The Abdus Salam
International Centre
for Theoretical Physics

Shared Memory Programming Paradigm

Ivan Girotto – igirotto@ictp.it

Information & Communication Technology Section (ICTS)

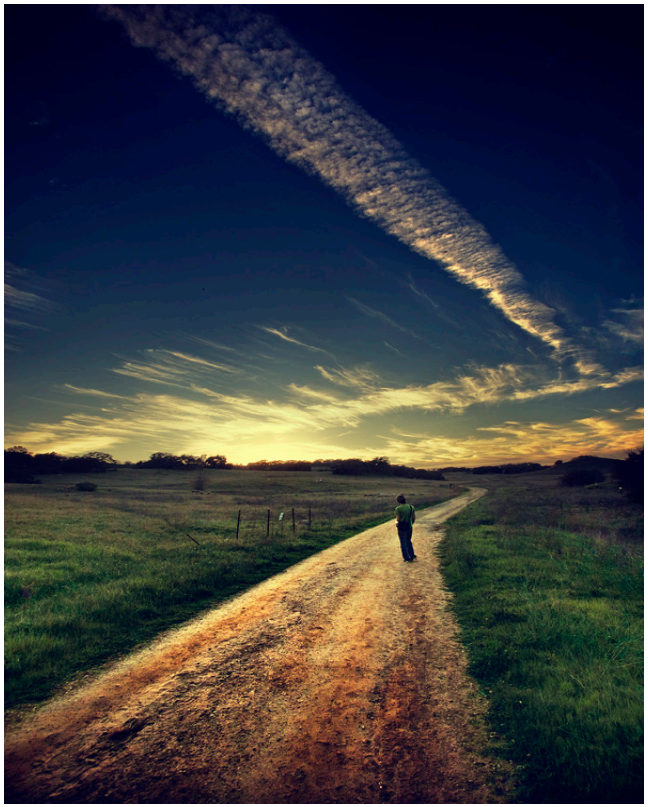
International Centre for Theoretical Physics (ICTP)

OUTLINE

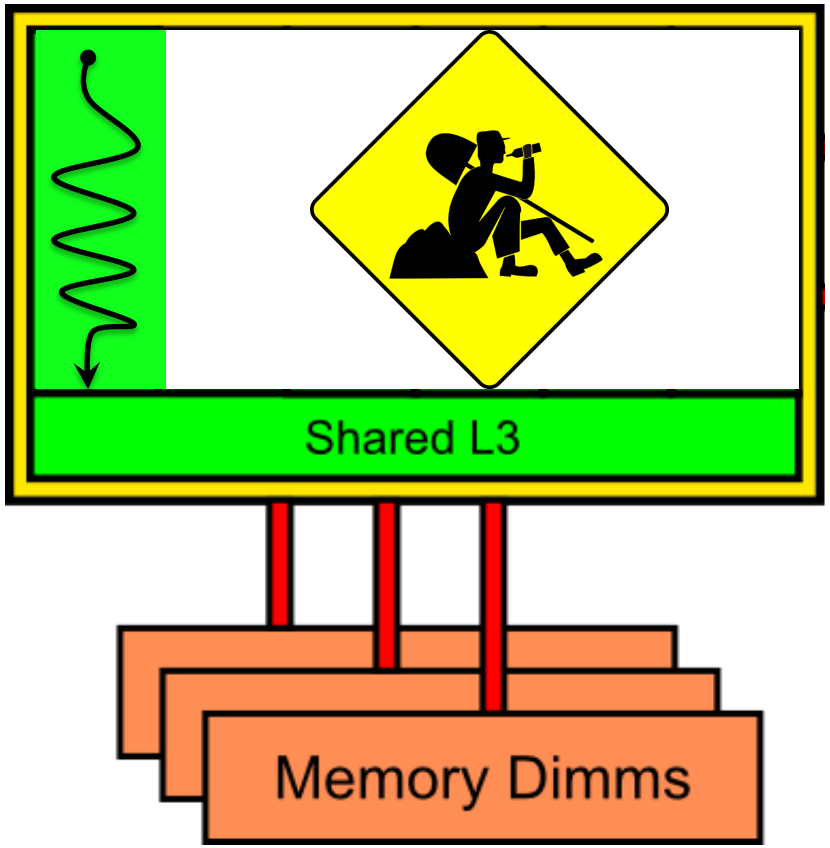
- The Shared Memory Programming Paradigm
- Processes and Threads
- The OpenMP Standard
- Parallel Programming with OpenMP
- Hands-on

