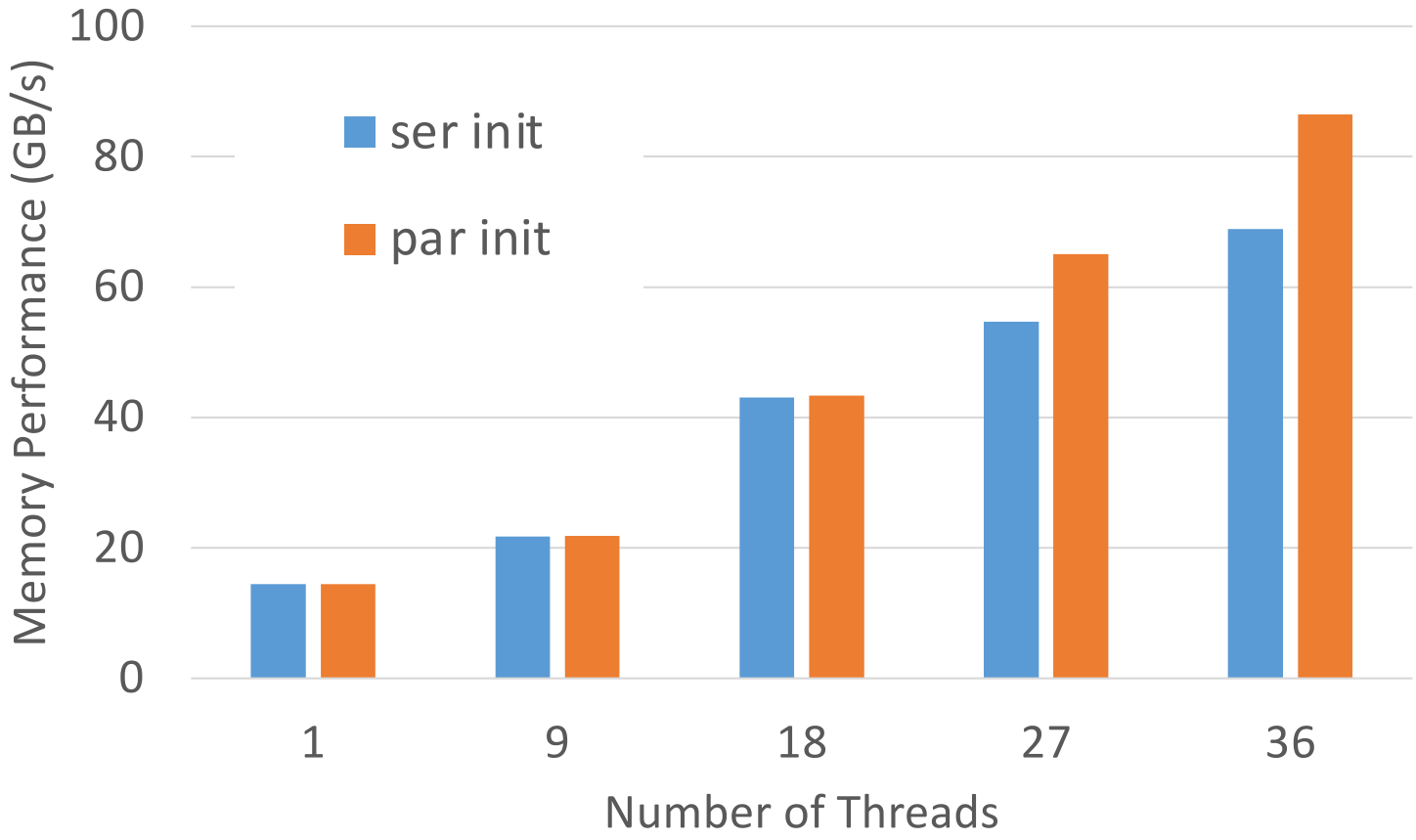


- *par init*: data will distributed among NUMA nodes



Performance: STREAM Triad

- *ser init*: all data is on NUMA node 0
  - remote memory access on NUMA node 1, 2, ...
- *par init*: all data is on each local memory
  - no remote memory access

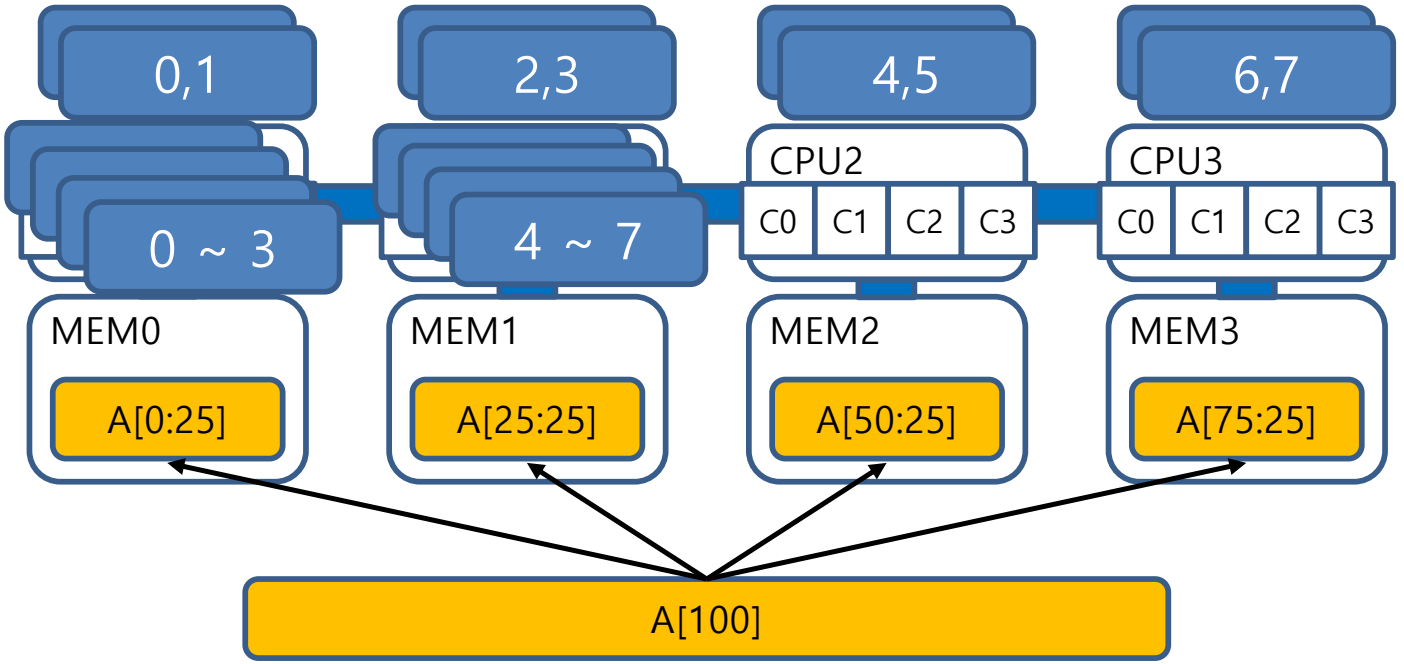


Thread Affinity

- OpenMP thread is equal to a physical core
- how to assign it to a physical core (**thread affinity**) is run time dependent
- you can control it by using **OMP_BIND_PROC** / **OMP_PLACES**

