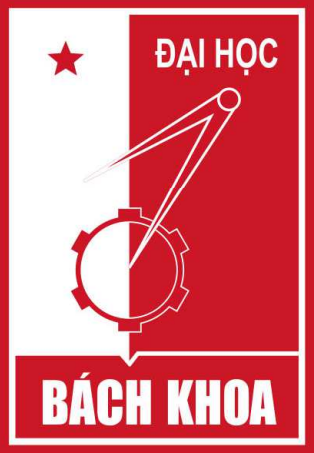


HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Shared Memory Programming-OpenMP

References

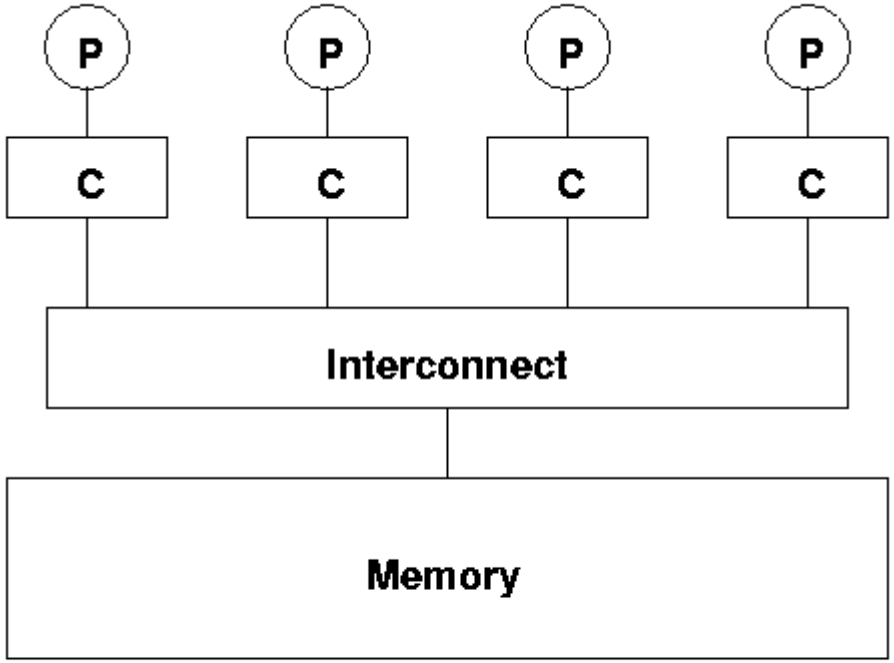
- Michael J. Quinn. **Parallel Computing. Theory and Practice**. McGraw-Hill
- Albert Y. Zomaya. **Parallel and Distributed Computing Handbook**. McGraw-Hill
- Ian Foster. **Designing and Building Parallel Programs**. Addison-Wesley.
- Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar **. Introduction to Parallel Computing, Second Edition.** Addison Wesley.
- Joseph Jaja**. An Introduction to Parallel Algorithm.** Addison Wesley.
- Nguyễn Đức Nghĩa**. Tính toán song song.** Hà Nội 2003.

4.1 Shared Memory

Shared Memory Architecture

- All processors have access to one global memory
- All processors share the same address space
- The system runs a single copy of the OS
- Processors communicate by reading/writing to the global memory
- Examples: multiprocessor PCs (Intel P4), Sun Fire 15K, NEC SX-7, Fujitsu PrimePower, IBM p690, SGI Origin 3000.

Shared Memory Systems



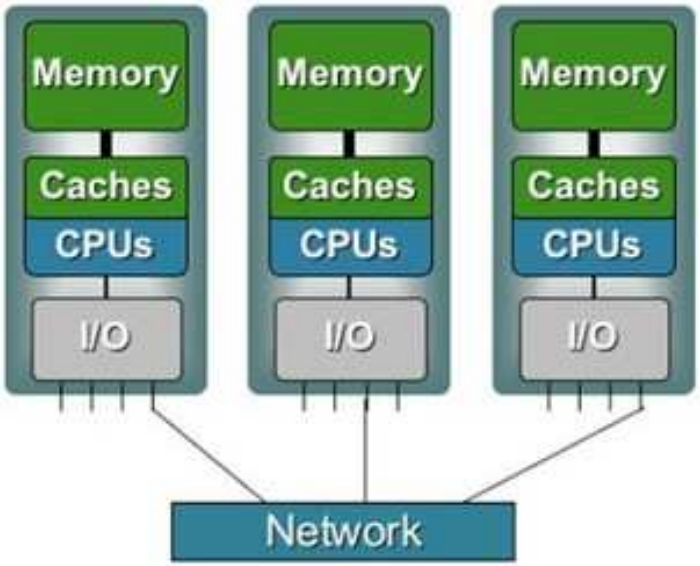
OpenMP

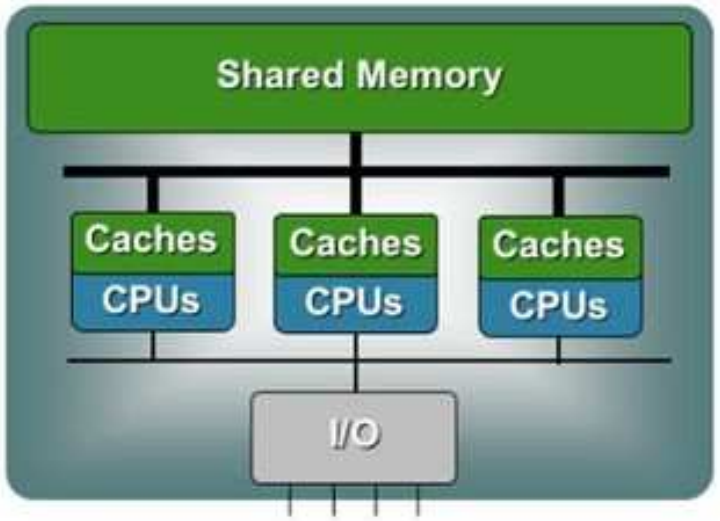
Pthreads

Shared memory

Shared Memory Systems (2)

