OpenMP

Parallel Programming for Multicore processors

M. Sato
CCS, University of Tsukuba

Contents

- Why multicore? ～ Trends of Microprocessors
- How to use multicore
  - POSIX Thread
- OpenMP
  - Programming models
- Advanced Topics
  - Hybrid Programming for Muticore clusters
  - OpenMP 3.0 (task)
  - OpenMP 4.0 (Accelerator extension)

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

Pipeline
Superscalar

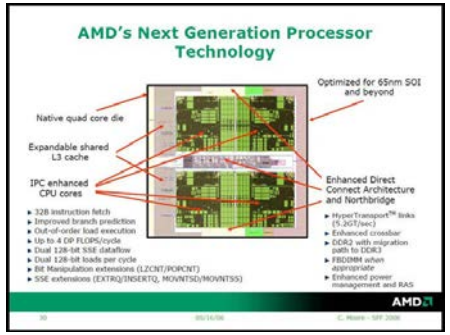
mulitcore

Shared memory multiprocessor

Distributed memory computer or Grid

Trends of Mulitcore processors

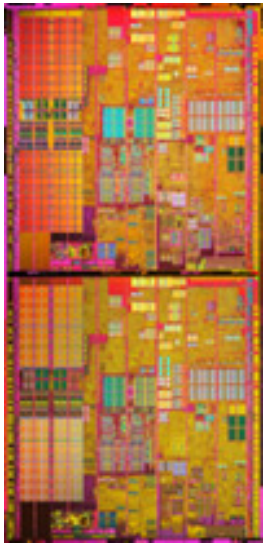
- Faster clock speed, and Finer silicon technology
  - "now clock freq is 3GHz, in future it will reach to 10GHz!?"
    - Intel changed their strategy -> multicore!
    - Clock never become faster any more
  - Silicon technology 45 nm -> 22 nm in near future!



Good news & bad news!

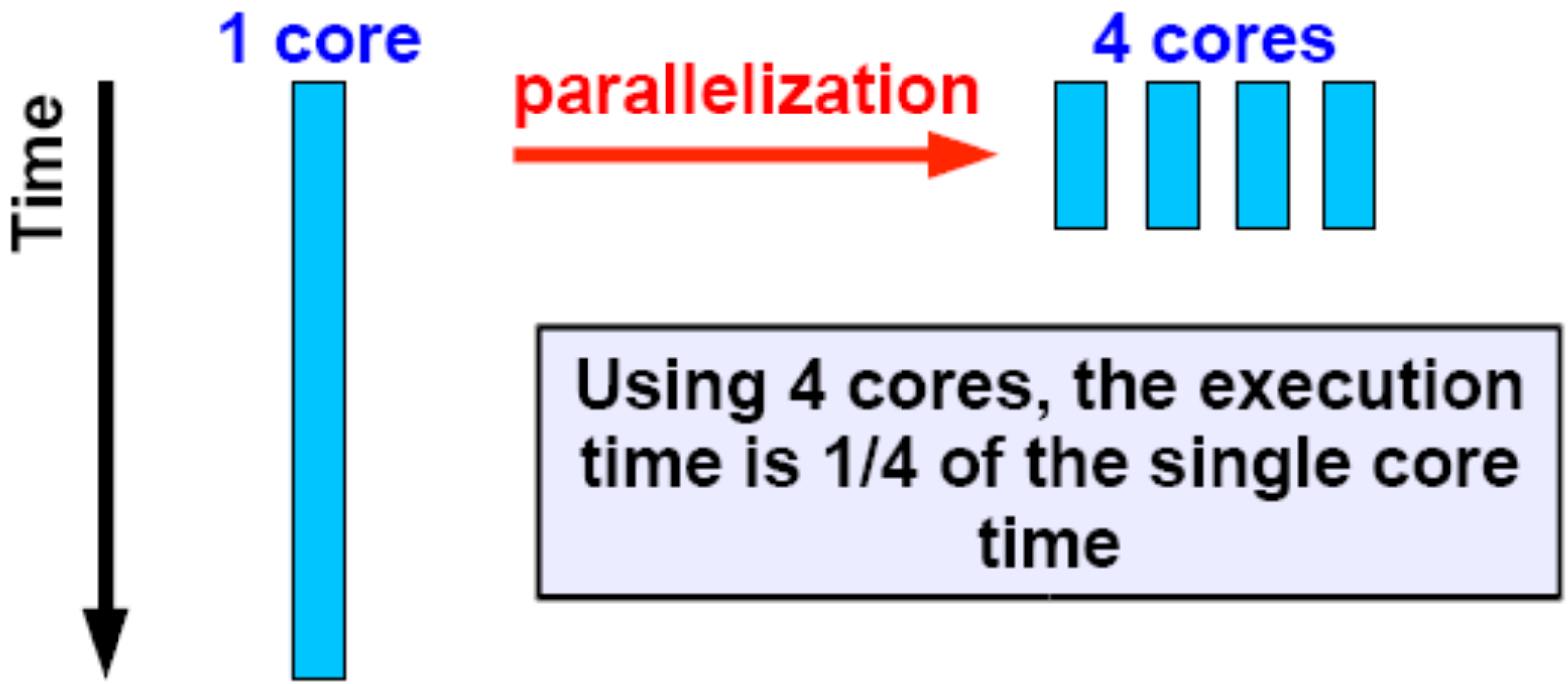
- Progress in Computer Architecture
  - Superpipeline, super scalar, VLIW …
  - Multi-level cache, L3 cache even in microprocessor
  - Multi-thread architecure、Intel Hyperthreading
    - Shared by multiple threads
  - Multi-core：multiple CPU core on one chip dai

Programming support is required

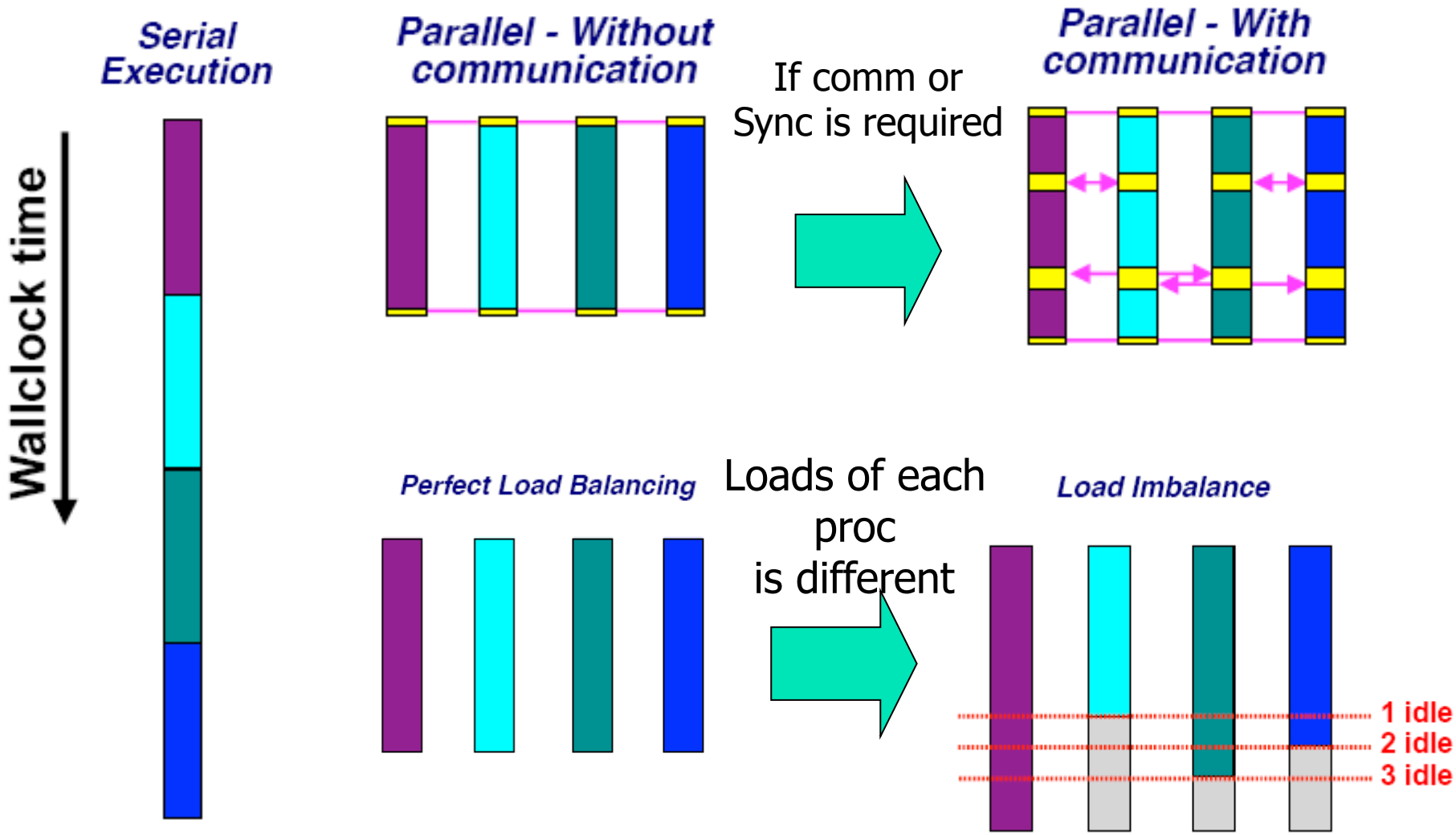


Inetl ® Pentium® processor
Dai of Extreme-edition

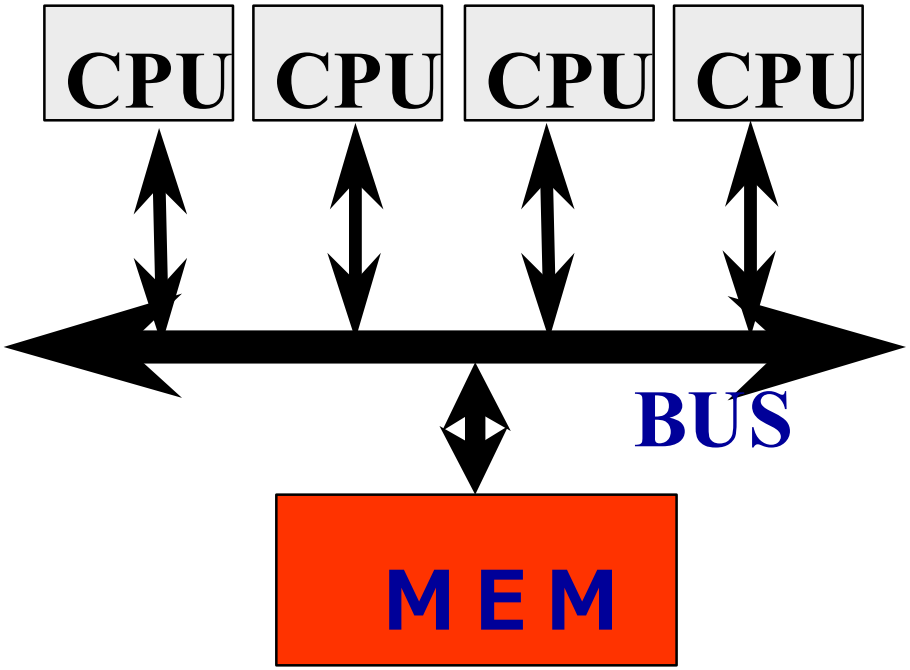
Why parallelization needs?
4 times speedup by using 4 cores!