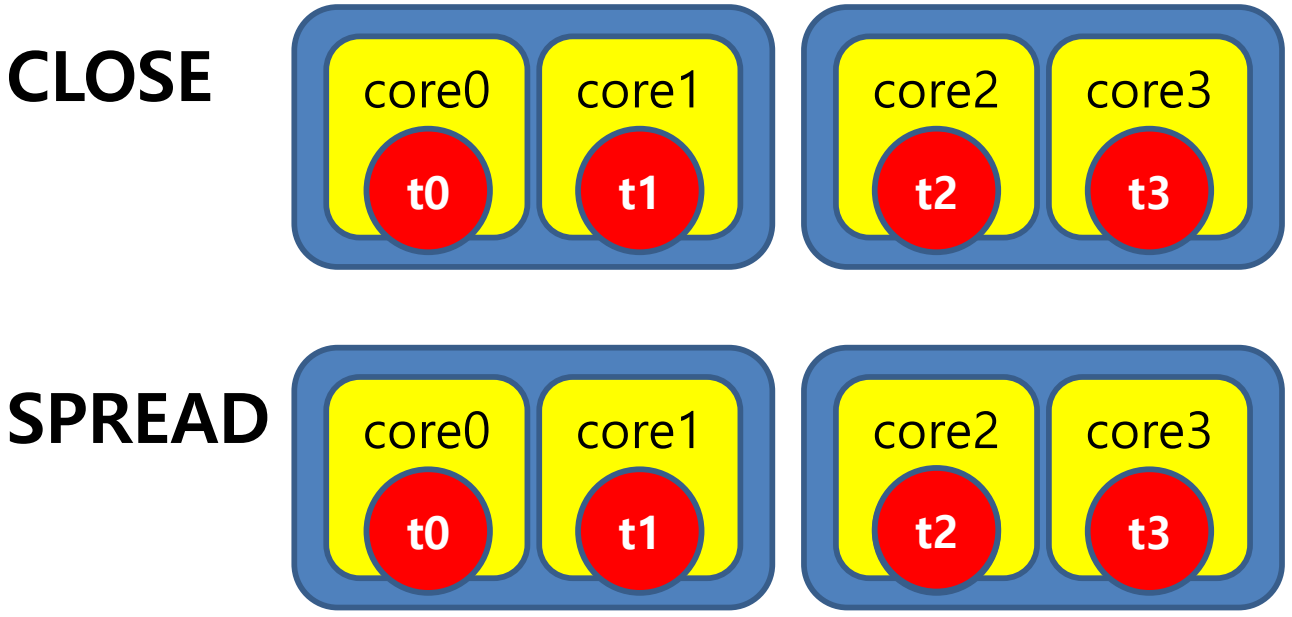
SPREAD

CLOSE



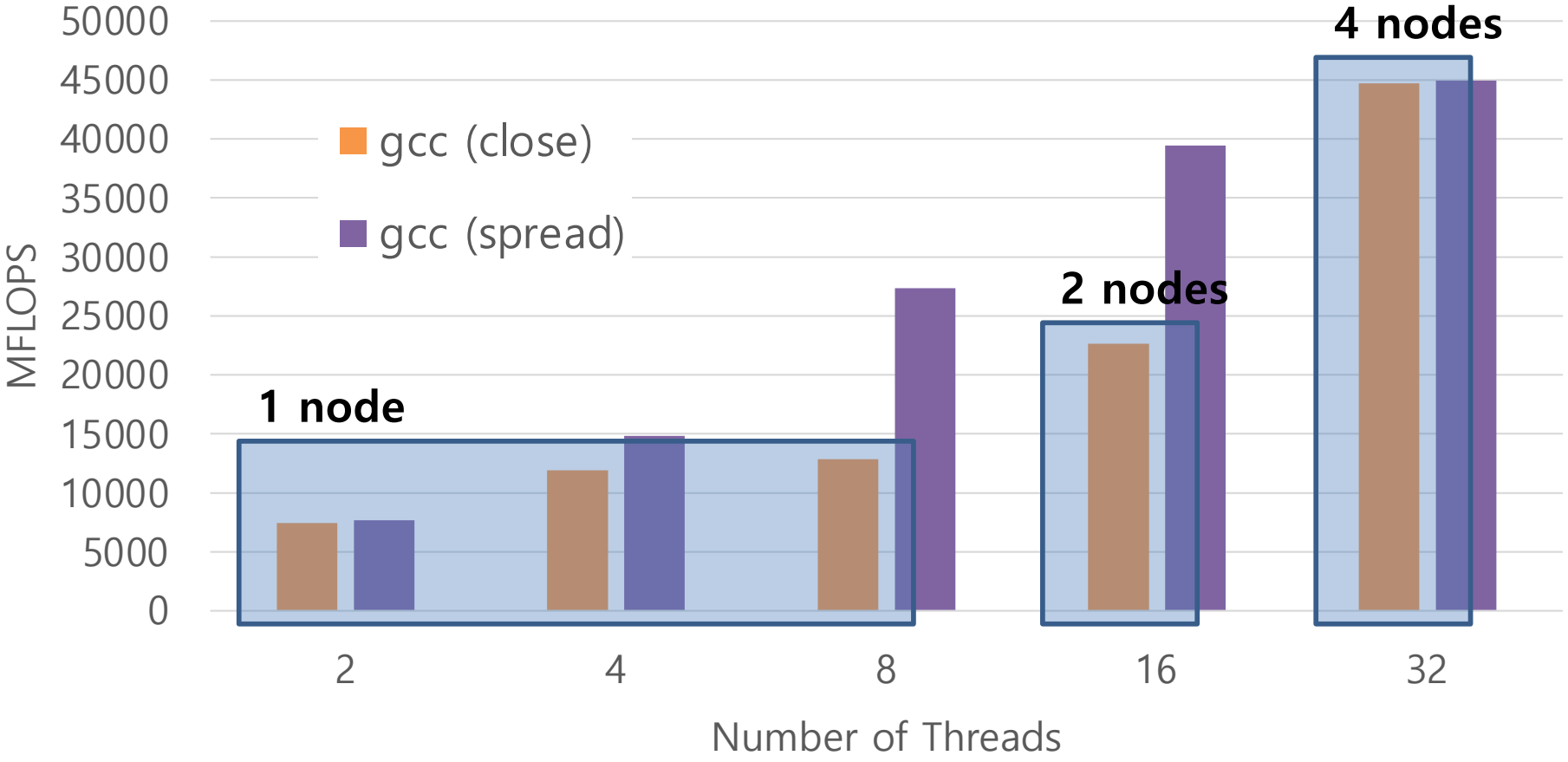
Thread Affinity Option in OpenMP

- can be specified via environment variables
- **OMP_PROC_BIND**
  - CLOSE | SPREAD
  - **proc_bind** clause available in parallel directive
- **OMP_PLACES**
  - THREADS | CORES | SOCKETS



Performance Impact with Thread Affinity

- comparison between CLOSE / SPREAD
- SPREAD uses more NUMA nodes
- Target Platform: 9 cores per NUMA node