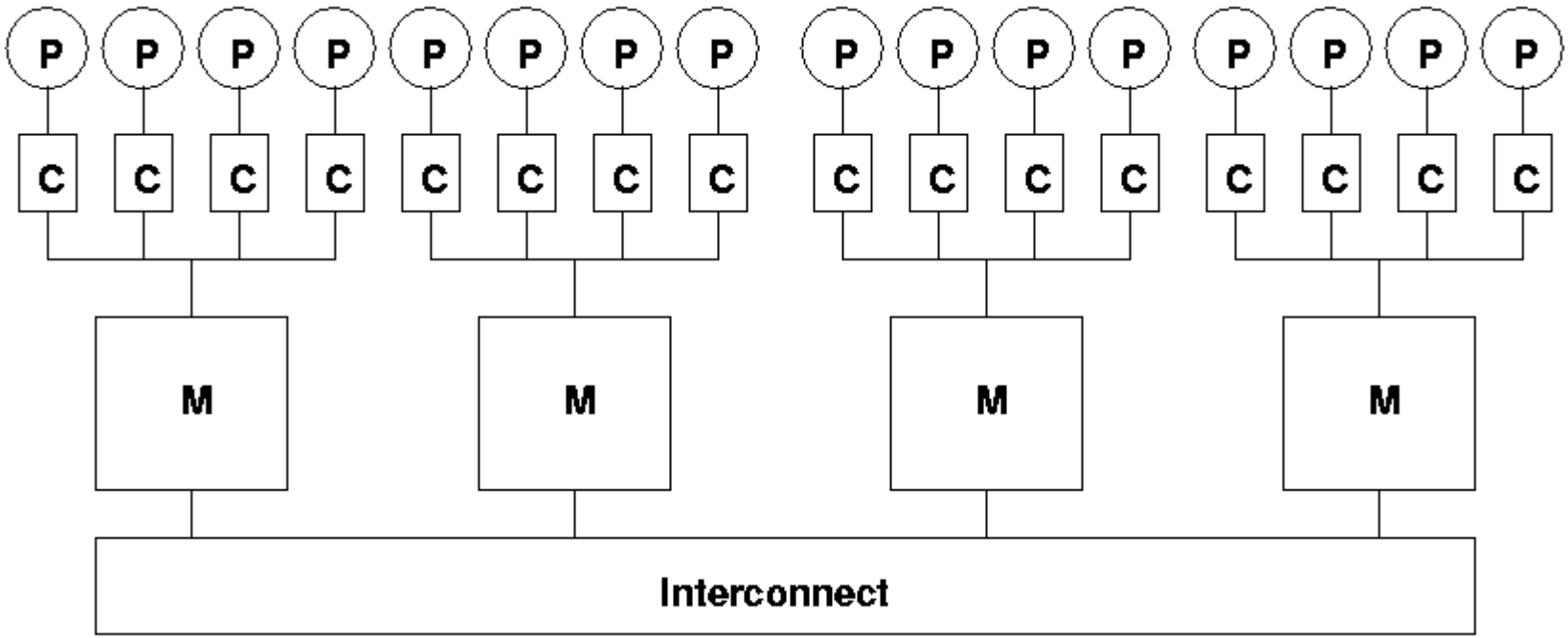
- All processors may access the whole main memory





- Non-UniformMemoryAccess
  - Memory access time is non-uniform
- UniformMemoryAccess
  - Memory access time is uniform

Clustered of SMPs



MPI

hybrid MPI + OpenMP

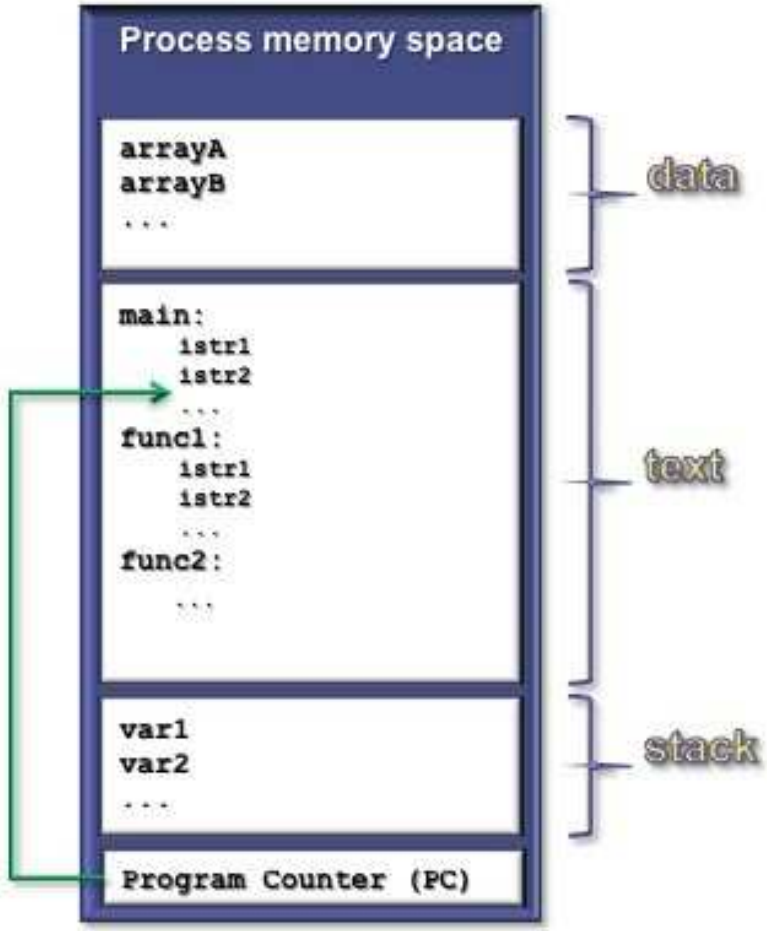
4.2 Multithread Programming

Shared Memory Programming

- Communication is implicitly specified
- Focus on constructs for expressing concurrency and synchronization
  - Minimize data-sharing overheads