Overhead of parallel execution

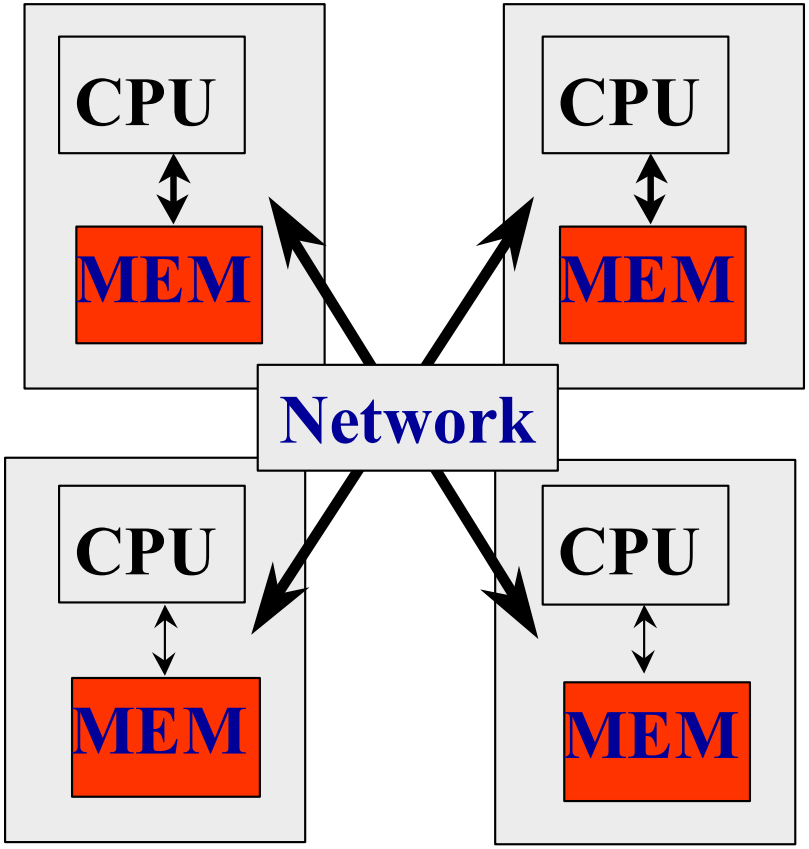


Shared memory multi-processor system



- **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

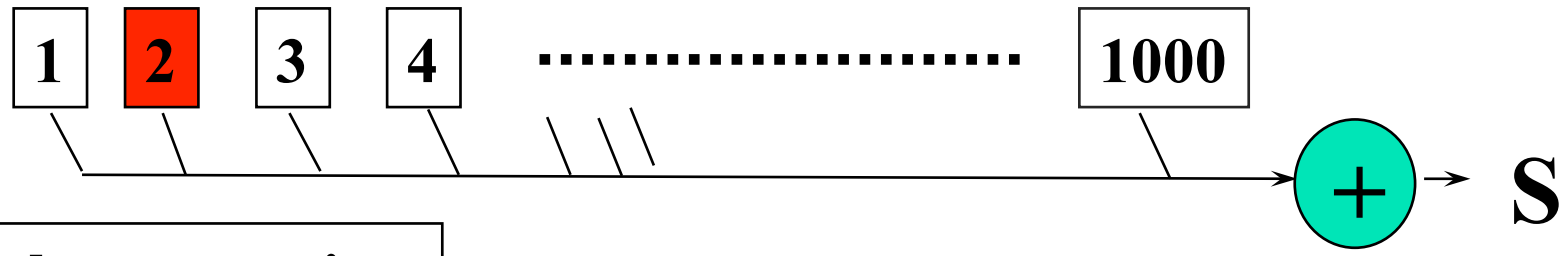


- ◆System with several computer of CPU and memory, connected by network.
- ◆Thread executed in each computer communicate with each other by exchanging data (message) via network.タ
- ◆PC Cluster

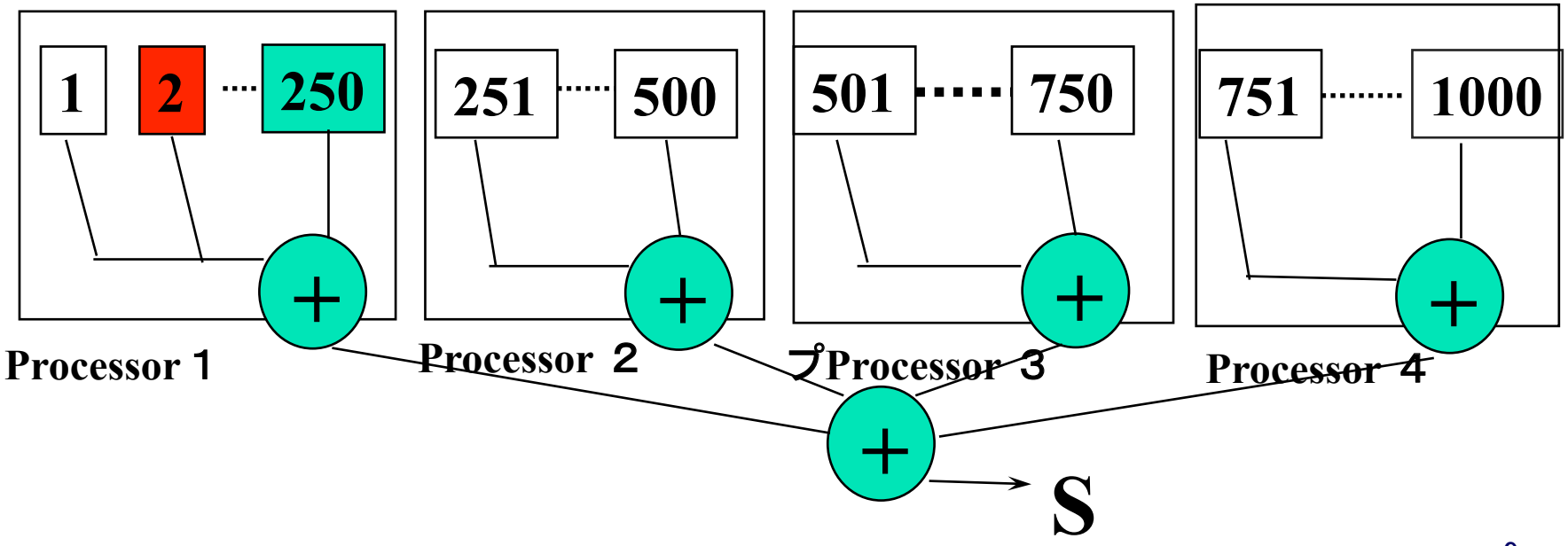
Very simple example of parallel computing

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

Sequential computation



Parallel computation



Parallel programming model

- Message passing programming model
  - Parallel programming by exchange data (message) between processors (nodes)
  - Mainly for distributed memory system (possible also for shared memory)
  - Program must control the data transfer explicitly.
  - Programming is sometimes difficult and time-consuming
  - Program may be scalable (when increasing number of Proc)

- Shared memory programming model
  - Parallel programming by accessing shared data in memory.
  - Mainly for shared memory system. (can be supported by software distributed shared memory)
  - System moves shared data between nodes (by sharing)
  - Easy to program, based on sequential version
  - Scalability is limited. Medium scale multiprocessors.