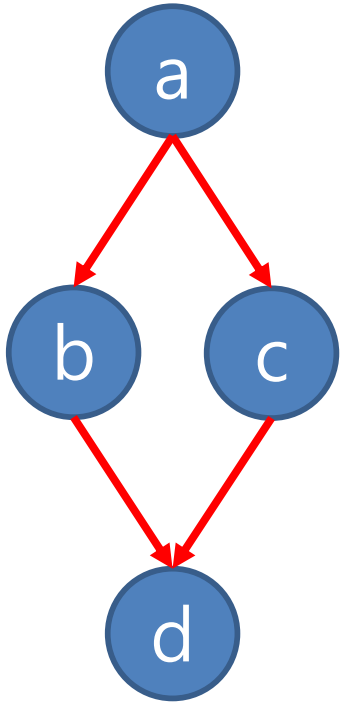


NAS Parallel Benchmarks MG Kernel

task Directive

- task-parallelism
- depend clauses specify data dependency
- tasks are created/scheduled in the parallel region

```
#pragma omp parallel
{
#pragma omp single
{
#pragma omp task depend(out:a)
  a = task_a();
#pragma omp task depend(in:a) depend(out:b)
  b = task_b(a);
#pragma omp task depend(in:a) depend(out:c)
  c = task_c(a);
#pragma omp task depend(in:b, c) depend(out:d)
  d = task_d()
}
}
```

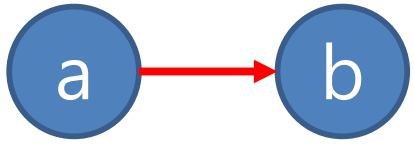


Task Dependency

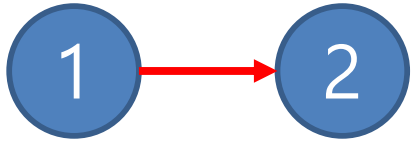
- OpenMP preserves the sequential code semantics
- all task dependency should be given properly

```
#pragma omp task depend(out:a)
  a = task_a();
#pragma omp task depend(in:b)
  c = task_c(b);
```


```
#pragma omp task depend(out:a)
  a = task_a();
#pragma omp task depend(in:a)
  b = task_b(a);
```




```
#pragma omp task depend(out:a)
  a = func1();
#pragma omp task depend(out:a)
  a = func2();
```



Task Synchronization

- use **taskwait** directive
- **barrier** directive has implicit task synchronization

```
#pragma omp parallel
{
  while (i < NITER) {
#pragma omp single
{
#pragma omp task
    task1()
#pragma omp task
    task2()
}
#pragma omp taskwait
  } // while()
} // omp parallel
```

```
#pragma omp parallel
{
  while (i < NITER) {
#pragma omp single
{
#pragma omp task
    task1()
#pragma omp task
    task2()
}
#pragma omp barrier
  } // while()
} // omp parallel
```

Code Example: Block Cholesky Decomposition

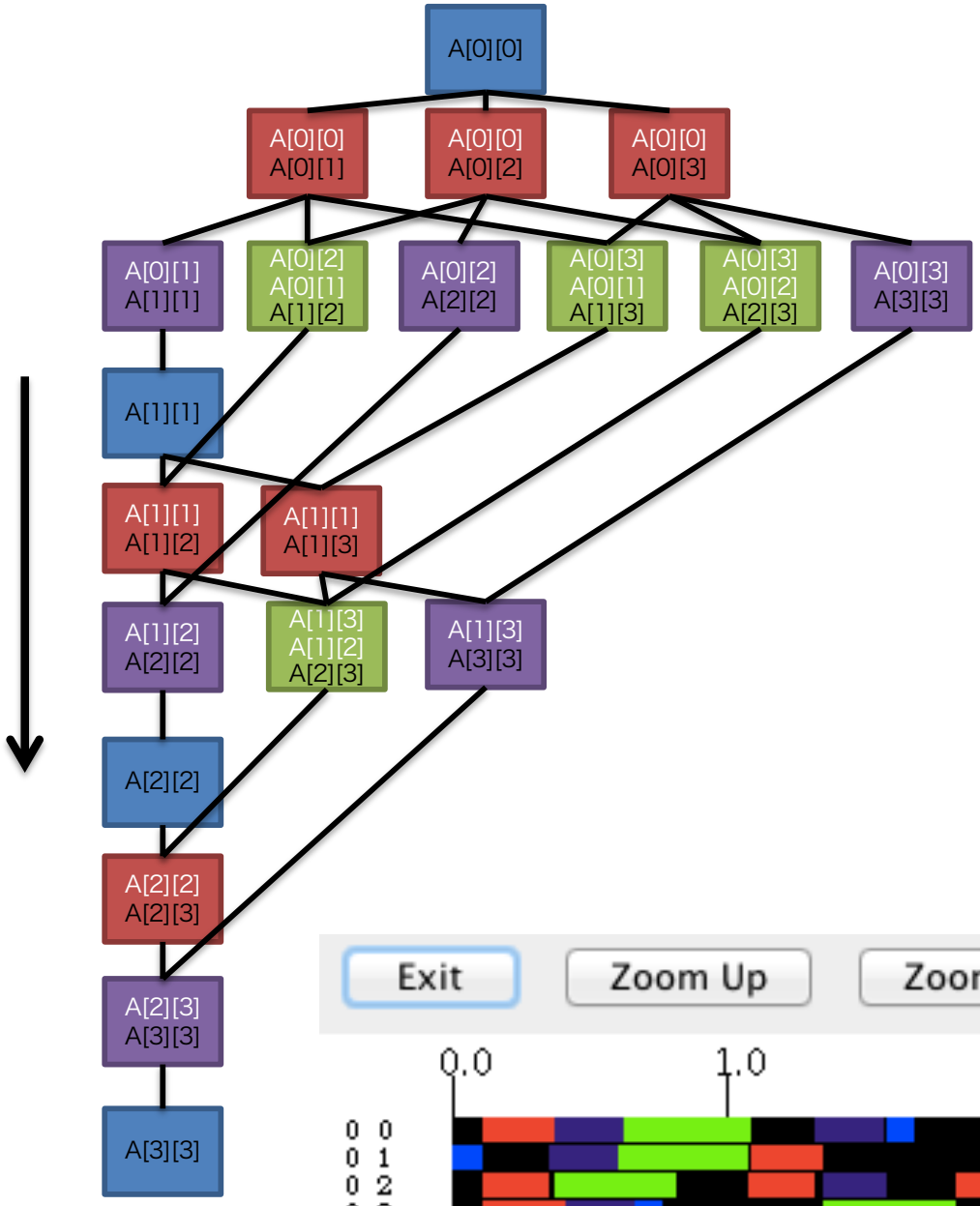
- irregular workload
- hard to parallelize with the **parallel for** directive
- requires dynamic load balancing → task-parallelism

```
for (int k = 0; k < nt; k++) {
  omp_potrf (A[k][k], ts, ts);

  for (int i = k + 1; i < nt; i++) {
    omp_trsm (A[k][k], A[k][i], ts, ts);
  }
  for (int i = k + 1; i < nt; i++) {
    for (int j = k + 1; j < i; j++) {
      omp_gemm (A[k][i], A[k][j], A[j][i], ts, ts);
    }
    omp_syrk (A[k][i], A[i][i], ts, ts);
  }
}
```