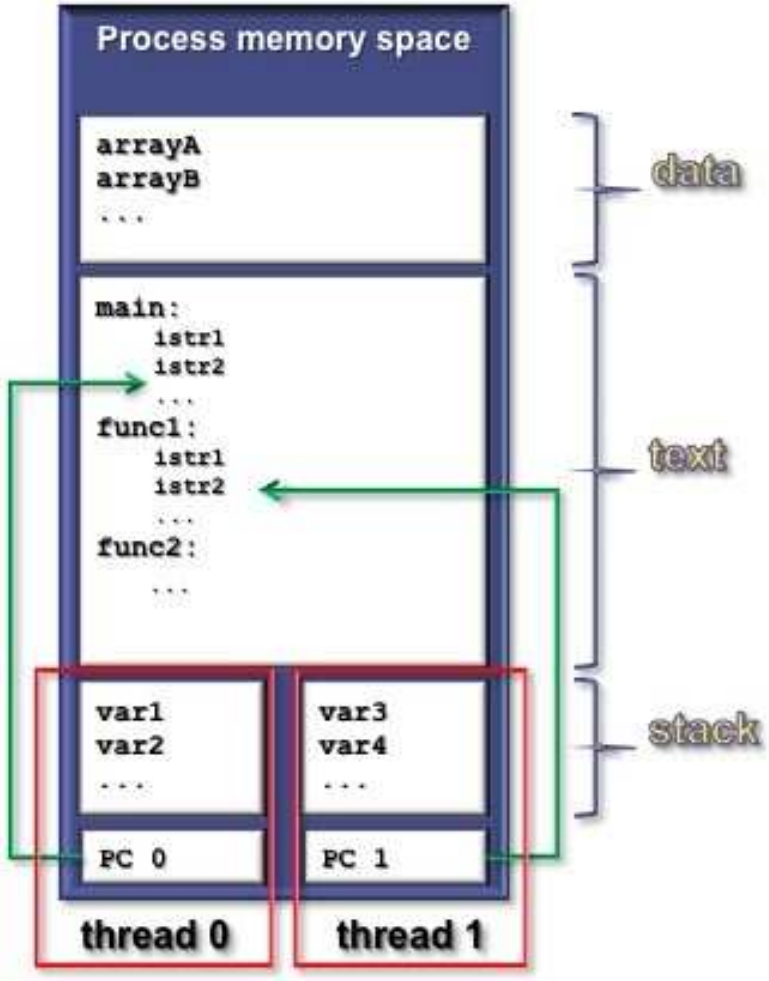
Process and thread

- A process is an instance of a computer program
- Information present in a process include:
  - Text
    - Machine code
  - Data
    - Global variables
  - Stack
    - Local variables
  - Program counter (PC)
    - A pointer to the istruction to be executed



Multi-threading

- The process contains several concurrent execution flows (threads)
  - Each thread has its own program counter (PC)
  - Each thread has its own private stack (variables local to the thread)
  - The instructions executed by a thread can access:
    - the process global memory (data)
    - the thread local stack



OpenMP vs. POSIX Threads

- POSIX threads is the other widely used shared programming API.
- Fairly widely available, usually quite simple to implement on top of OS kernel threads.
- Lower level of abstraction than OpenMP
  - library routines only, no directives
  - more flexible, but harder to implement and maintain
  - OpenMP can be implemented on top of POSIX threads
- Not much difference in availability
  - not that many OpenMP C++ implementations
  - no standard Fortran interface for POSIX threads

4.3 OPENMP

What is OpenMP?

- What does OpenMP stands for?
  - Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia.
- OpenMP is an Application Program Interface (API) that may be used to explicitly direct ***multi-threaded, shared memory parallelism.***
    - API components: Compiler Directives, Runtime Library Routines. Environment Variables
- OpenMP is a directive-based method to invoke parallel computations on share-memory multiprocessors

What is OpenMP?

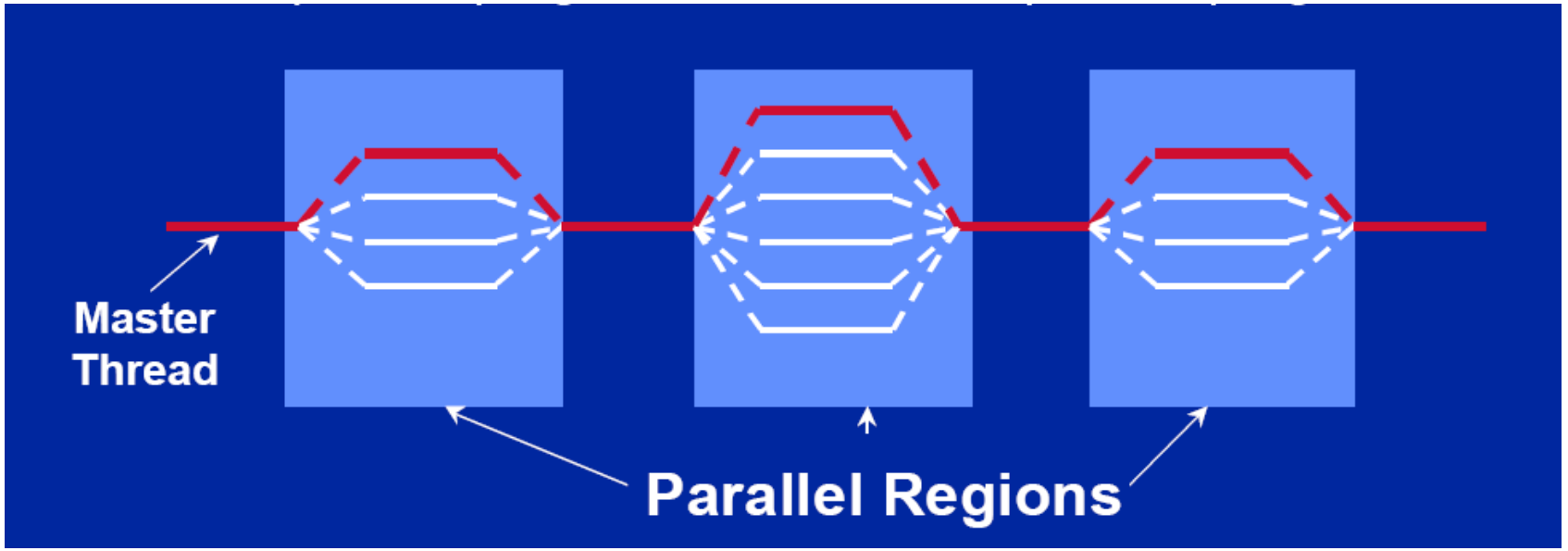
- OpenMP API is specified for C/C++ and Fortran.
- OpenMP is not intrusive to the original serial code: instructions appear in comment statements for fortran and pragmas for C/C++.
- OpenMP website: http://www.openmp.org
  - Materials in this lecture are taken from various OpenMP tutorials in the website and other places.