Parallel programming models

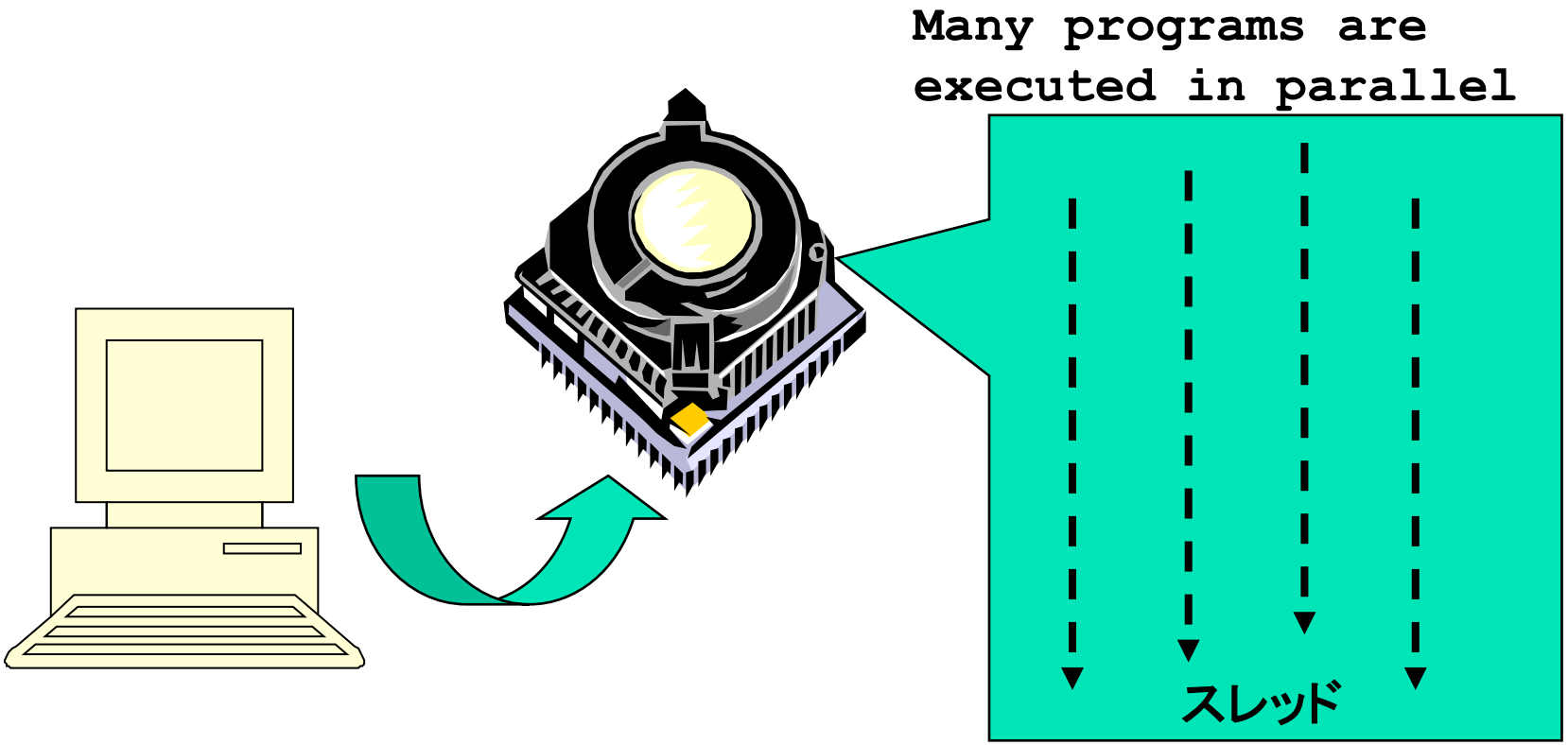
- There are numerous parallel programming models
- The ones most well-known are:
  - Distributed Memory
    - ✓ Sockets (standardized, low level)
    - ✓ PVM - Parallel Virtual Machine (obsolete)
    - ✓ MPI - Message Passing Interface (de-facto std)
  - Shared Memory
    - ✓ Posix Threads (standardized,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

Any Cluster and/or single SMP

SMP only

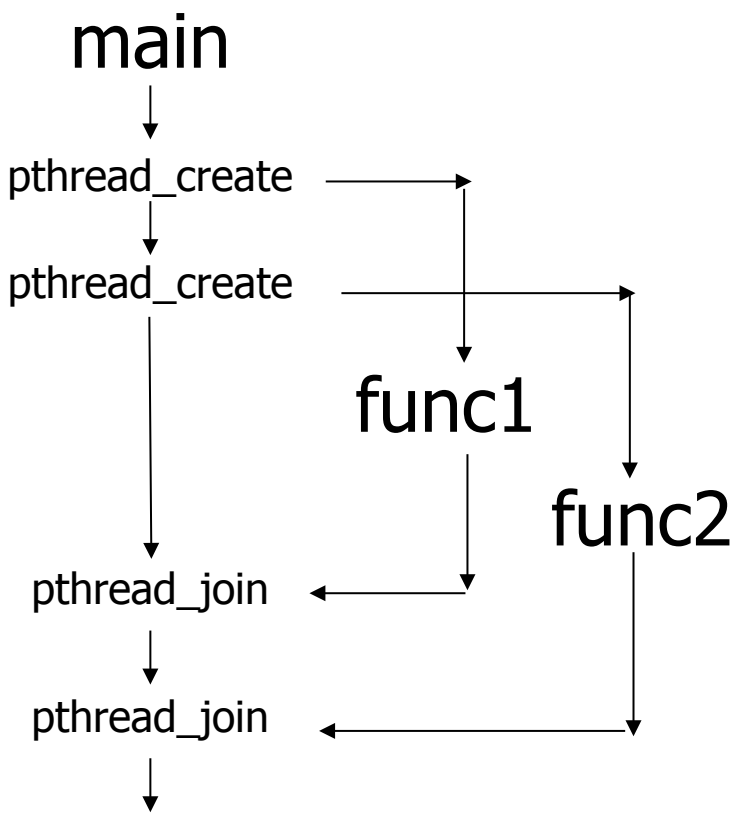
Multithread(ed) programming

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



POSIX thread library

- Create thread: thread_create
- Join threads: pthread_join
- Synchronization, lock




```
#include <pthread.h>

void func1( int x );   void func2( int x );

main() {
        pthread_t t1 ;
        pthread_t t2 ;
           pthread_create( &t1, NULL,
                           (void *)func1, (void *)1 );
           pthread_create( &t2, NULL,
                           (void *)func2, (void *)2 );
           printf("main()\n");
           pthread_join( t1, NULL );
           pthread_join( t2, NULL );
}
void func1( int x ) {
      int i ;
       for( i = 0 ; i<3 ; i++ ) {
             printf("func1( %d ): %d \n",x, i );
         }
}
void func2( int x ) {
             printf("func2( %d ): %d \n",x);
}
```

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;
}
```

Simple example of Message Passing Programming

- Sum up 1000 element in array

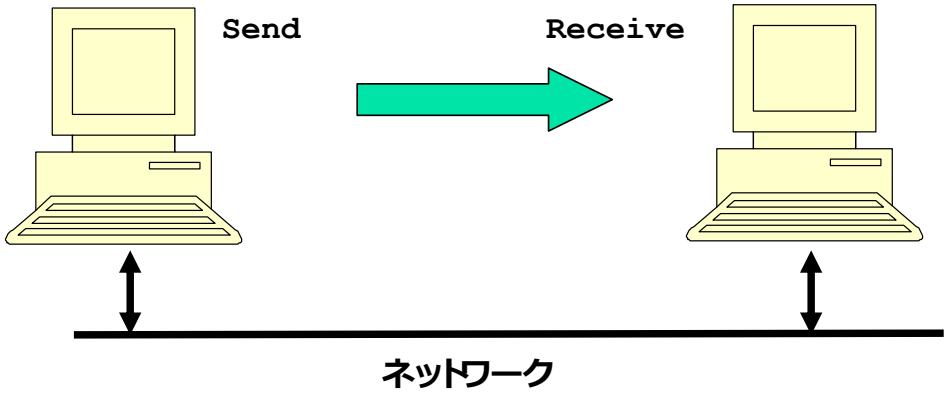
```
int a[250]; /* 250 elements are allocated in each node */

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

Over-specs for Embedded system Programming?!

- Communication with message
  - Send/Receive
- Collective operations
  - Reduce/Bcast
  - Gather/Scatter



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,&statu
            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;
}
```

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/

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];
```

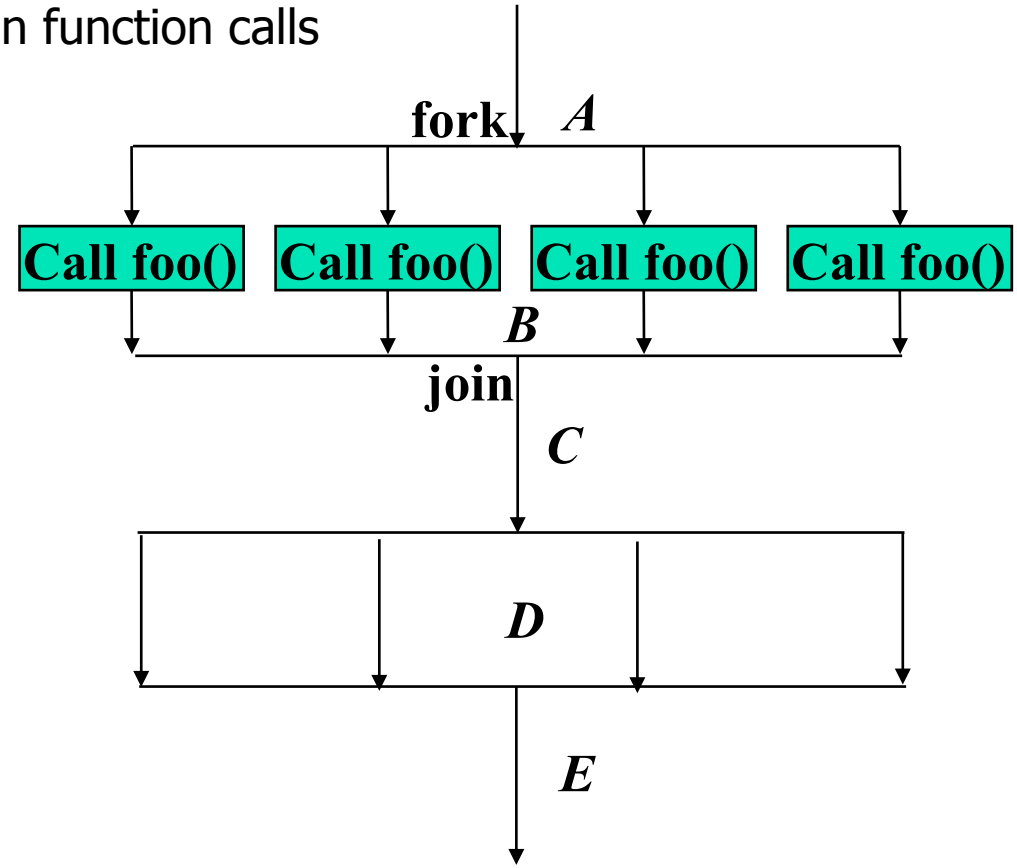
OpenMP API

- 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 …
```



Parallel Region

- A code region executed in parallel by multiple threads (team)
  - Specified by Parallel constructs
  - A set of threads executing the same parallel region is called "team"
  - Threads in team execute the same code in region (duplicated execution)