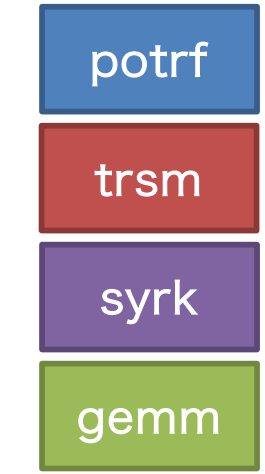
Code Example: Block Cholesky Decomposition

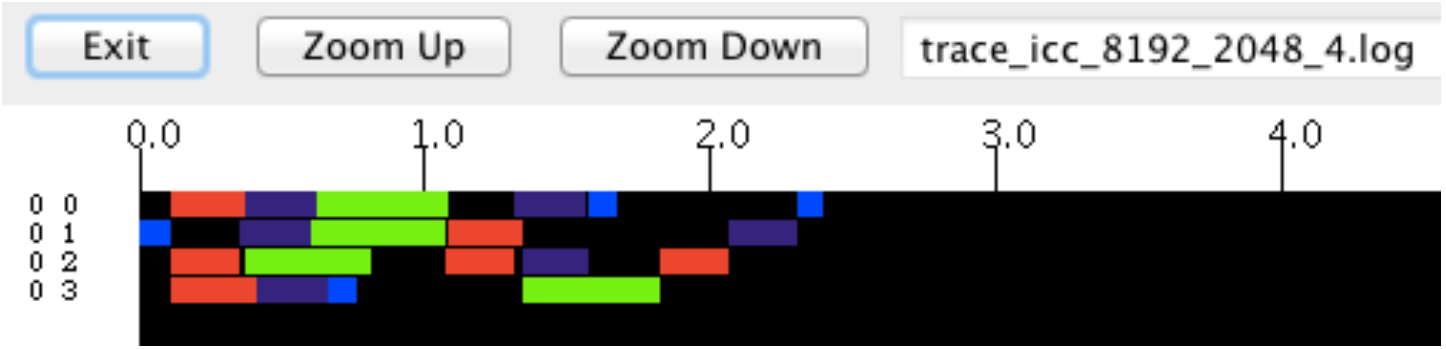
```
#pragma omp parallel
#pragma omp single
  for (int k = 0; k < nt; k++) {
#pragma omp task depend(inout:A[k][k])
    omp_potrf (A[k][k], ts, ts);

    for (int i = k + 1; i < nt; i++) {
#pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
      omp_trsm (A[k][k], A[k][i], ts, ts);
    }
    for (int i = k + 1; i < nt; i++) {
      for (int j = k + 1; j < i; j++) {
#pragma omp task depend(in:A[k][i], A[k][j]) depend(inout:A[j][i])
        omp_gemm (A[k][i], A[k][j], A[j][i], ts, ts);
      }
#pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
      omp_syrk (A[k][i], A[i][i], ts, ts);
    }
  }
#pragma omp taskwait
```

Task Dependency Graph



black: inout
white: in
—— : dependency





History of SIMD Instructions

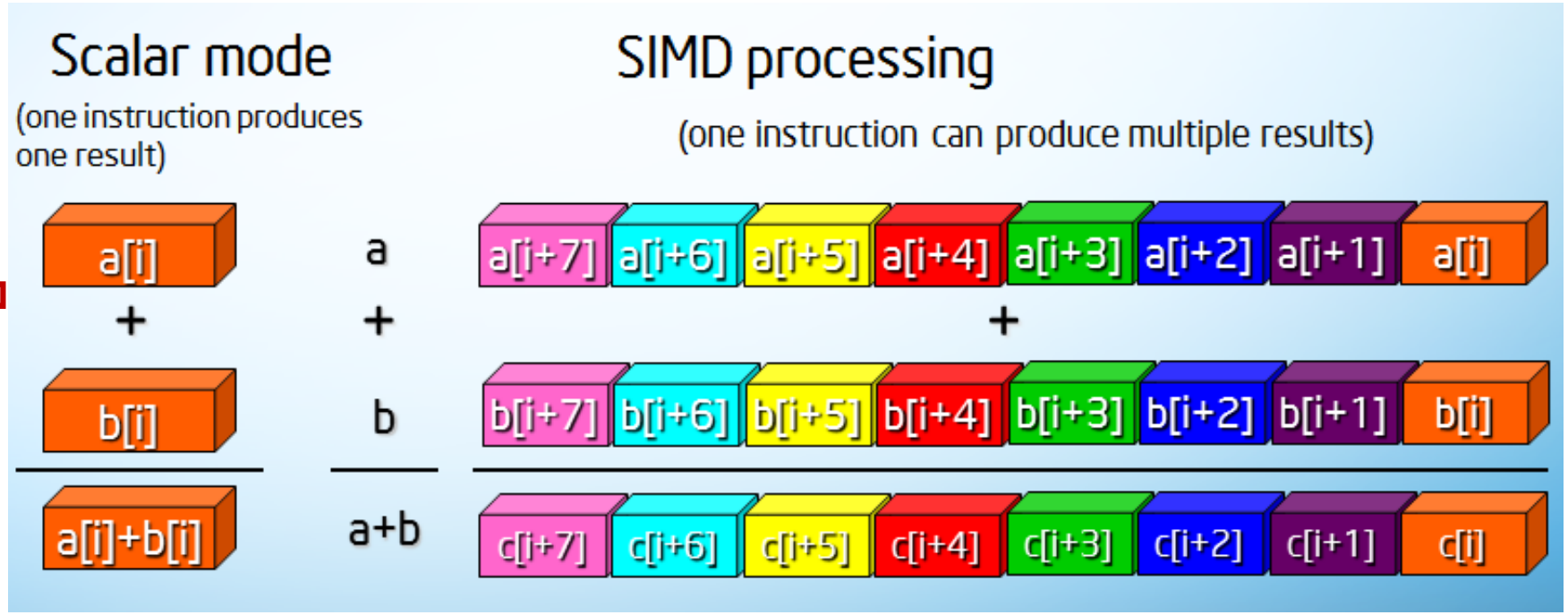
- **Intel MMX**
  - MMX Pentium (233 MHz)
  - 57 instructions for **2 32-bit integers (64-bit)**
- **AMD 3DNow!**
  - AMD K6-2 (300 MHz)
  - MMX + 21 instructions for **2 32-bit float values (64-bit)**

SIMD Instructions (cont'd)

- **Intel SSE**
  - 128-bit SIMD
  - supports 32-bit floating point operations
- **Intel SSE2**
  - added 64-bit floating point operations
- **Intel SSE3**
  - added DSP-oriented, complex operations
- **Intel SSE4**
  - added a dot-product instruction, etc...
- **ARM Advanced SIMD (NEON)**
  - DSP-oriented SIMD instruction
  - 64-bit/128-bit SIMD
  - supports 32-bit floating point operations