OpenMP Usage

- Applications
  - Applications with intense computational needs
  - From video games to big science & engineering
- Programmer Accessibility
  - From very early programmers in school to scientists to parallel computing experts
- Available to millions of programmers
  - In every major (Fortran & C/C++) compiler

Programming Model

- Fork-Join Parallelism:
  - Master thread spawns a team of threads as needed.
  - Parallelism is added incrementally: i.e. the sequential program evolves into a parallel program.



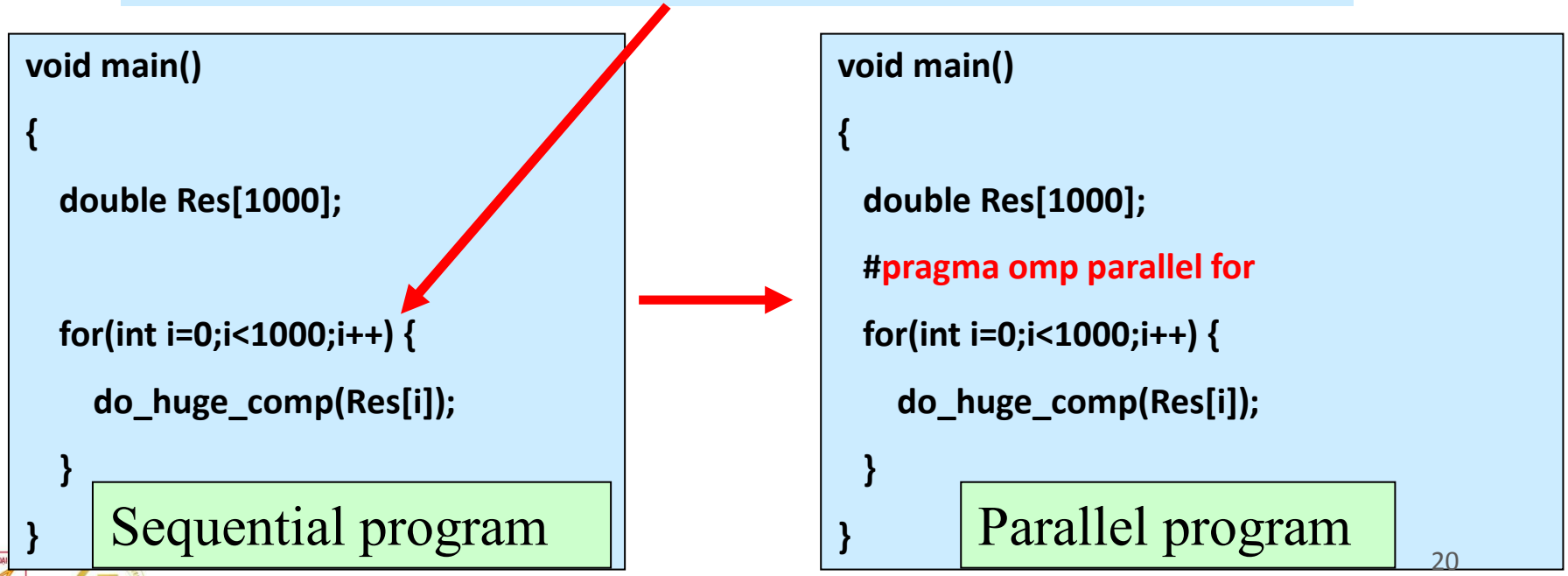
OpenMP Syntax

- Most of the constructs in OpenMP are compiler directives or pragmas.
  - For C and C++, the pragmas take the form:
    - #pragma omp *construct [clause [clause] …]*
  - For Fortran, the directives take one of the forms:
    - C$OMP *construct [clause [clause] …]*
    - !$OMP *construct [clause [clause] …]*
    - *$OMP *construct [clause [clause] …]*
- Since the constructs are directives, an OpenMP program can be compiled by compilers that don't support OpenMP.

OpenMP: How is OpenMP Typically Used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

Split-up this loop between multiple threads



How to compile and run OpenMP programs?

- Gcc 4.2 and above supports OpenMP 3.0
  - gcc –fopenmp a.c
  - Try example1.c
- To run: ‘a.out’
  - To change the number of threads:
    - setenv OMP_NUM_THREADS 4 (tcsh) or export OMP_NUM_THREADS=4(bash)

OpenMP:
How do Threads Interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.
- Unintended sharing of data can lead to race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is stored to minimize the need for synchronization.

OpenMP Constructs

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
- OpenMP is basically the same between Fortran and C/C++