The Recommended Approach

- Parallelism is no longer an option for only either larger scale problems or improve the time of response
- It is inescapable to exploit current & next generations of compute processors

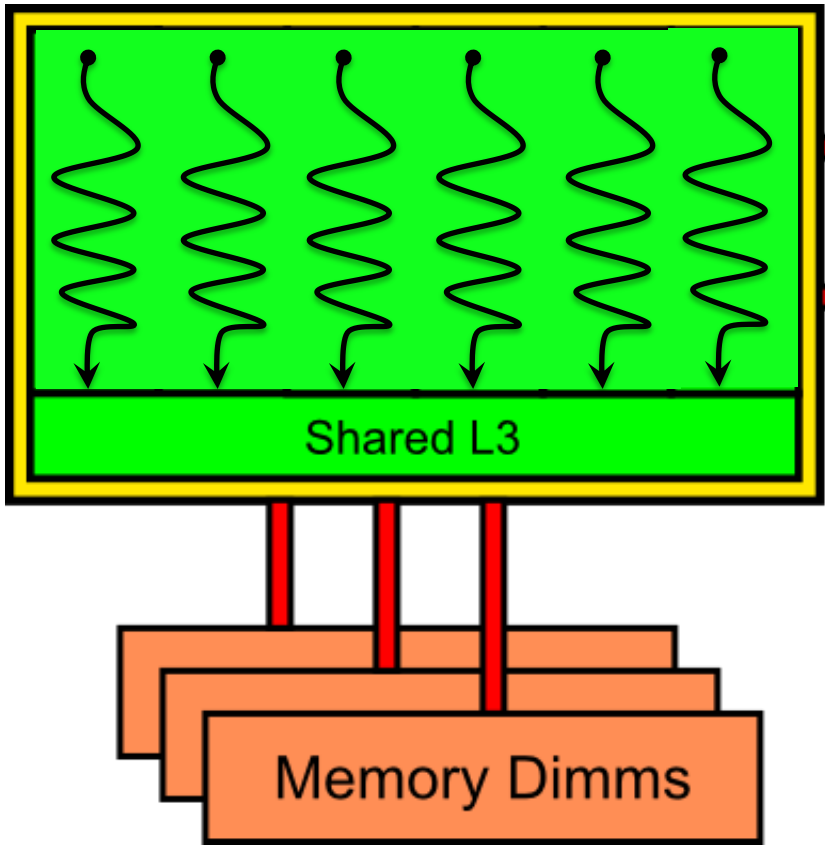


Multi-core system Vs Serial Programming



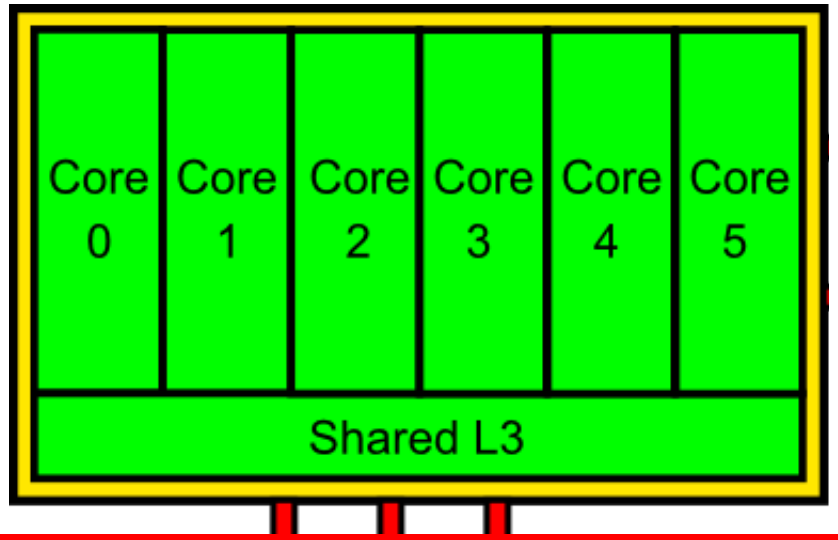
Xeon E5650 hex-core processors (12GB - RAM)

Multi-core system Vs // Programming



Xeon E5650 hex-core processors (12GB - RAM)