How SIMD works

- calculate multiple values in one instruction



Intel Developer Zone

SIMD Vectorization

- calculate multiple values in one instruction
- most fine-grain level parallelism
- instruction level parallelism
- essential to exploit the ***core*** performance
  - Intel Xeon Phi 7290 (Knights Landing)
  - AVX-512, 72 cores
  - single core (NO SIMD)
    2 ALU × 2 OPs × 1.5 GHz = 6 GFLOPS
  - single core + **SIMD** (64-bit FP)
    8 FPs × 2 ALU × 2 OPs × 1.5 GHz
    = 48 GFLOPS
  - **multiple cores** + **SIMD** (64-bit FP)
    72 cores × **8 FPs** × 2 ALU × 2 OPs × 1.5 GHz
    = 3,456 GFLOPS

How to Program: Assembly Language

- most efficient
- least productive
- instruction set dependent

```
float A[1000];
float B[1000];
float C[1000];

void foo() {
  for (i = 0; i < 1000; i++) {
    C[i] = A[i] + B[i];
  }
}
```

```
.L3:
  movaps A(%rax), %xmm0
  addq $16, %rax
  addps B-16(%rax), %xmm0
  movaps %xmm0, C-16(%rax)
  cmpq $4000, %rax
  jne .L3
```

How to Program: Intrinsic Function

- better productivity
- equivalent performance to assembly languages
- still difficult to use
- instruction set dependent

```
float A[1000];
float B[1000];
float C[1000];

void foo() {
  for (i = 0; i < 1000; i++) {
    C[i] = A[i] + B[i];
  }
}
```

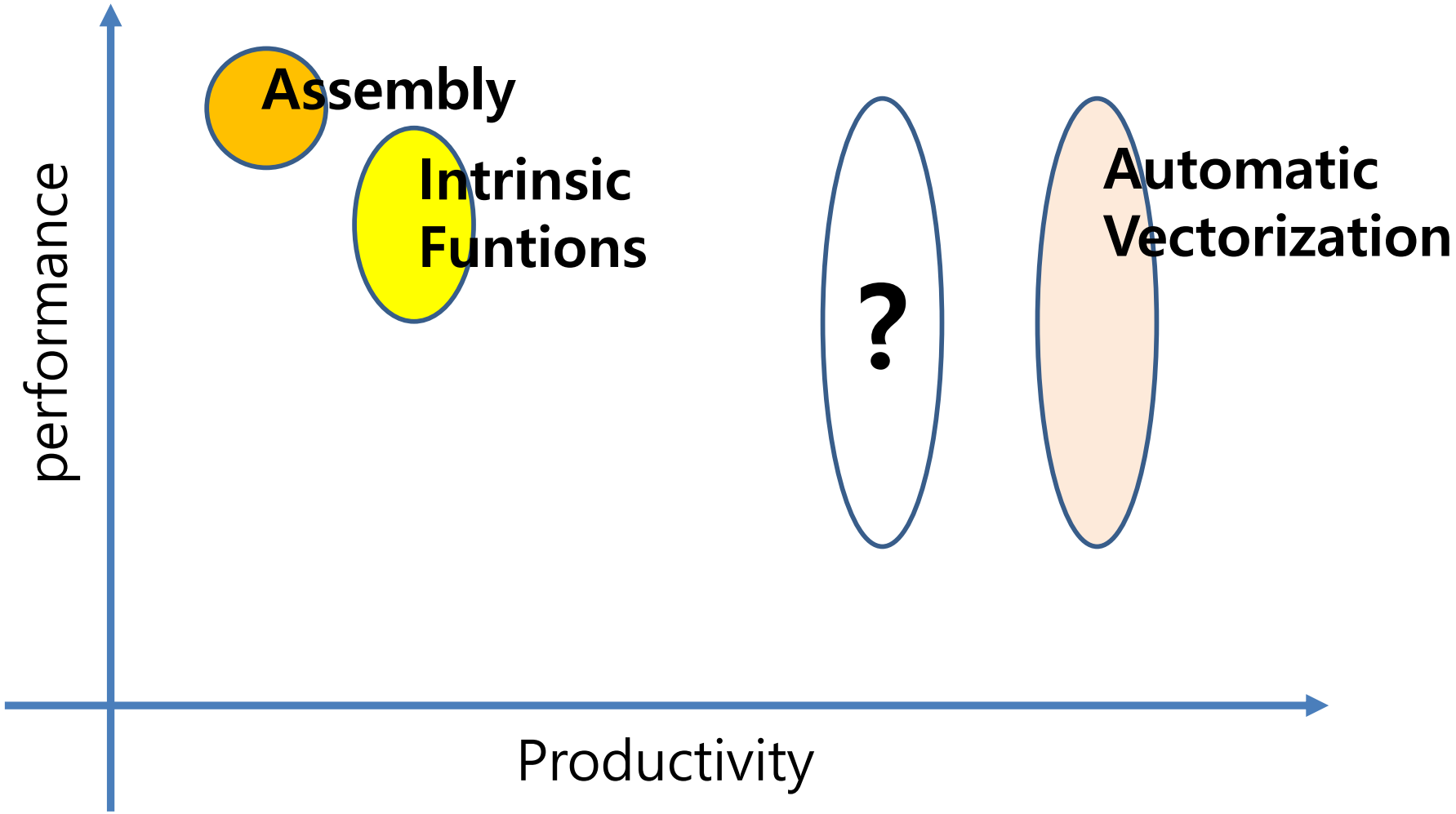
```
float A[1000];
float B[1000];
float C[1000];

void foo() {
  for (i = 0; i < 250; i++) {
    __m128 t0, t1;
    t0 = _mm_load_ps(&A[i]);
    t1 = _mm_load_ps(&B[i]);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(&C[i], t0);
  }
}
```

How to Program: Auto Vectorization

- program in high-level languages (C, C++, Fortran)
- compiler generates vector code
- most productive
- difficult to optimize

```
float A[1000];
float B[1000];
float C[1000];

void foo() {
  for (i = 0; i < 1000; i++) {
    C[i] = A[i] + B[i];
  }
}
```

Productivity & Performance



How to Program: OpenMP

- program in high-level languages (C, C++, Fortran)
- compiler generates vector code ...
- according to **OpenMP directives**
- **explicit** SIMD programming in C/C++, Fortran

```
float A[1000];
float B[1000];
float C[1000];

void foo() {
  for (i = 0; i < 1000; i++) {
    C[i] = A[i] + B[i];
  }
}
```

```
float A[1000];
float B[1000];
float C[1000];

void foo() {
#pragma omp simd
  for (i = 0; i < 1000; i++) {
    C[i] = A[i] + B[i];
  }
}
```

simd Directive

- explicitly specifying loop vectorization
- provides hidden information for SIMD vectorization

```
extern double *a;
extern double *b;
extern double *c;

for (int i = 0; i < 128; i++) {
  c[i] += a[i] * b[i];
}
```

```
vmovupd a+1024(%rcx), %ymm1
vmulpd  b+1024(%rcx), %ymm1, %ymm1
vaddpd  c+1024(%rcx), %ymm1, %ymm1
vmovupd %ymm1, c+1024(%rcx)
```

or it may not be vectorized...

```
extern double *a;
extern double *b;
extern double *c;

#pragma omp simd aligned(a,b,c:32)
for (int i = 0; i < 128; i++) {
  c[i] += a[i] * b[i];
}
```

```
vmovapd (%rcx,%rdi,8), %ymm1
vmulpd  (%rdx,%rdi,8), %ymm1, %ymm1
vaddpd  (%rsi,%rdi,8), %ymm1, %ymm1
vmovapd %ymm1, (%rsi,%rdi,8)
```