```
#pragma omp parallel
{
    …
    … Parallel region…
    …
}
```

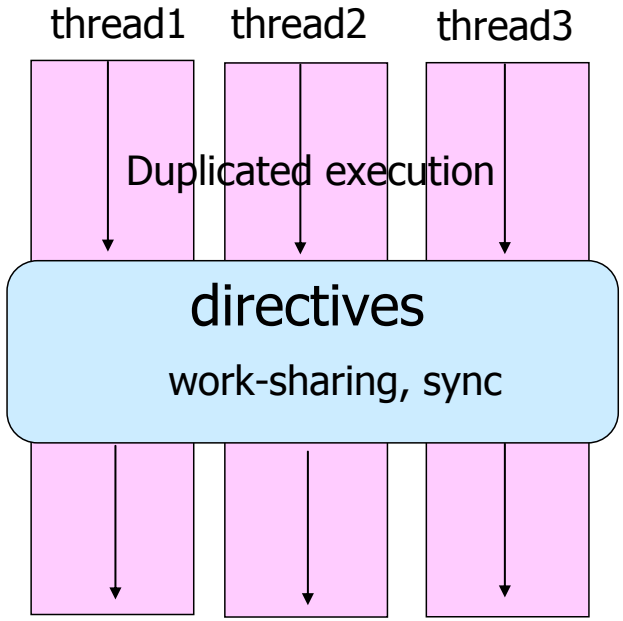
Demo

- Get CPU information by looking at /proc/cpuinfo
- gcc –fopenmp, gcc support OpenMP from 4.2, gfortran
- Control #proessors by OMP_NUM_THREADS

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

main()
{
   printf("omp-test ... n_thread=%d\n",omp_get_max_threads());
#pragma omp parallel
   {
         printf("thread (%d/%d)...\n",
               omp_get_thread_num(),omp_get_num_threads());
   }
   printf("end...\n");
}
```

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

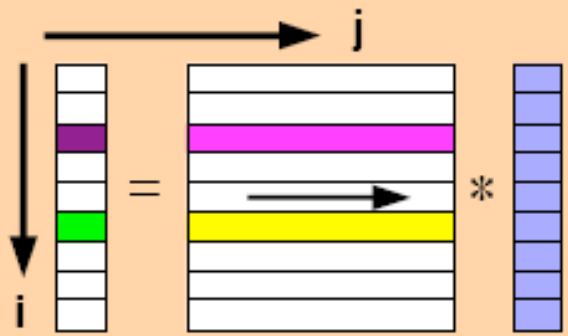
- Execute iterations specified For-loop in parallel
- For-loop specified by the directive must be in *canonical shape*

```
#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

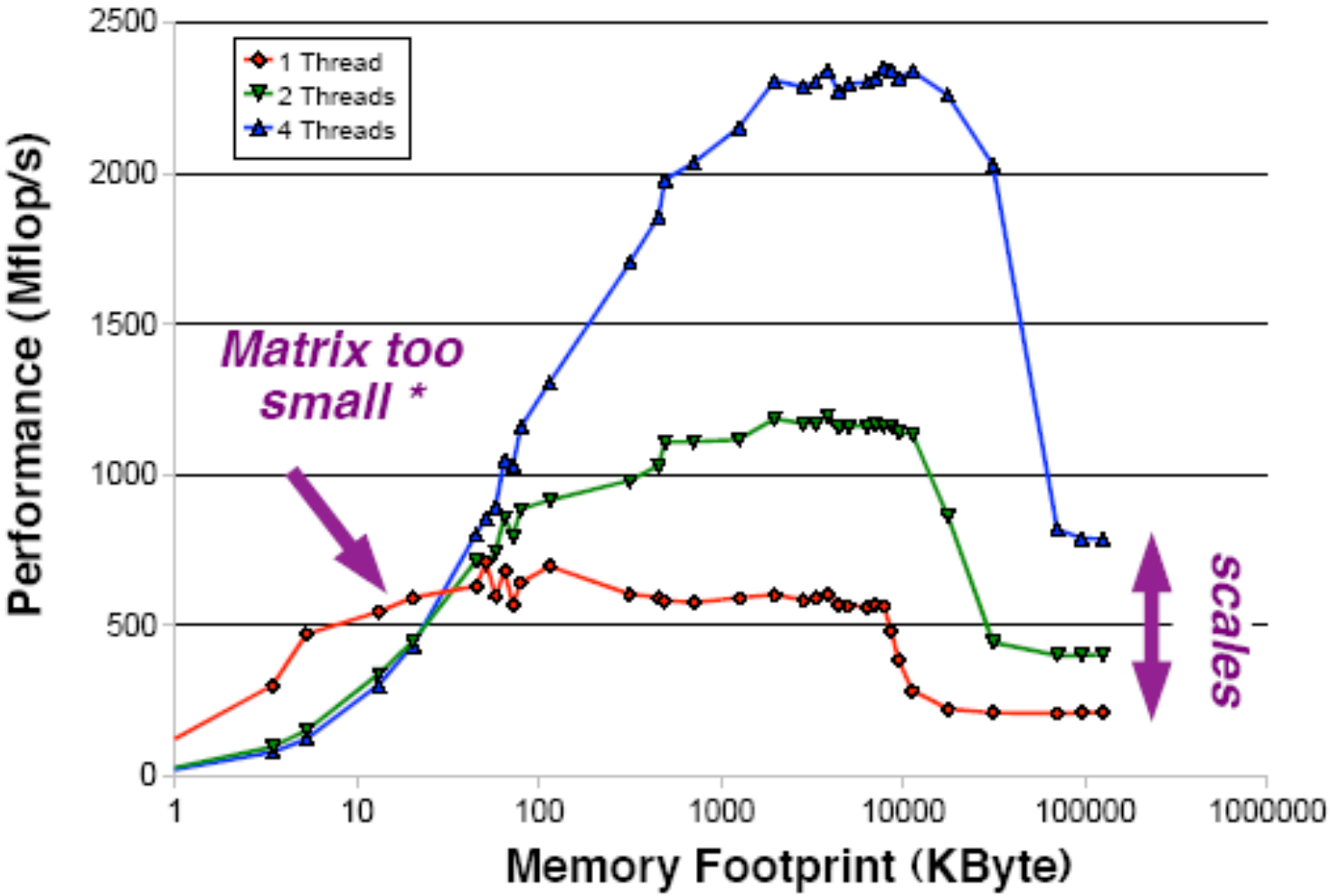
```
for (i=0,1,2,3,4)
 i = 0
   sum = Σ b[i=0][j]*c[j]
   a[0] = sum
 i = 1
   sum = Σ b[i=1][j]*c[j]
   a[1] = sum
```

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 ...

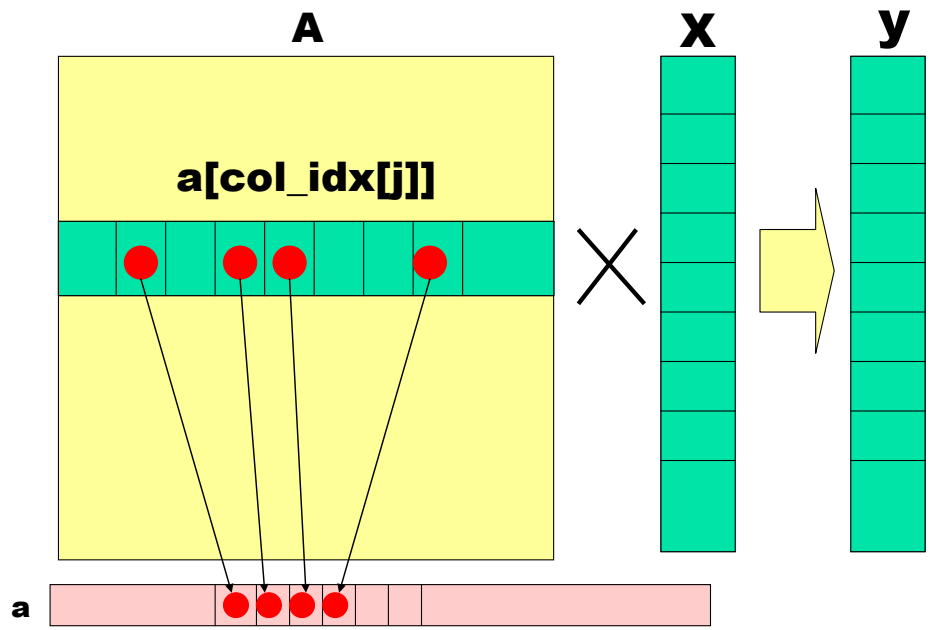
The performance looks like …



Example code

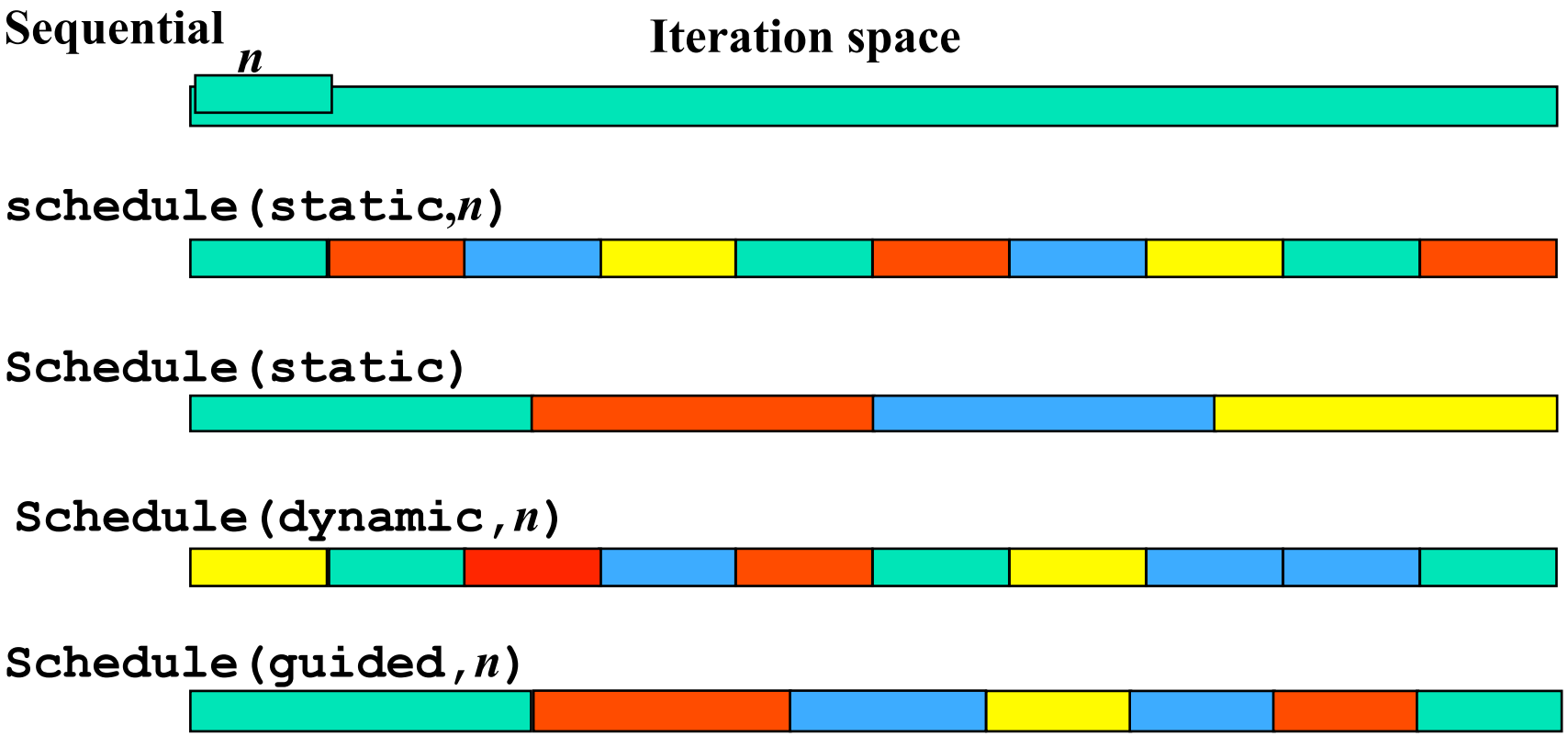
Sparse matrix vector product