Multi-CPUs & Multi-cores system

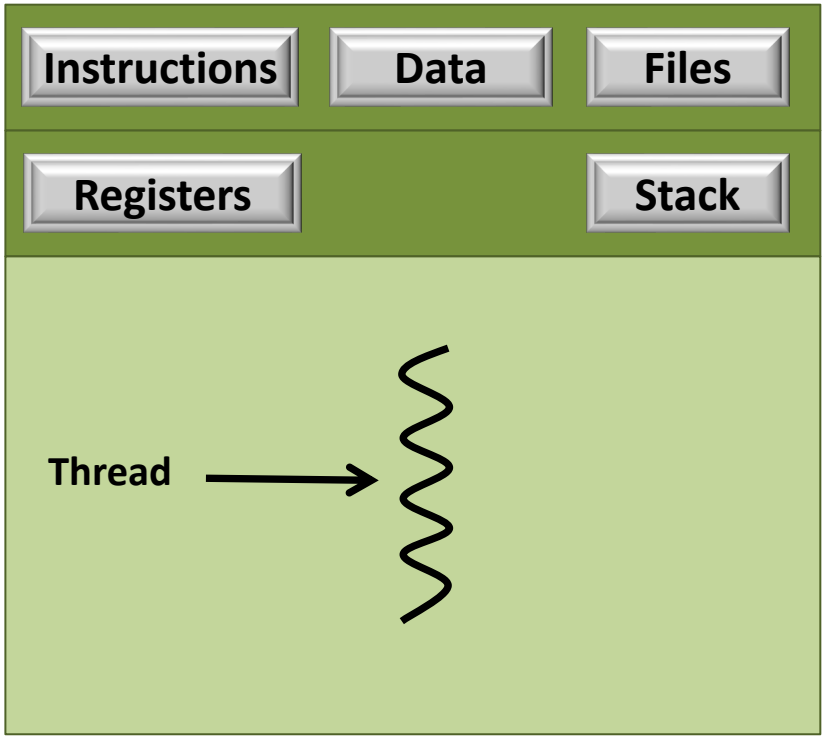


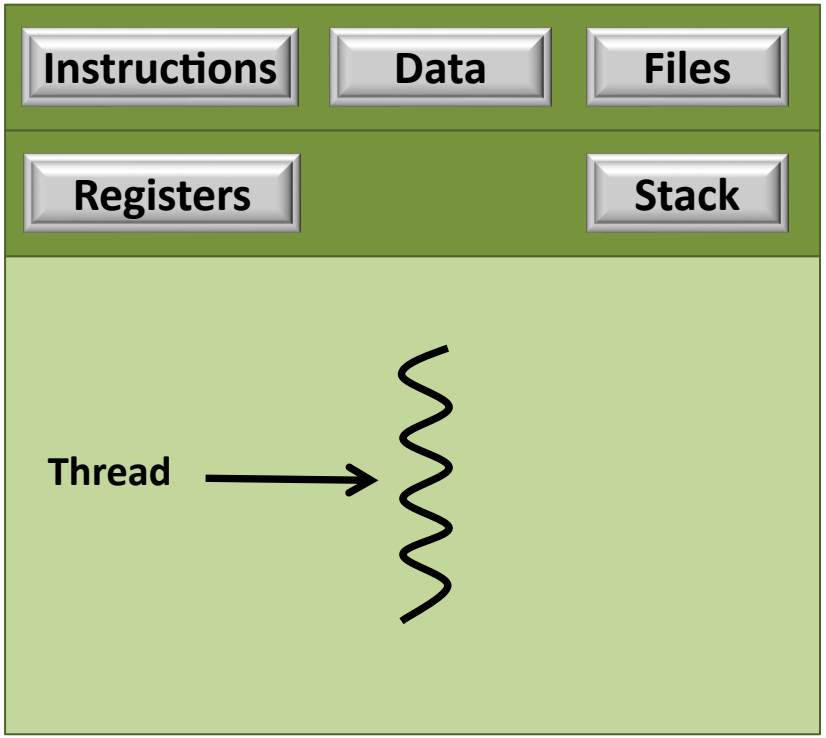
Xeon E5650 hex-core processors

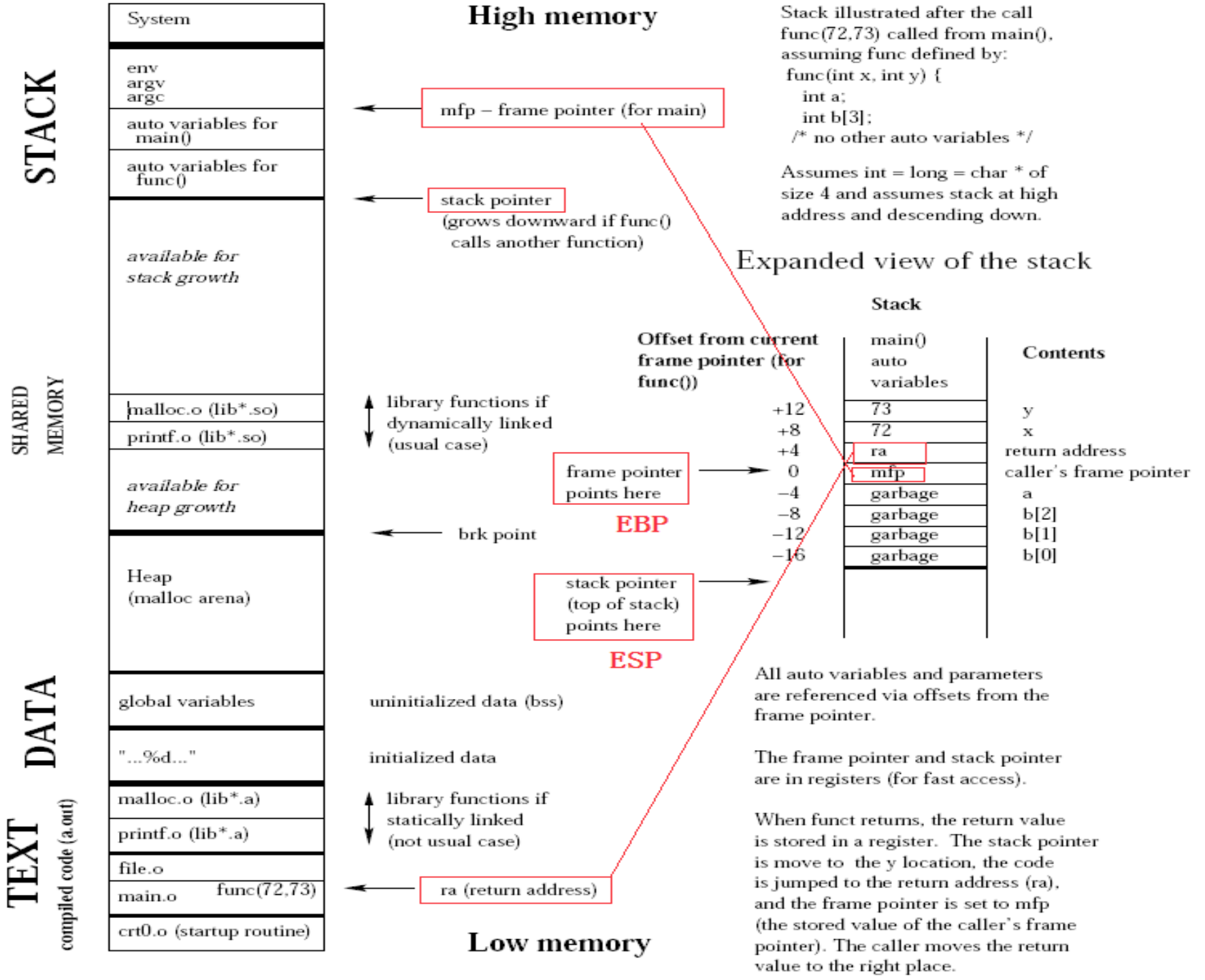
Main Memory

Dual Socket (Westmere) - 24GB RAM

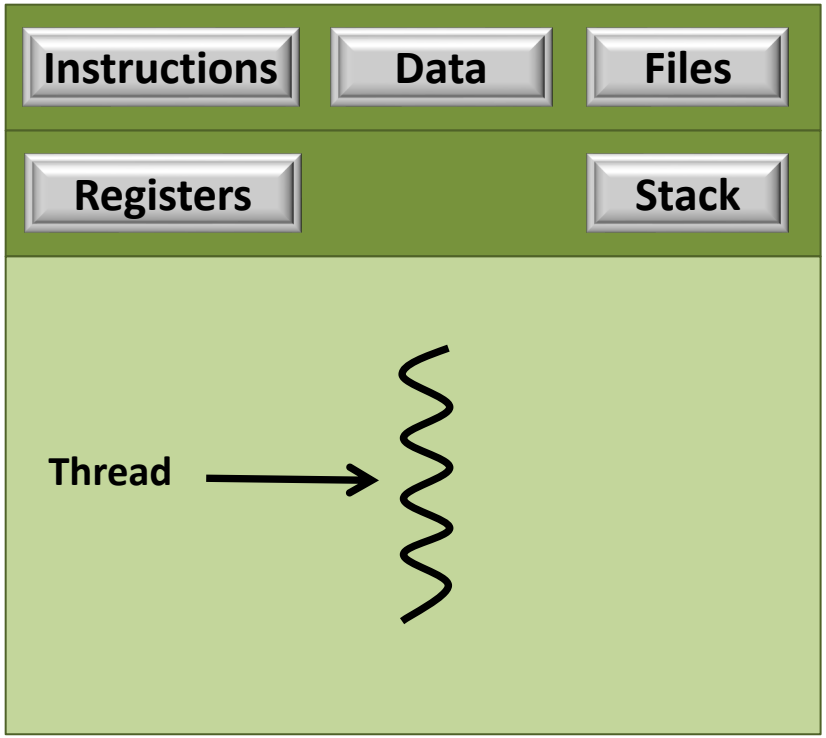
Processes and Threads