Clauses in simd Directive

- data attribute clauses
  - **private**, **firstprivate**
- **reduction** clause
- **aligned** clause
- **parallel for simd** → thread parallel + SIMD vectorization

- should make memory allocation aligned by using:
  - posix_memalign(), aligned_alloc()

```
extern double *A;
extern double *B;
double sum;
#pragma omp parallel for simd reduction(+:sum) aligned(A,B:32)
for (int i = 0; i < N; i++) {
  sum += (A[i] + B[i]);
}
```

declare simd Directive

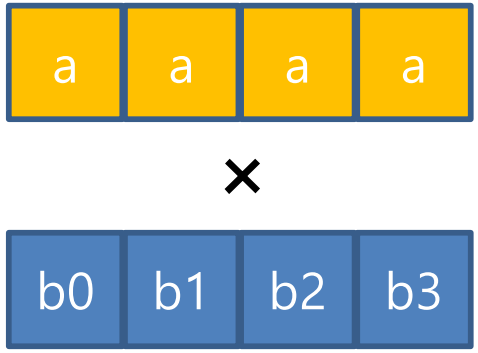
- whole function vectorization
- scalar function → vector function
- should be used in a SIMD-vectorized loop

```
extern double *a;
extern double *b;
extern double *c;

#pragma omp declare simd notinbranch
double mult(double a, double b) {
  return a * b;
}

#pragma omp simd aligned(a,b,c:32)
for (int i = 0; i < 1024; i++) {
  c[i] = mult(a[i], b[i]);
}
```

Clauses in declare simd Directive

- **notinbranch** clause
- **uniform** clause
- **linear** clause

```
#pragma omp declare simd notinbranch uniform(a)
double mult(double a, double b) {
  return a * b;
}

extern double a;
extern double *b;
extern double *c:

#pragma omp simd aligned(b,c:32)
for (int i = 0; i < 1024; i++) {
  c[i] = mult(a, b[i]);
}
```



About simd Directive

- multithreading will improve performance instantly
- using simd directive is not straightforward
  - compiler already vectorized your code
  - most (simple) sequential code is memory-bound

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

```
for (int i = 0; i < N; i++) {
  for (int k = 0; k < N; k++) {
    for (int j = 0; j < N; j++) {
      C[i][j] += A[i][k] * B[k][j];
    }
  }
}
```
5.4 GFLOPS

```
for (int ii = b; ii < b+BLK; ii++) {
  for (int kk = b; kk < b+BLK; kk++) {
    for (int jj = b; jj < b + BLK; jj++) {
      C[ii][jj] += A[ii][kk] * B[kk][jj];
```
9.4 GFLOPS

still, 25% of
peak performance

Many-core Architecture

- low-power + many-core
- power efficient architecture
- first generation: accelerator