```
Matvec(double a[],int row_start,int col_idx[],
 double x[],double y[],int n)
{
   int i,j,start,end; double t;
#pragma omp parallel for private(j,t,start,end)
   for(i=0; i<n;i++){
      start=row_start[i];
      end=row_start[i+1];
      t = 0.0;
      for(j=start;j<end;j++)
         t += a[j]*x[col_idx[j]];
      y[i]=t;
  }
}
```



Scheduling methods of parallel loop

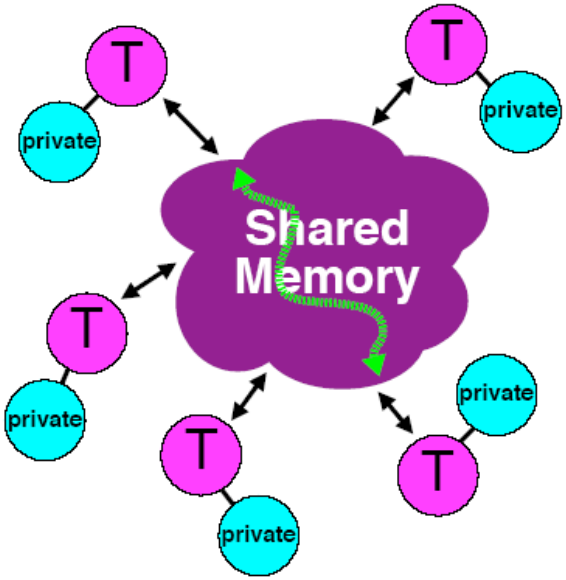
- #processor = 4



Data scope attribute clause

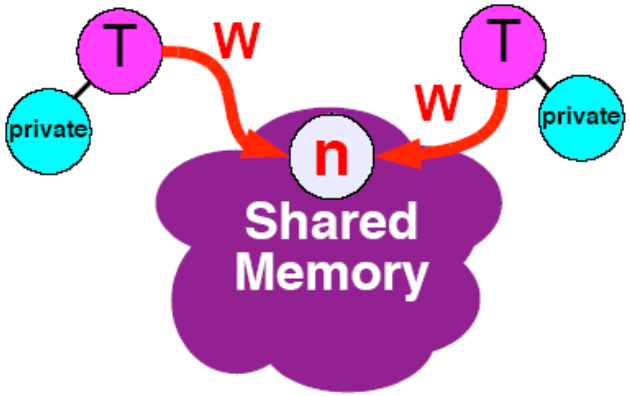
- Clause specified with `parallel`consruct、work sharing construct
- `shared(var_list)`
  - Specified variables are shared among threads.
- `private(var_list)`
  - Specified variables replicated as a private variable
- `firstprivate(var_list)`
  - Same as private, but initialized by value before loop.
- `lastprivate(var_list)`
  - Same as private, but the value after loop is updated by the value of the last iteration.
- `reduction(op:var_list)`
  - Specify the value of variables computed by reduction operation op.
  - Private during execution of loop, and updated at the end of loop

Data Race

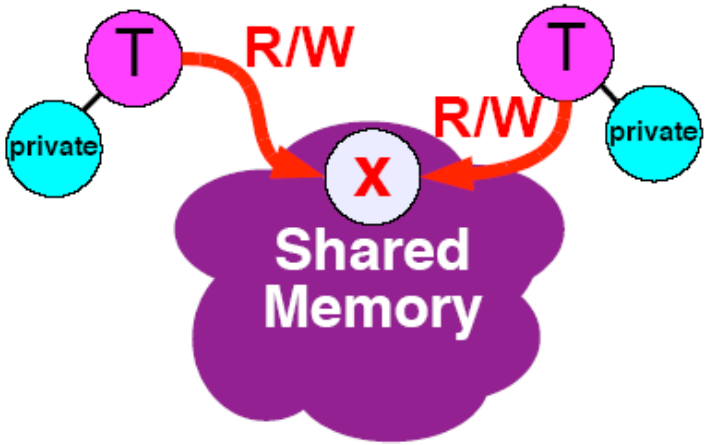


OpenMP
Is shared
Memory!

```
#pragma omp parallel shared(n)
{n = omp_get_thread_num();}
```




```
#pragma omp parallel shared(x)
{x = x + 1;}
```



Data Race =
Write a same variable by
different threads

You cannot parallelize this loop

```
for (i=0; i<8; i++)
   a[i] = a[i] + b[i];
```

Every iteration in this loop is independent of the other iterations

Thread 1	Thread 2
a[0]=a[0]+b[0]	a[4]=a[4]+b[4]
a[1]=a[1]+b[1]	a[5]=a[5]+b[5]
a[2]=a[2]+b[2]	a[6]=a[6]+b[6]
a[3]=a[3]+b[3]	a[7]=a[7]+b[7]

Time

```
for (i=0; i<8; i++)
   a[i] = a[i+1] + b[i];
```

The result is not deterministic when run in parallel !

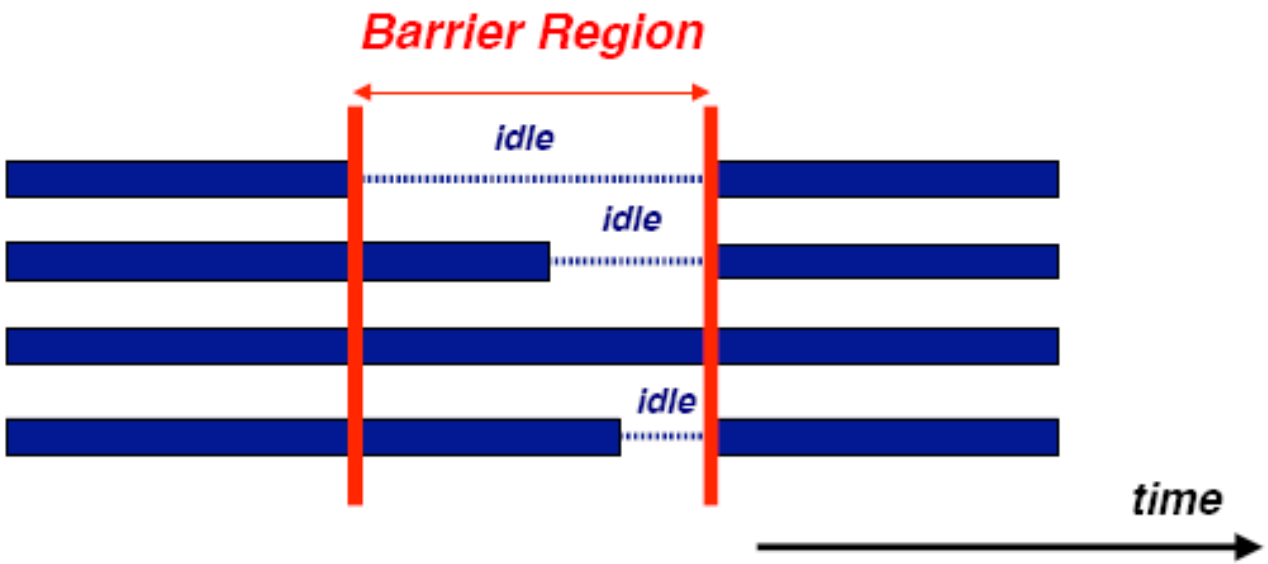
Thread 1	Thread 2
a[0]=a[1]+b[0]	a[4]=a[5]+b[4]
a[1]=a[2]+b[1]	a[5]=a[6]+b[5]
a[2]=a[3]+b[2]	a[6]=a[7]+b[6]
a[3]=a[4]+b[3]	a[7]=a[8]+b[7]

Time

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
```



Barrier is important in this case


```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

wait !

barrier

```
for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
```

You don't need to put barrier directive
Because for directive without nowait performs implicit barrier.

How to use nowait

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

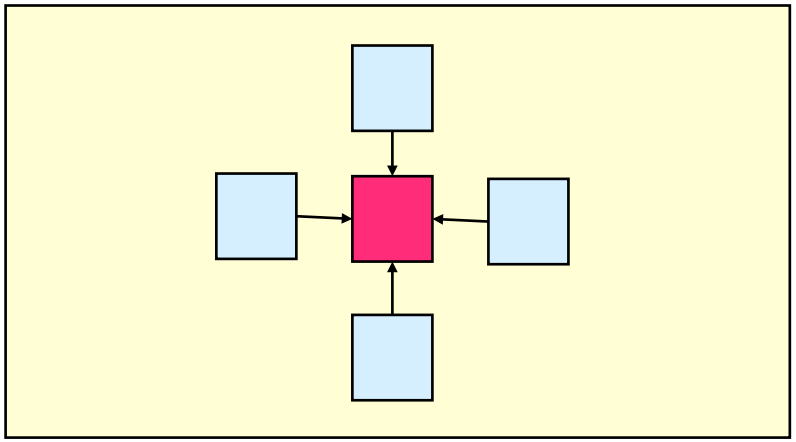
(implied barrier)

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

Example of OpenMP program：laplace

- 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: lap.c
  - 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);
}
```

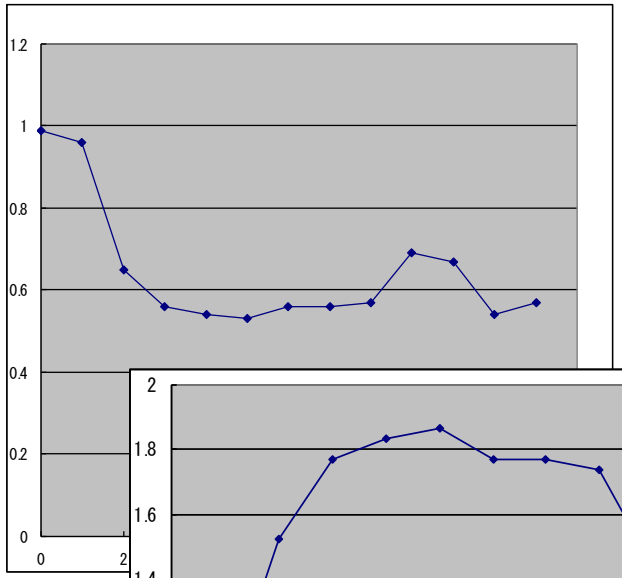
What about performance?