OpenMP: Parallel Regions

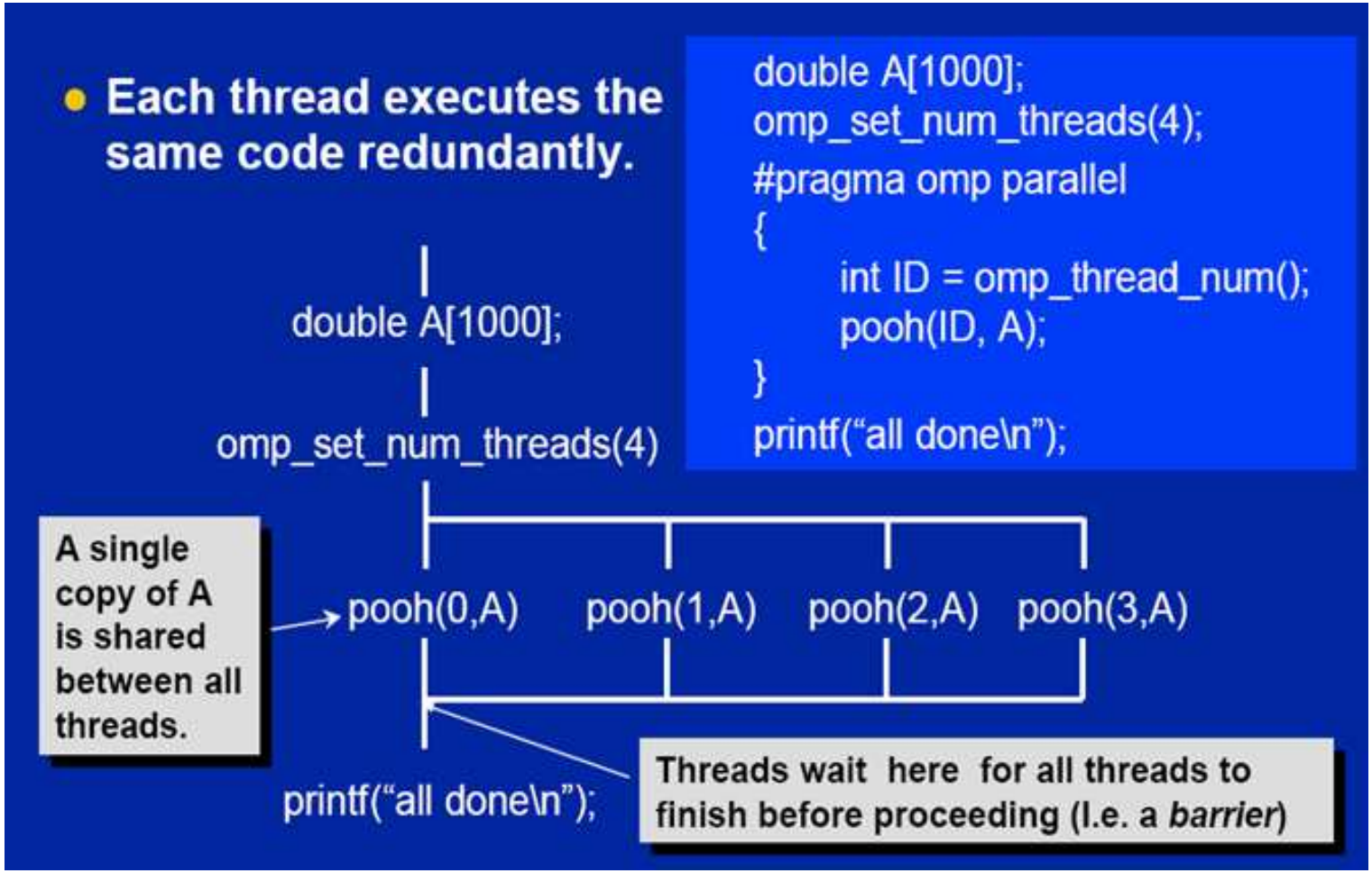
- You create threads in OpenMP with the "omp parallel" pragma.
- For example, To create a 4-thread Parallel region:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID =omp_get_thread_num();
    pooh(ID,A);
}
```

Each thread redundantly executes the code within the structured block

- Each thread calls pooh(ID,A) for `ID` = `0` to `3`

Parallel Region



OpenMP: Work-Sharing Constructs

- The “for” Work-Sharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
#pragma omp for
for (I=0;I<N;I++){
    NEAT_STUFF(I);
}
```

By default, there is a barrier at the end of the “omp for”. Use the “nowait” clause to turn off the barrier.

Work Sharing Constructs
A motivating example

Sequential code

```
for(i=0;I<N;i++) { a[i] = a[i] + b[i];}
```

OpenMP Parallel Region

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    for(i=istart;I<iend;i++) {a[i]=a[i]+b[i];}
}
```

OpenMP Parallel Region and a work-sharing for construct

```
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0;I<N;i++) { a[i]=a[i]+b[i];}
```

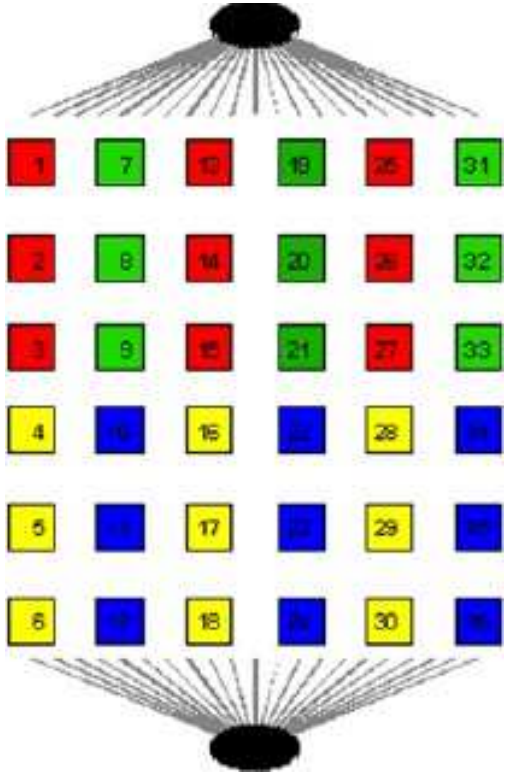
OpenMP For Construct:
The Schedule Clause

- The schedule clause effects how loop iterations are mapped onto threads
  - ◆schedule(static [,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - ◆schedule(dynamic[,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - ◆schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - ◆schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable.

static Scheduling

- Iterations are divided into chunks of size **`chunk`**, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number
- It is the default schedule and the default **`chunk`** is approximately $N_{iter}/N_{threads}$
- For example:

```
!$omp parallel do &
!$omp schedule(static,3)
```

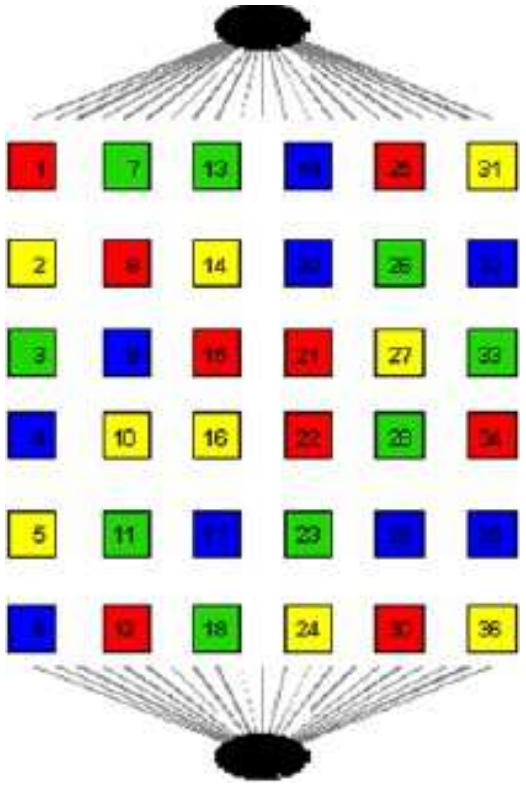


dynamic Scheduling

- Iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a **`chunk`** of iterations, then requests another **`chunk`**, until no chunks remain to be distributed.
- The default **`chunk`** is 1

- For example:

```
!$omp parallel do &
!$omp schedule(dynamic,1)
```

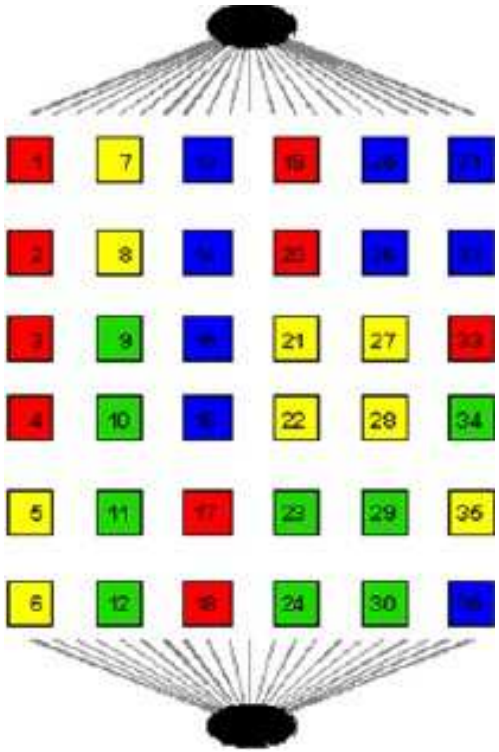


guided Scheduling

- Iterations are assigned to threads in the team in chunks as the executing threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. The chunk decreases to **`chunk`**
- The default value of **`chunk`** is 1

- For example:

```
!$omp parallel do &
!$omp schedule(guided,1)
```



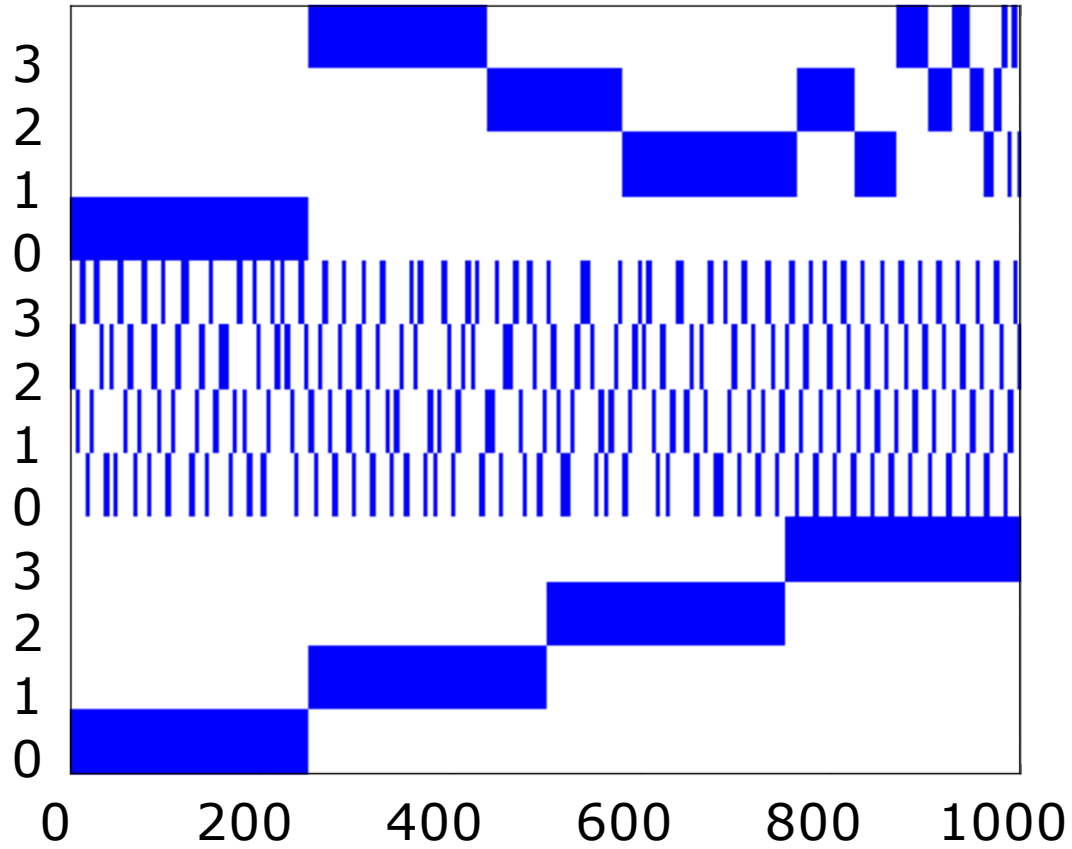
runtime and *auto* Scheduling

- **runtime**: iteration scheduling scheme is set at runtime through the enviroment variable **OMP_SCHEDULE**
  - For example:

    ```
    !$omp parallel do &
    !$omp schedule(runtime)
    ```

  - the scheduling scheme can be modified without recompiling the program changing the environment variable **OMP_SCHEDULE**, for example: **setenv OMP_SCHEDULE "dynamic,50"**
  - Only useful for experimental purposes during the parallelization
- **auto**: the decision regarding scheduling is delegated to the compiler and/or runtime system

Scheduling experiment



Different scheduling for a 1000 iterations loop with 4 threads: guided (top), dynamic (middle), static (bottom)

Sections: Work-Sharing Constructs

- The Sections work-sharing construct gives a different structured block to each thread.

```
#pragma omp parallel
#pragma omp sections
{
  X_calculation();
#pragma omp section
  y_calculation();
#pragma omp section
   z_calculation();
}
```

By default, there is a barrier at the end of the “omp sections”. Use the “nowait” clause to turn off the barrier.

Data-sharing attributes

- In a **`parallel`** construct the data-sharing attributes are *implicitily determined* by the **`default`** clause, if present
  - if no **`default`** clause is present they are shared
- Certain variables have a *predetermined* data-sharing attributes
  - Variables with automatic storage duration that are declared in a scope inside a construct are private
  - Objects with dynamic storage duration are shared
  - The loop iteration variable(s) in the associated for-loop(s) of a **`for`** construct is (are) private
  - A loop iteration variable for a sequential loop in a **`parallel`** construct is private in the innermost such construct that encloses the loop (only Fortran)
  - Variables with static storage duration that are declared in a scope inside the construct are shared
  - ...

Data-sharing attributes clauses

- *Explicitly determined* data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing attribute clause
  - **`shared(list)`**: there is only one istance of the objects in the list accessible by all threads in the team
  - **`private(list)`**: each thread has a copy of the variables in the list
  - **`firstprivate(list)`**: same as **`private`** but all variables in the list are initialized with the value that the original object had before entering the parallel construct
  - **`lastprivate(list)`**: same as **`private`** but the thread that executes the sequentially last iteration or section updates the value of the objects in the list
- The **`default`** clause sets the implicit default
  - **`default(none|shared)`** in C/C++

firstprivate

- Particular case of **private**.
  - Each private copy is initialized with the value of the variable of the **master** thread.

Example