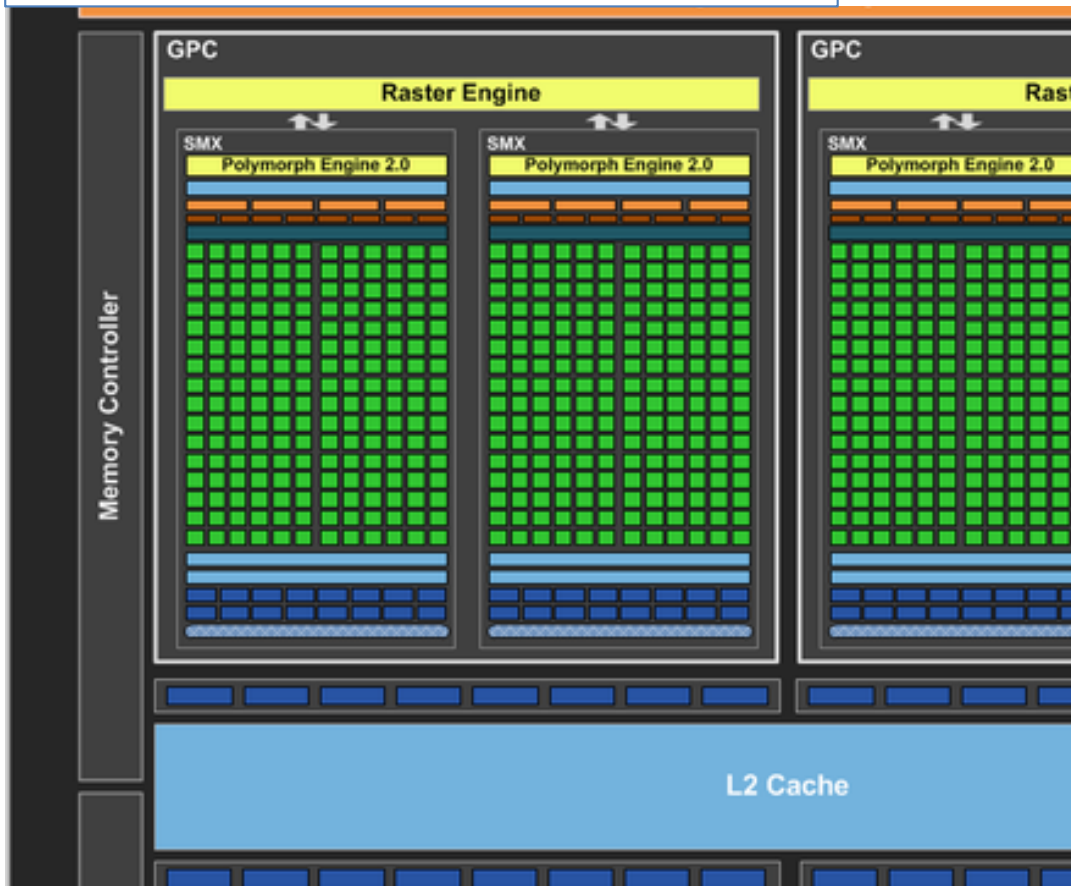
CPU
1~10 cores
General-purpose

Accelerator
Many-core
Power-efficient

Graphic Processing Unit

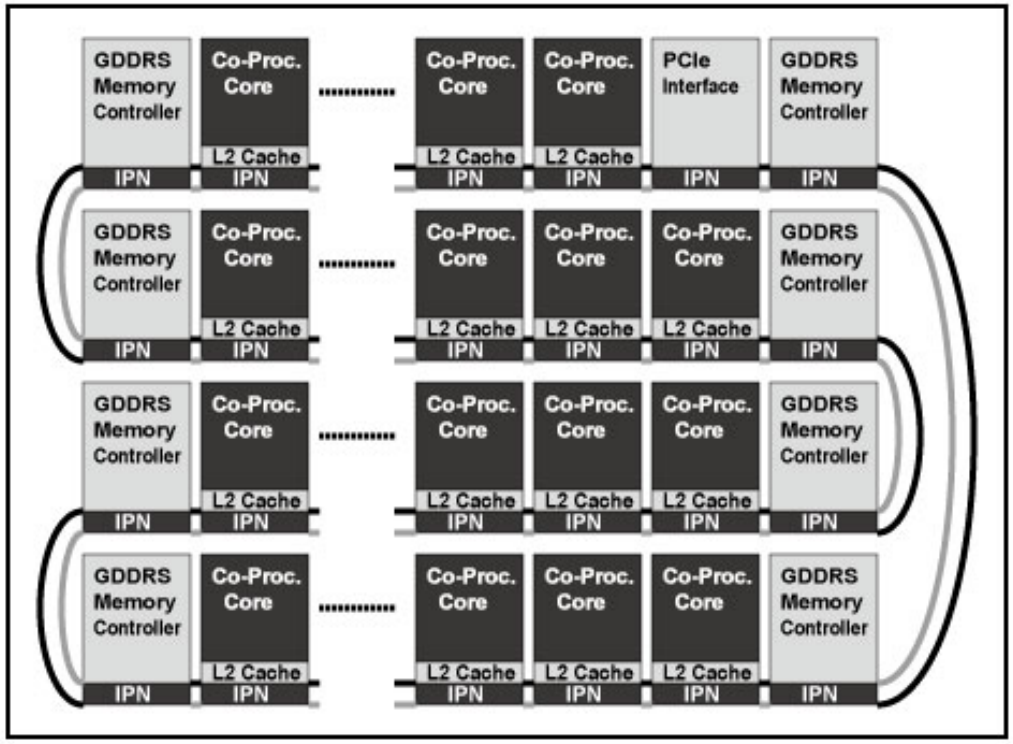
- originally for CG, gaming → programmable
- architecture dedicated to calculation
- high Watt/Flops

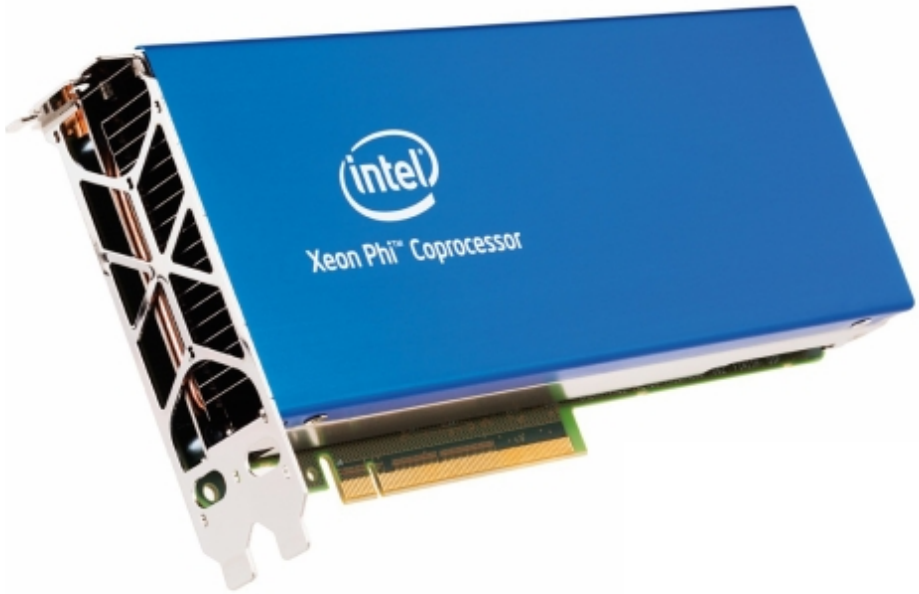
Architecture of NVIDIA GPU



Intel Xeon Phi

- Intel many-core architecture
- 60~ cores per chip
- PCI card as an accelerator
- compatible with Intel architecture
- Knights Landing Gen can be used as a stand-alone CPU