- OpenMP really speedup my problem?!
- It depends on hardware and problem size/characteristics
- Esp. problem sizes is an very important factor
  - Trade off between overhead of parallelization and grain size of parallel execution.
- To understand performance, …
  - How to lock
  - How to exploit cache
  - Memory bandwidth

Laplace performance

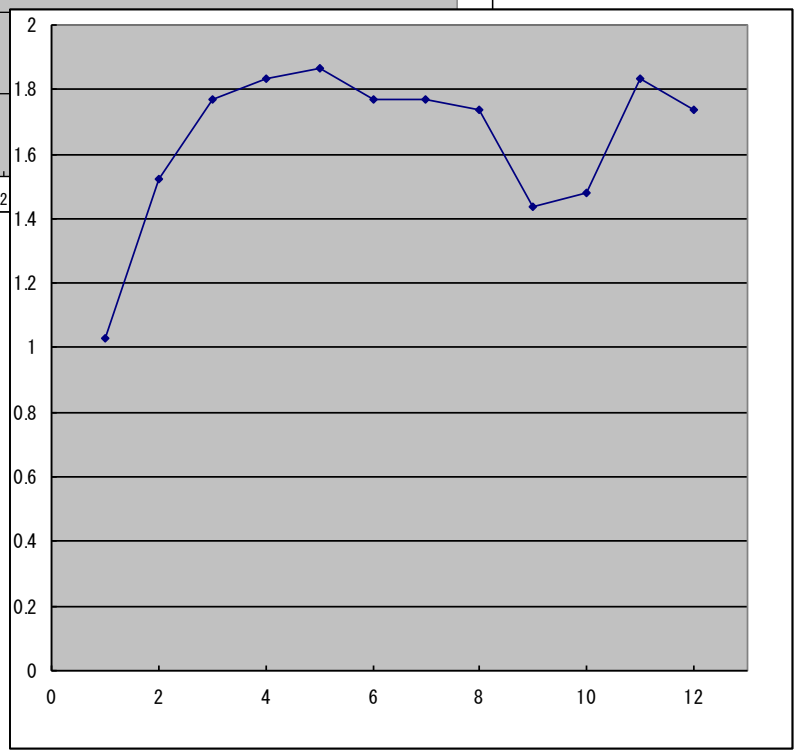
AMD Opteron quad , 2 socket

XSIZE=YSIZE=1000

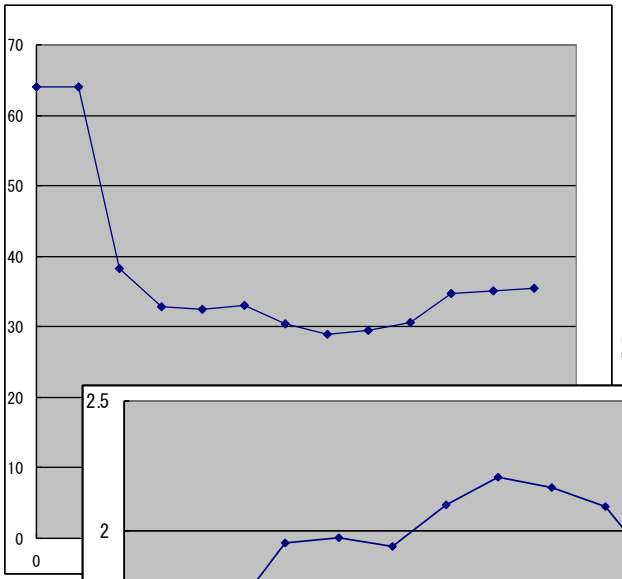


Exec time

speedup

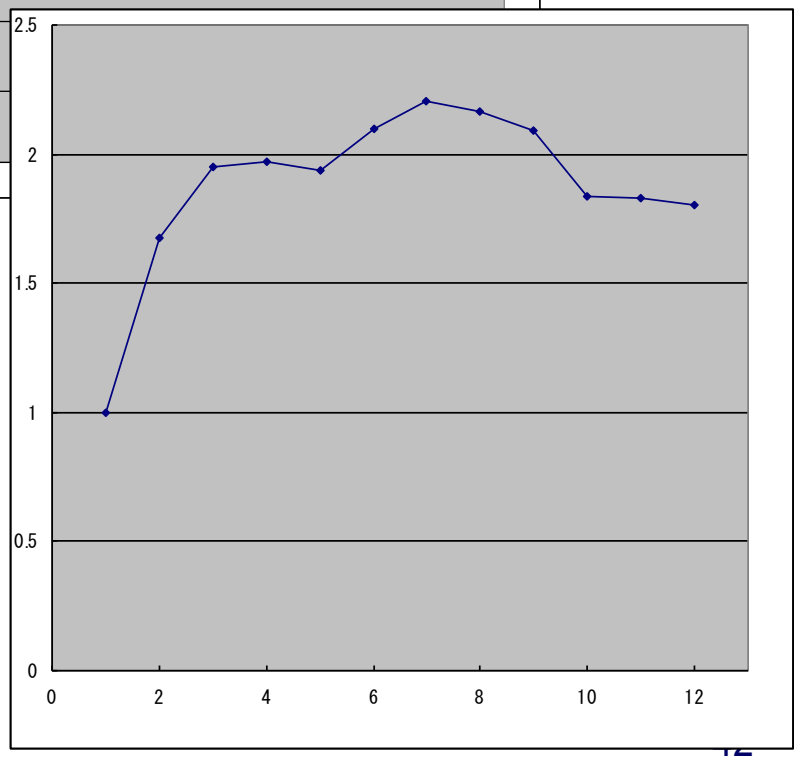


XSIZE=YSIZE=8000



Exec time

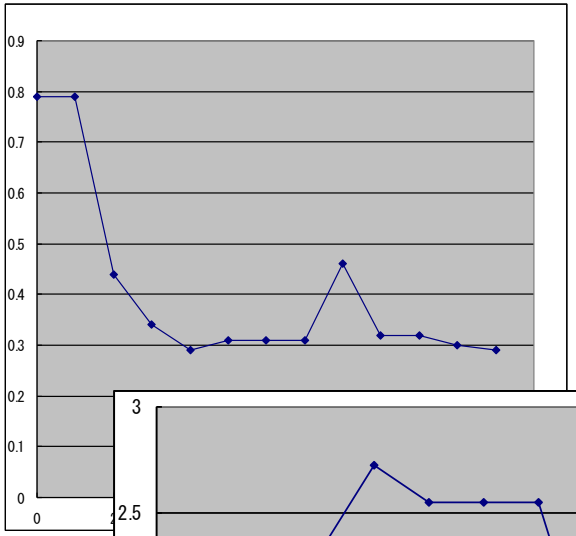
speedup



Laplace performance

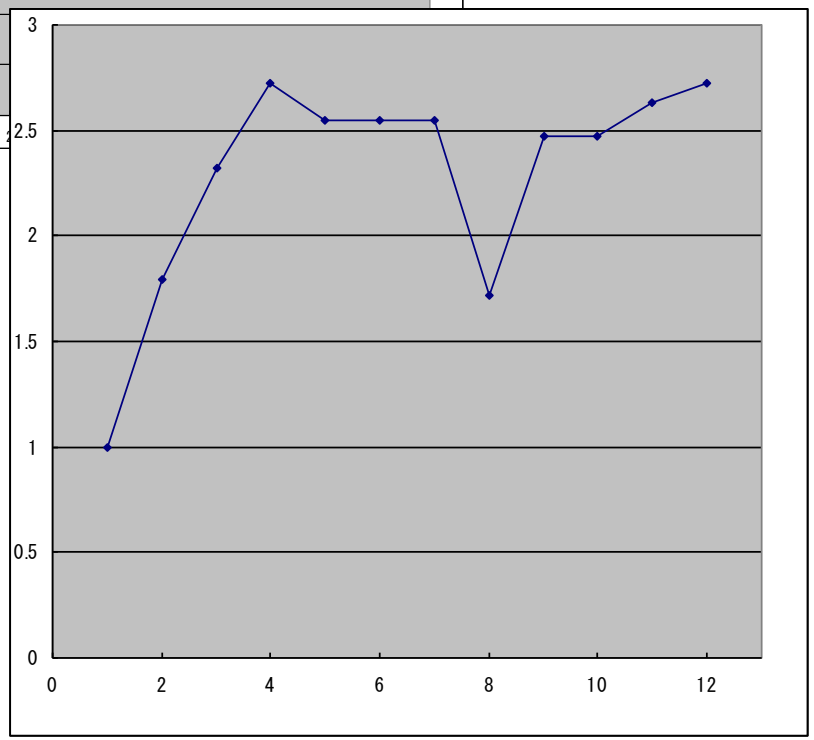
Core i7 920 @ 2.67GHz, 2 socket

XSIZE=YSIZE=1000

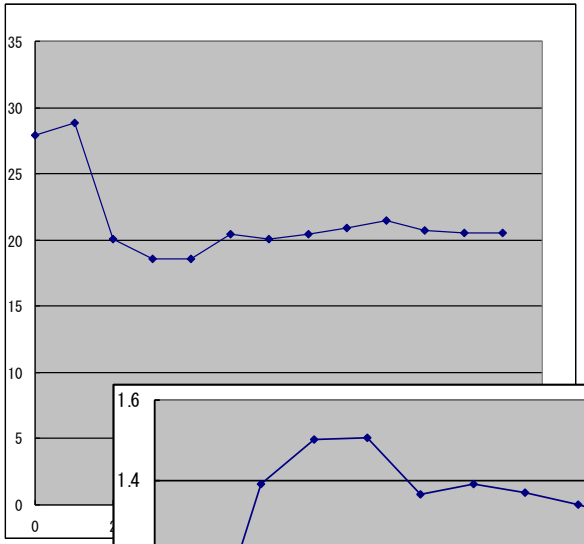


Exec time

speedup

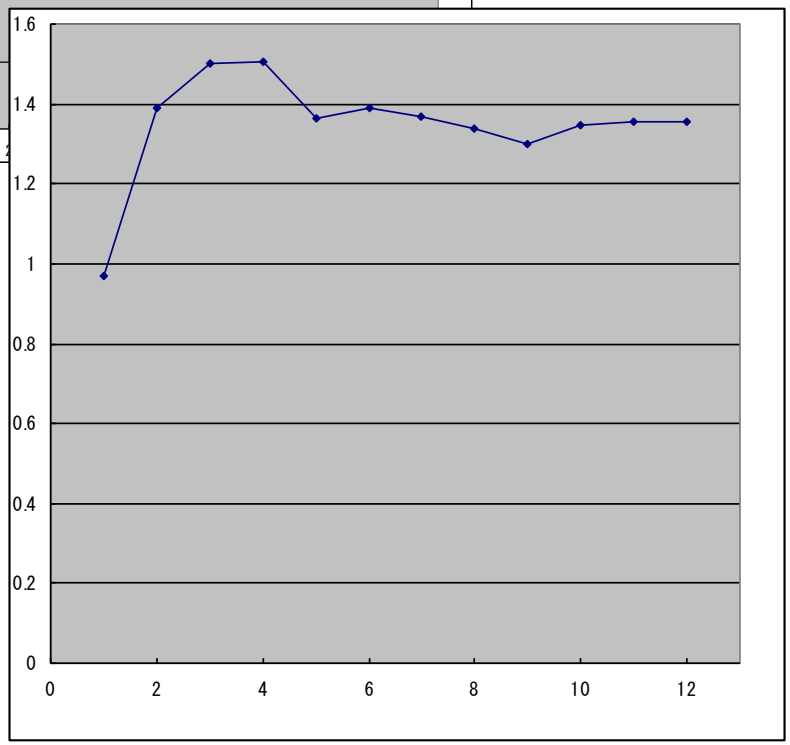


XSIZE=YSIZE=8000



Exec time

speedup

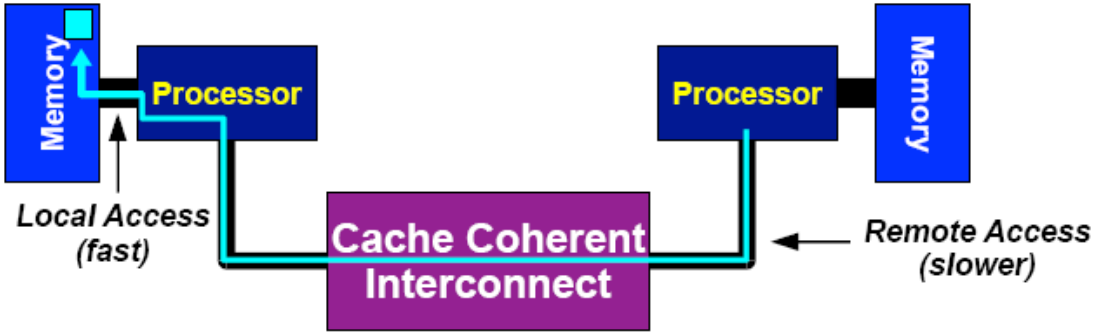