```
void f() {
   int x= 17;
   #pragma omp parallel for firstprivate(x)
   for (long i=0;i<maxval;++i) {
      x+=i;       //xisinitially17
   }
   std::cout<<x<<std::endl;         //x==17
}
```

lastprivate

- Pass the value of the private variable of the last **sequential** iteration to the global variable.

Example

```
void f() {
   int x= 17;
   #pragma omp parallel for firstprivate(x) lastprivate(x)
   for (long i=0;i<maxval;++i) {
      x+=i;       //xisinitially17
   }
   std::cout<<x<<std::endl;        //xvalueiniterationi==maxval          −1
}
```

The `threadprivate` directive

C/C++

```
#pragma omp threadprivate(list)
```

- Is a declarative directive
- Is used to create private copies of
  - *file-scope*, *namespace-scope* or **`static`** variables in C/C++
- Follows the variable declaration in the same program unit
- Initial data are undefined, unless the **`copyin`** clause is used

Data Environment:
Default Storage Attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
- But not everything is shared...
  - Stack variables in sub-programs called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

Private Clause

- private(var) creates a local copy of var for each thread.
  - The value is uninitialized
  - Private copy is *not* storage associated with the original

```
void wrong(){
    int IS = 0;
    #pragma parallel for private(IS)
    for(int J=1;J<1000;J++)
        IS = IS + J;
    printf("%i", IS);
}
```

OpenMP: Reduction

- Another clause that effects the way variables are shared:
  - reduction (op : list)
- The variables in “list” must be shared in the enclosing parallel region.
- Inside a parallel or a worksharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”)
  - pair wise “op” is updated on the local value
  - Local copies are reduced into a single global copy at the end of the construct.

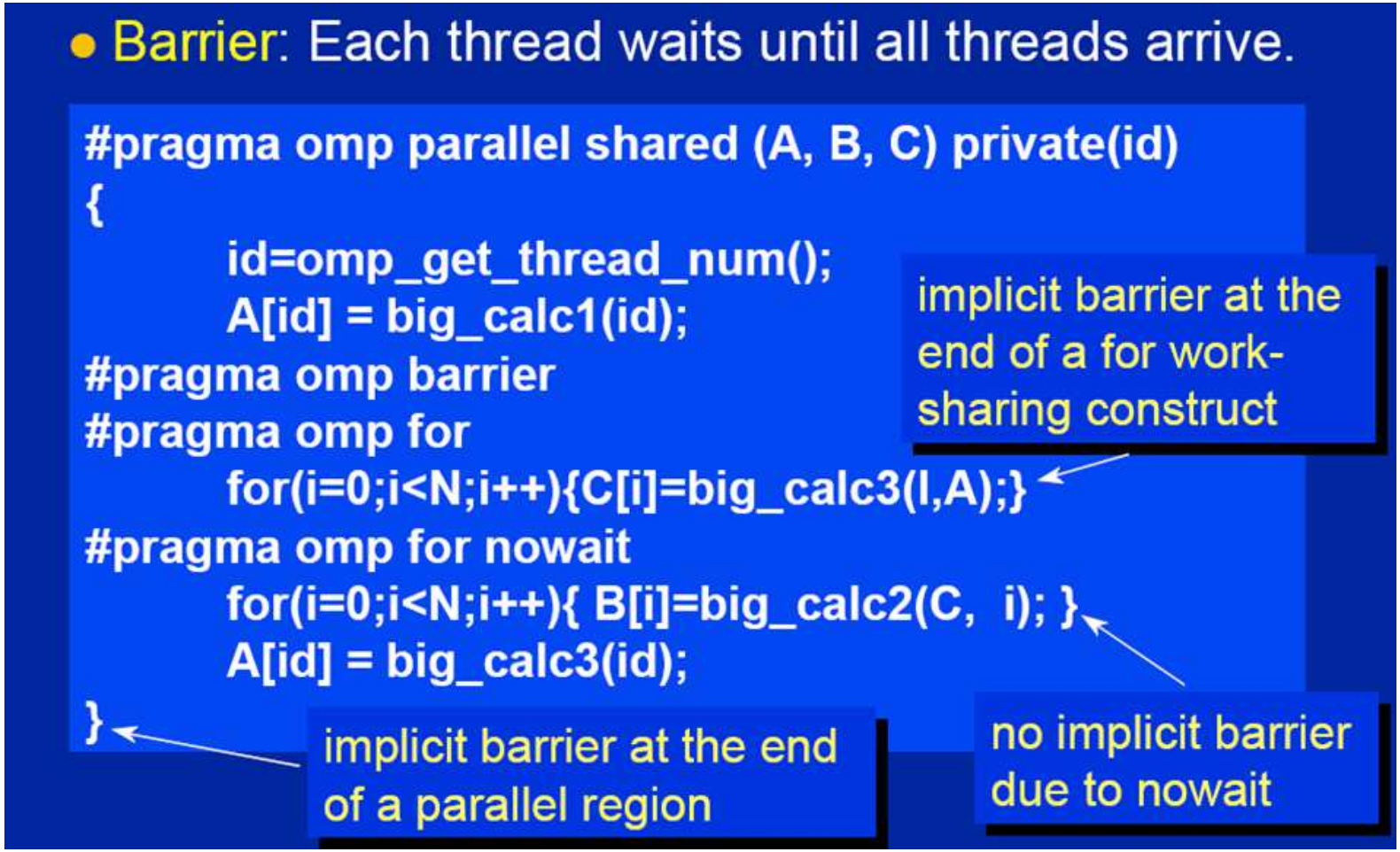
OpenMP:
An Reduction Example