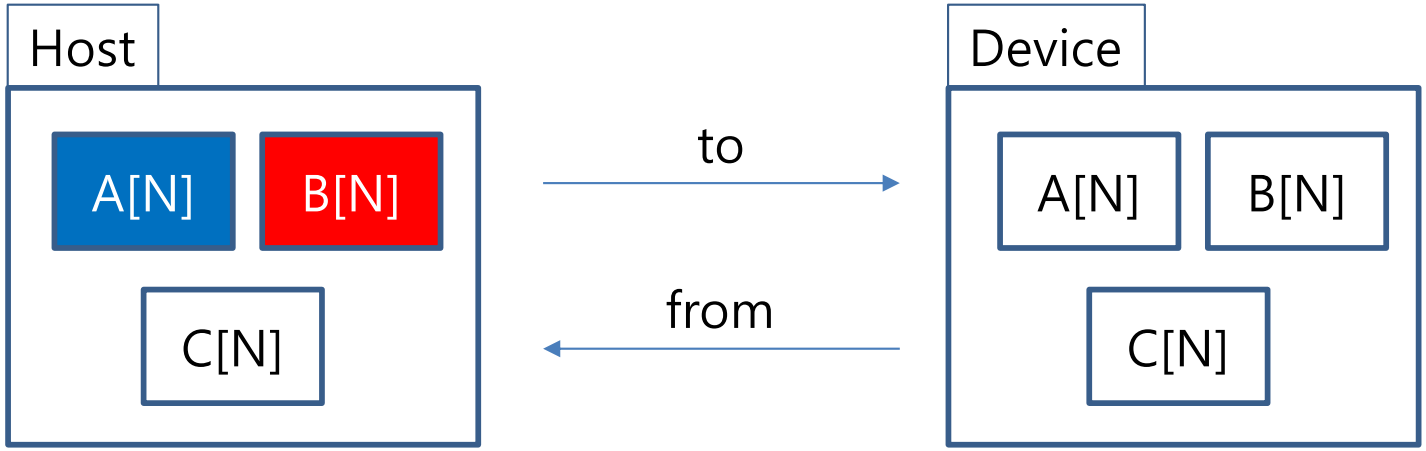




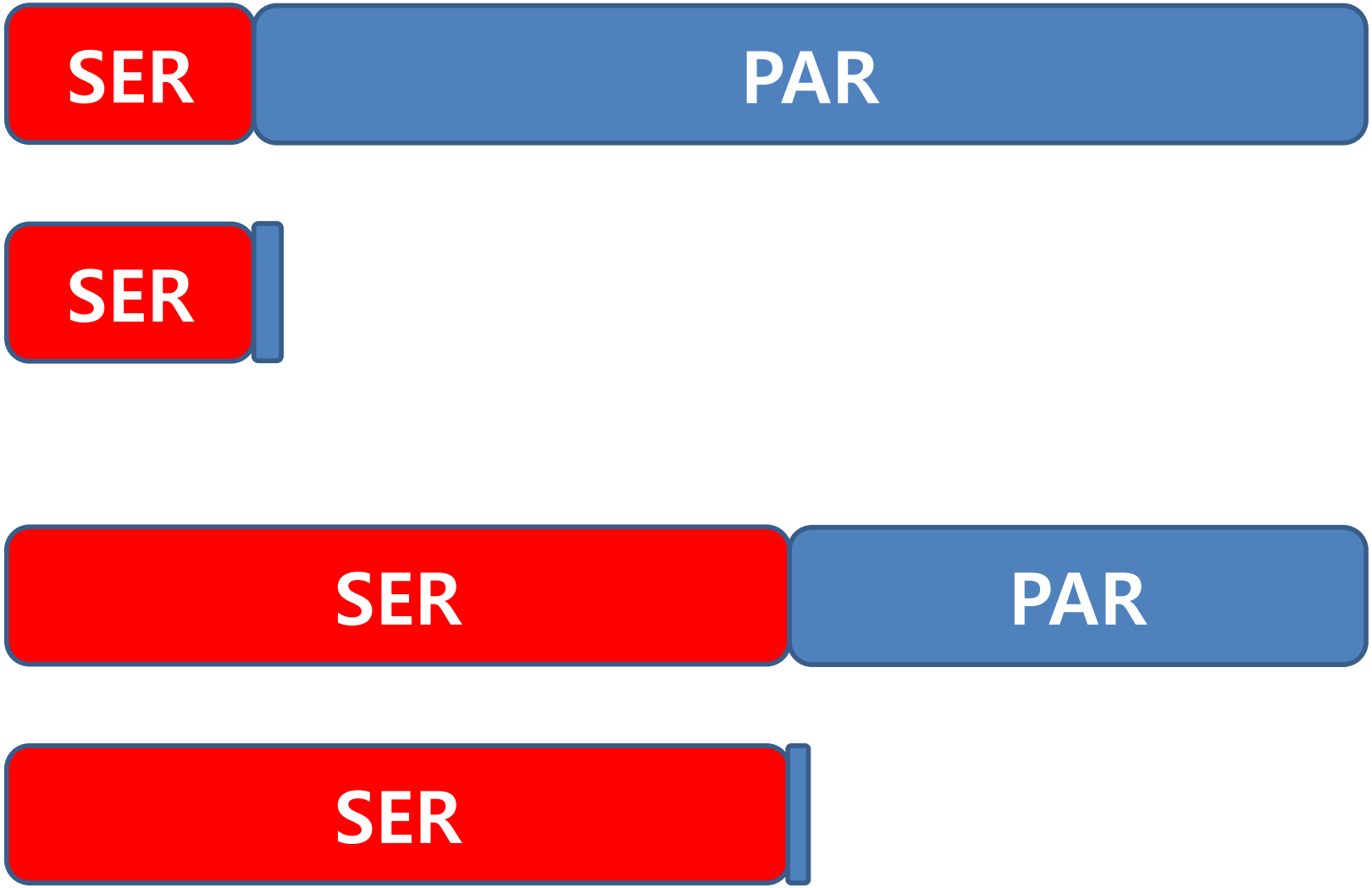
target Directive

- accelerator has dedicated memory
- data should be transferred between host and device
- **map** clause

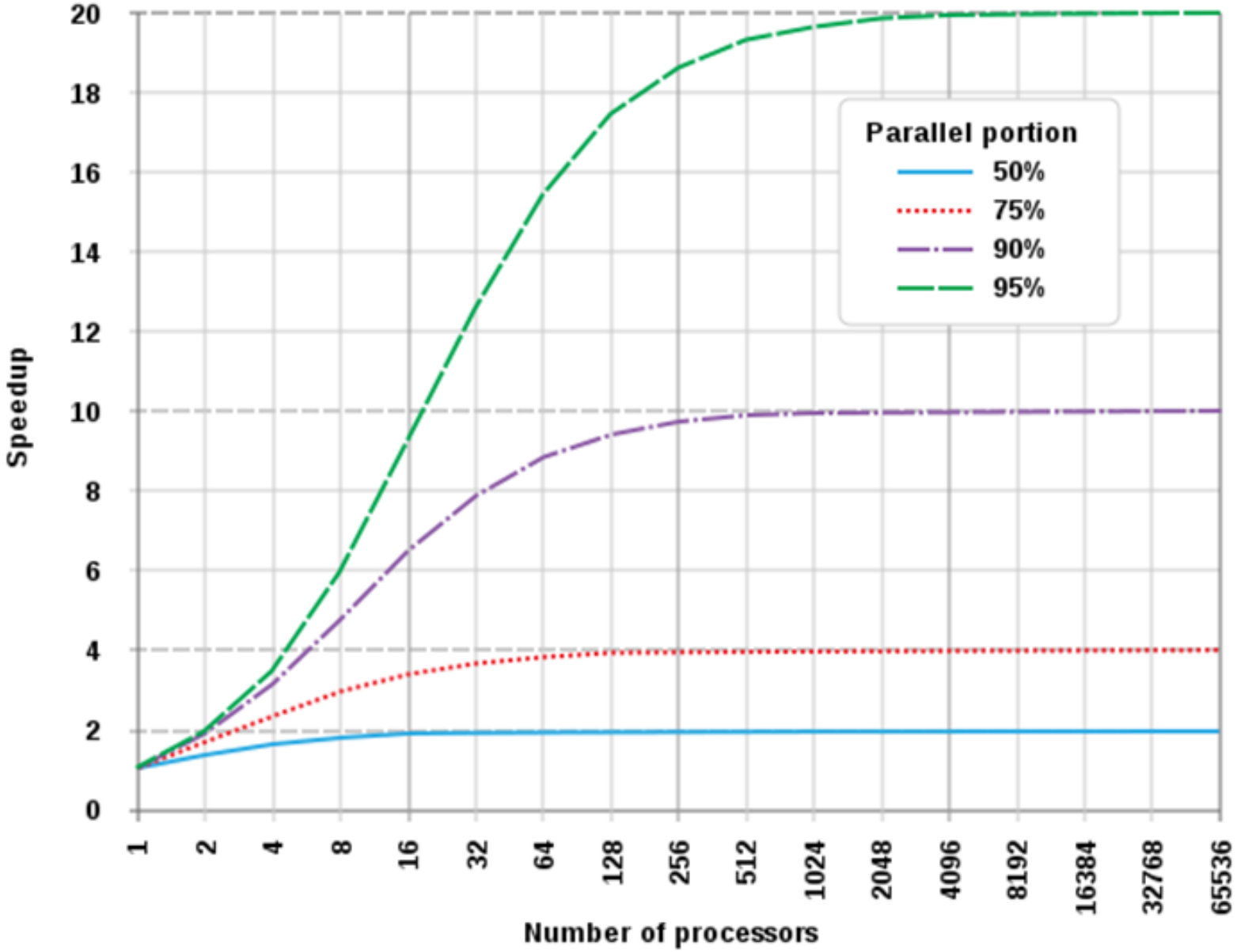
```
  double A[N], B[N], C[N];

#pragma omp target map(to: A[0:N], B[0:N]) map(from: C[0:N])
#pragma omp parallel for
  for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
```



Parallel Efficiency



Amdahl's Law


Speedup
20
18
16
14
12
10
8
6
4
2
0
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of processors
Parallel portion
50%
75%
90%
95%

Thanks for listening!

Any questions?

レポート課題

- 以下のように宣言された行列Aとxの積yを計算するプログラムを作成すること（データの任意の値で初期化）
- 逐次コードの変更はせず、OpenMP指示文の挿入だけで並列化を行うこと

```
#define M 4096
#define N 4096

double A[M][N], x[N], y[M];
```

- 複数のコア（最低8コア）を持つ計算機環境で性能測定を行い、1,2,4,8スレッドでの性能（実行時間）をグラフで示すこと