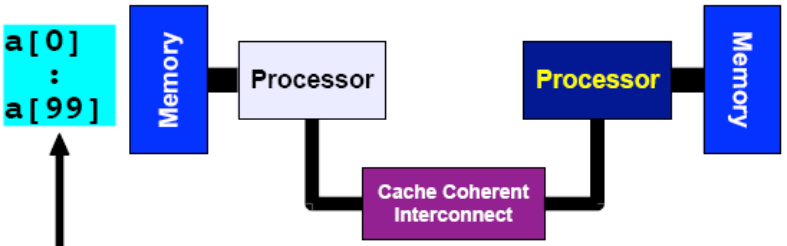
The Myth

"OpenMP Does Not Scale"

- The transparency of OpenMP is a mixed blessing
  - Makes things pretty easy
  - May mask performance bottlenecks
- In the ideal world, an OpenMP application just performs well
- Unfortunately, this is not the case
- Two of the more obscure effects that can negatively impact performance are cc-NUMA behavior and False Sharing
- Neither of these are restricted to OpenMP, but they are important enough to cover in some detail here

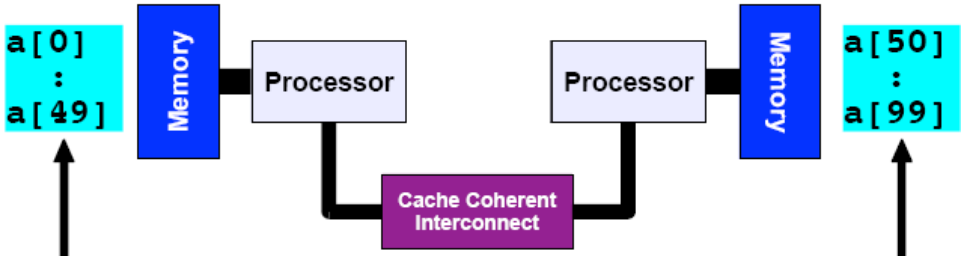
CC-NUMA and first touch






```
for (i=0; i<100; i++)
    a[i] = 0;
```

First Touch
All array elements are in the memory of the processor executing this thread

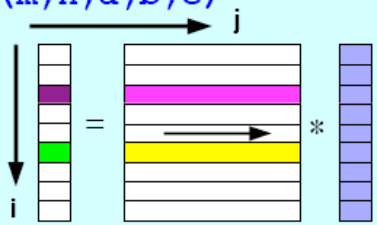


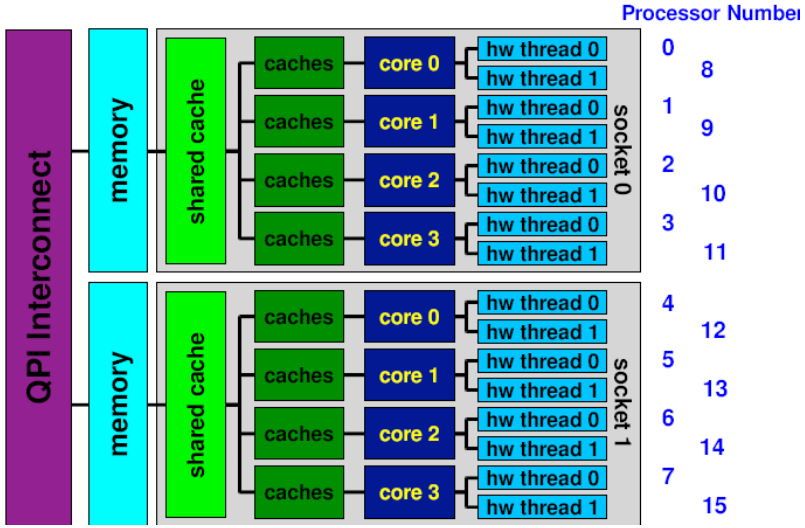

```
#pragma omp parallel for num_threads(2)
for (i=0; i<100; i++)
    a[i] = 0;
```