```
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
  int i;
  double ZZ, func(), sum=0.0;
  omp_set_num_threads(NUM_THREADS)
  #pragma omp parallel for reduction(+:sum)
  private(ZZ)
  for (i=0; i< 1000; i++){
        ZZ = func(i);
        sum = sum + ZZ;
  }
}
```

OpenMP: Synchronization

- OpenMP has the following constructs to support synchronization:
  - barrier
  - critical section
  - atomic
  - flush
  - ordered
  - single
  - master

Synchronization



`atomic` construct

- The **`atomic`** construct applies only to statements that update the value of a variable
  - Ensures that no other thread updates the variable between reading and writing
- The allowed instructions differ between Fortran and C/C++
  - Refer to the OpenMP specifications
- It is a special lightweight form of a **`critical`**
  - Only read/write are serialized, and only if two or more threads access the same memory address

C/C++

```
#pragma omp atomic [clause]
 <statement>
```

`atomic` Examples

C/C++

```
#pragma omp atomic update
x += n*mass; // default update

#pragma omp atomic read
v = x; // read atomically

#pragma omp atomic write
x = n*mass; write atomically

#pragma omp atomic capture
v = x++; // capture x in v and
         // update x atomically
```

Critical session

- Only one thread at a time can enter a critical section

```
For(I=0; I<N; I++) {
   ……
   sum += A[I];
   ……
}
```

Cannot be parallelized if sum is shared.

Fix:

```
For(I=0; I<N; I++) {
   ……
   #pragma omp critical
   {
     sum += A[I];
   }
   ……
}
```

Master directive

- The master construct denotes a structured block that is only executed by the master thread. The other threads just skip it (no implied barriers or flushes).

```
#pragma omp parallel private (tmp)
{
  do_many_things();
  #pragma omp master
  { exchange_boundaries(); }
  #pragma barrier
  do_many_other_things();
}
```

Single directive

- The single construct denotes a block of code that is executed by only one thread.
- A barrier and a flush are implied at the end of the single block.

```
#pragma omp parallel private (tmp)
{
    do_many_things();
    #pragma omp single
    { exchange_boundaries(); }
    do_many_other_things();
}
```

Ordering

- An ordered region is executed in sequential order.

```
#pragma omp parallel
{
#pragma omp for ordered reduction(+:res)
for ( int i =0;i<max;++i) {
double tmp = f(i) ;
#pragma ordered
res += g(tmp);
}
}
```

Simple locks

- Locks in the OpenMP library.
- Also nested locks.

```
omp_lock_t l;
omp_init_lock(&l);
#pragma omp parallel
{
int id = omp_get_thread_num();
double x = f( i ) ;
omp_set_lock(&l);
cout << "ID=" << id << " tmp= " << tmp << endl;
omp_unset_lock(&l);
}
omp_destroy_lock(&l);
```

OpenMP: Library routines

- Lock routines
  - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock()
- Runtime environment routines:
  - Modify/Check the number of threads
    - omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()
  - Turn on/off nesting and dynamic mode
    - omp_set_nested(), omp_set_dynamic(), omp_get_nested(), omp_get_dynamic()
  - Are we in a parallel region?
    - omp_in_parallel()
  - How many processors in the system?
    - omp_num_procs()

OpenMP: Environment Variables

- OMP_NUM_THREADS
  - bsh:
    - export OMP_NUM_THREADS=2
  - csh:
    - setenv OMP_NUM_THREADS 4

CODE. Matrix-vector multiply using a parallel loop and critical directive

```
/*** Spawn a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads,chunk)
private(tid,i,j,k)
{
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    {
      printf("thread=%d did row=%d\n",tid,i);
      for(j=0; j<NCB; j++)
      for (k=0; k<NCA; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
}
```

Ex.Travelling Salesman Problem

- The map is represented as a graph with nodes representing cities and edges representing the distances between cities.
- A special node (cities) is the starting point of the tour.
- Travelling salesman problem is to find the circle (starting point) that covers all nodes with the smallest distance.
- This is a well known NP-complete problem.

CODE. Sequential TSP

```
Init_q(); init_best();
While ((p = dequeue()) != NULL) {
    for each expansion by one city {
        q = addcity (p);
        if (complete(q)) {update_best(q);}
        else enqueue(q);
    }
}
```

CODE. OpenMP TSP

```
Do_work() {
  While ((p = dequeue()) != NULL) {
    for each expansion by one city {
        q = addcity (p);
        if (complete(q)) {update_best(q);}
        else enqueue(q);
    } } }
main() {
   init_q(); init_best();
   #pragma omp parallel for
   for (i=0; I < NPROCS; i++)
      do_work();
}
```

Summary

- OpenMP provides a compact, yet powerful programming model for shared memory programming
  - It is very easy to use OpenMP to create parallel programs.
- OpenMP preserves the sequential version of the program
- Developing an OpenMP program:
  - Start from a sequential program
  - Identify the code segment that takes most of the time.
  - Determine whether the important loops can be parallelized
    - The loops may have critical sections, reduction variables, etc
  - Determine the shared and private variables.
  - Add directives

Conclusion

- OpenMP is successful in small-to-medium SMP systems
- Multiple cores/CPUs dominate the future computer architectures; OpenMP would be the major parallel programming language in these architectures.
- Simple: everybody can learn it in 2 weeks
- Not so simple: Don’t stop learning! keep learning it for better performance

OpenMP discussion

- Ease of use
  - OpenMP takes cares of the thread maintenance.
    - Big improvement over pthread.
  - Synchronization
    - Much higher constructs (critical section, barrier).
    - Big improvement over pthread.

- OpenMP is easy to use!!

OpenMP discussion

- Expressiveness
  - Data parallelism:
    - MM and SOR
    - Fits nicely in the paradigm
  - Task parallelism:
    - TSP
    - Somewhat awkward. Use OpenMP constructs to create threads. OpenMP is not much different from pthread.

OpenMP discussion

- Exposing architecture features (performance):
  - Not much, similar to the pthread approach
    - Assumption: dividing job into threads = improved performance.
    - How valid is this assumption in reality?
      - Overheads, contentions, synchronizations, etc
  - This is one weak point for OpenMP: the performance of an OpenMP program is somewhat hard to understand.

OpenMP final thoughts

- Main issues with OpenMP: performance
  - Is there any obvious way to solve this?
    - Exposing more architecture features?
  - Is the performance issue more related to the fundamantal way that we write parallel program?
    - OpenMP programs begin with sequential programs.
    - May need to find a new way to write efficient parallel programs in order to really solve the problem.

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions !

soict.hust.edu.vn/ fb.com/groups/soict