First Touch
Both memories each have "their half" of the array

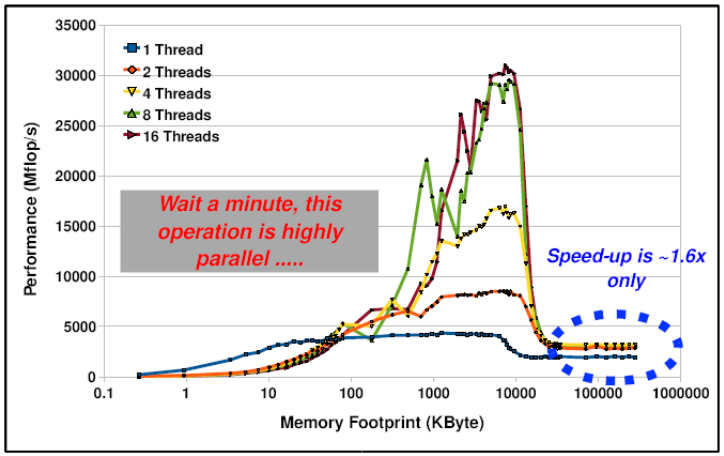
First touch

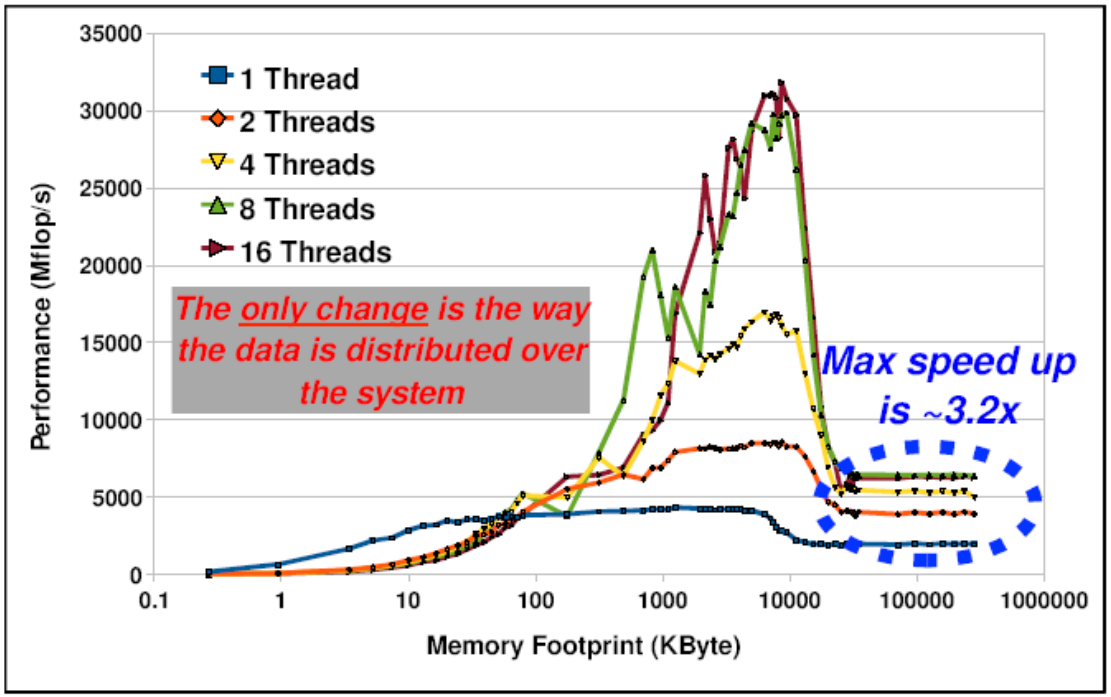
```
#pragma omp parallel for default(none) \
        private(i,j) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
  a[i] = 0.0;
  for (j=0; j<n; j++)
     a[i] += b[i][j]*c[j];
 }
```





2 socket Nehalem



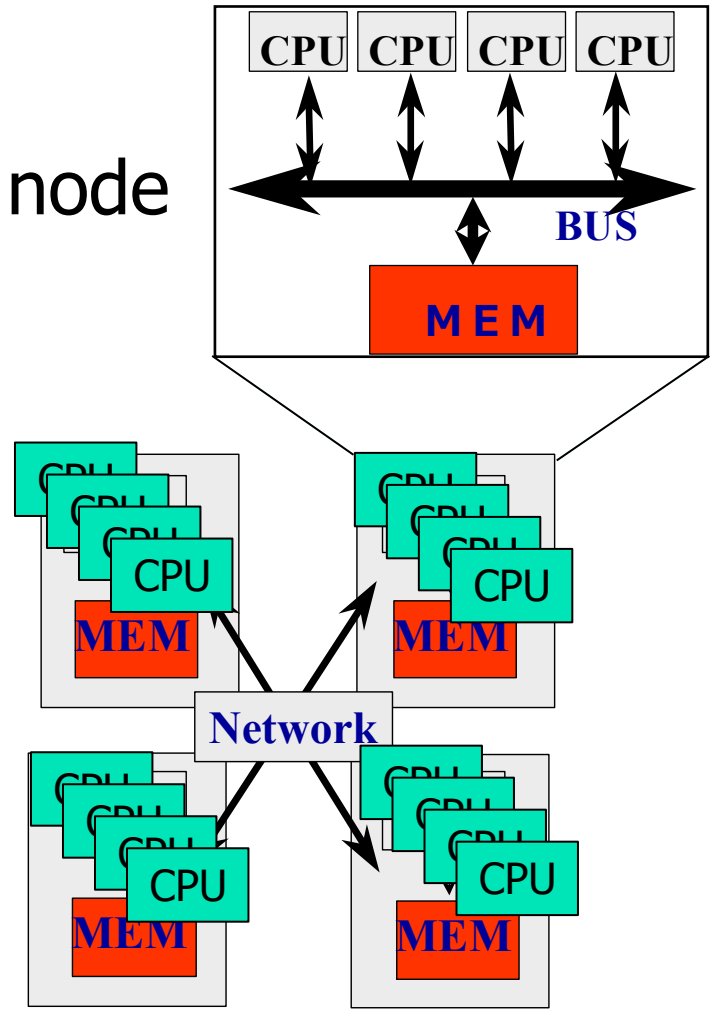


Advanced topics

- MPI/OpenMP Hybrid Programming
  - Programming for SMP (multicore) cluster
- OpenMP 3.0
  - Approved in 2007
  - Task
- OpenMP 4.0
  - Approved in 2013
  - Accelerator device extension

MPI-OpenMP hybrid programming How to use multi-core cluster

- Flat MPI: Run MPI process in core (CPU)
  - Many MPI processes
  - Only MPI programming is needed
- MPI-OpenMP hybrid
  - Use MPI between nodes
  - Use OpenMP in node
  - Save number of MPI process, resulting in saving memory. Important in large-scale system
  - Cost: Need two (MPI-OpenMP) programming
  - Sometimes OpenMP performance is worse than MPI



Thread-safety of MPI

- Use MPI_ MPI_Init_thread to get info about thread-safety
- MPI_THREAD_SINGLE
  - A process has only one thread of execution.
- MPI_THREAD_FUNNELED
  - A process may be multithreaded, but only the thread that initialized MPI can make MPI calls.
- MPI_THREAD_SERIALIZED
  - A process may be multithreaded, but only one thread at a time can make MPI calls.
- MPI_THREAD_MULTIPLE
  - A process may be multithreaded and multiple threads can call MPI functions simultaneously.

Update in OpenMP3.0

- The concept of "task" is introduced:
  - An entity of thread created by Parallel construct and Task construct.
  - Task Construct & Taskwait construct
- Interpretation of shared memory consistency in OpenMP
  - Definition of Flush semantics
- Nested loop
  - Collapse clauses
- Specify stack size of thread.
- constructor, destructor of private variables in C++

Example of Task Constructs


```
struct node {
        struct node *left;
        struct node *right;
};

void postorder_traverse( struct node *p ) {
        if (p->left)
                #pragma omp task // p is firstprivate by default
                postorder_traverse(p->left);
        if (p->right)
                #pragma omp task // p is firstprivate by default
                postorder_traverse(p->right);
        #pragma omp taskwait
        process(p);
}
```

Task Construct

```
long comp_fib_numbers(int n){

   // Basic algorithm: f(n) = f(n-1) + f(n-2)

   long fnm1, fnm2, fn;

   if ( n == 0 || n == 1 ) return(n);

#pragma omp task shared(fnm1)
   {fnm1 = comp_fib_numbers(n-1);}

#pragma omp task shared(fnm2)
   {fnm2 = comp_fib_numbers(n-2);}

#pragma omp taskwait
   fn   = fnm1 + fnm2;

   return(fn);
}
```

```
long comp_fib_numbers(int n){

   // Basic algorithm: f(n) = f(n-1) + f(n-2)

   long fnm1, fnm2, fn;

   if ( n == 0 || n == 1 ) return(n);
   if ( n < 20 ) return(comp_fib_numbers(n-1) +
                        comp_fib_numbers(n-2));

#pragma omp task shared(fnm1)
   {fnm1 = comp_fib_numbers(n-1);}

#pragma omp task shared(fnm2)
   {fnm2 = comp_fib_numbers(n-2);}

#pragma omp taskwait
   fn   = fnm1 + fnm2;

   return(fn);
}
```

Must be in parallel construct

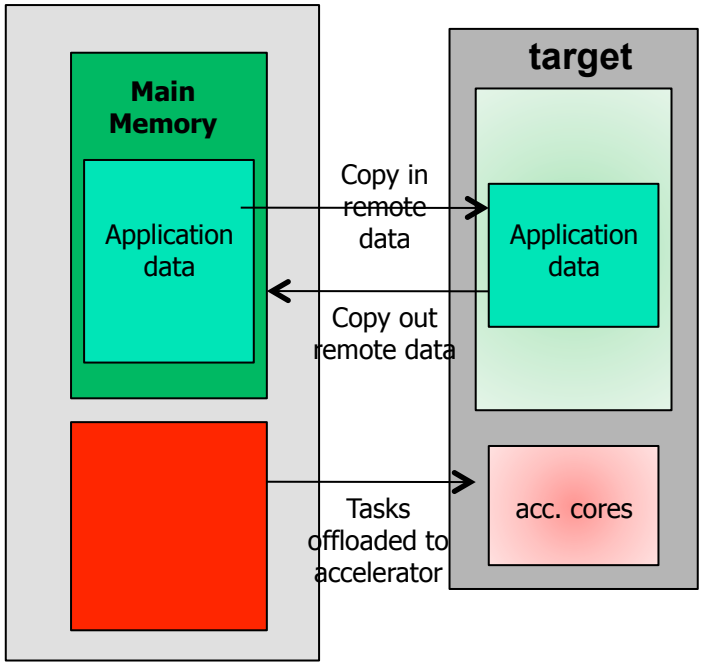
OpenMP 4.0

- Released July 2013
  - http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
  - A document of examples is expected to release soon
- Changes from 3.1 to 4.0 (Appendix E.1):
  - *Accelerator: 2.9*
  - *SIMD extensions: 2.8*
  - *Places and thread affinity: 2.5.2, 4.5*
  - *Taskgroup and dependent tasks: 2.12.5, 2.11*
  - *Error handling: 2.13*
  - *User-defined reductions: 2.15*
  - *Sequentially consistent atomics: 2.12.6*
  - *Fortran 2003 support*

slide by Yonghong@UH

Accelerator (2.9): offloading

- Execution Model: Offload data and code to accelerator
- *target* construct creates tasks to be executed by devices
- Aims to work with wide variety of accs
  - GPGPUs, MIC, DSP, FPGA, etc
  - A target could be even a remote node, intentionally




```
#pragma omp target
{
/* it is like a new task
* executed on a remote device */
{
```


slide by Yonghong@UH

target and map examples


```
void vec_mult(int N)
{
   int i;
   float p[N], v1[N], v2[N];
   init(v1, v2, N);
   #pragma omp target map(to: v1, v2) map(from: p)
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

```
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

slide by Yonghong@UH

Final comments

- Parallelization is a must in multicore!

- OpenMP provide easy way to parallelize from sequential code.
- It is good way up to 64 processors.
- Easy way to use multi-core processor.⇒ now, can be applied to accelerator devices such as GPU and DSP.

- OpenMP is sometime not scalable. MPI is preferable beyond 100 processors.
  - MPI programming is not easy, like OpenMP.
  - Hybrid programming may be required in a large-scale system.