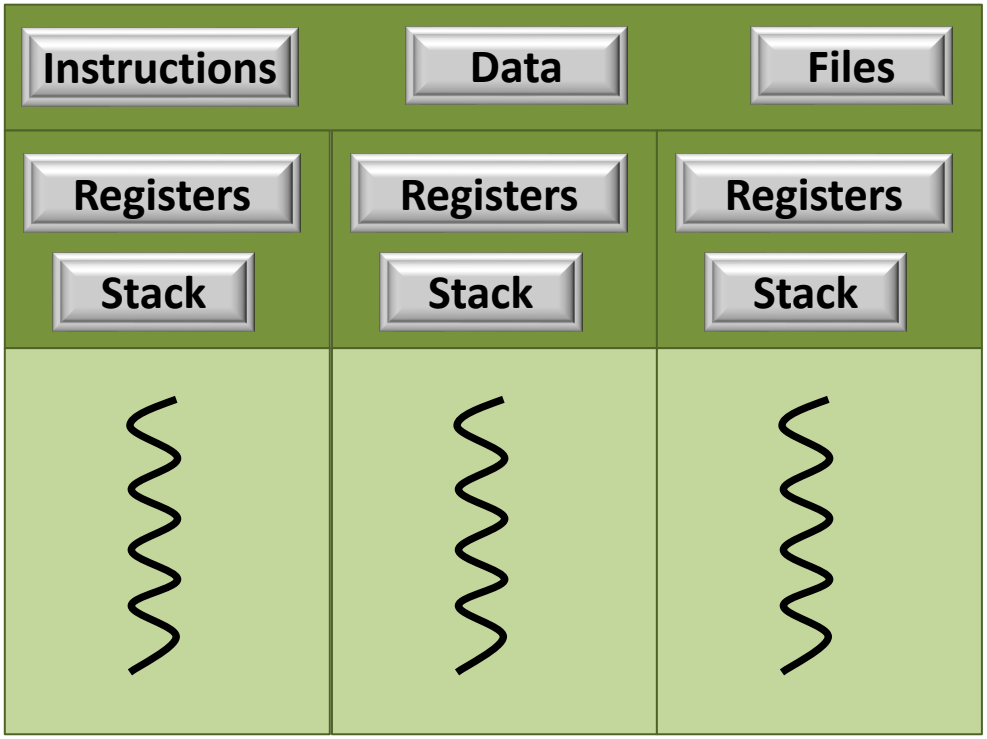





High memory
System
STACK
env
argv
argc
auto variables for main()
auto variables for func()
available for stack growth
SHARED MEMORY
malloc.o (lib*.so)
printf.o (lib*.so)
available for heap growth
Heap (malloc arena)
DATA
global variables
"...%d..."
TEXT
compiled code (a.out)
malloc.o (lib*.a)
printf.o (lib*.a)
file.o
main.o
func(72,73)
crt0.o (startup routine)
Low memory
mfp – frame pointer (for main)
stack pointer
(grows downward if func() calls another function)
library functions if dynamically linked (usual case)
brk point
uninitialized data (bss)
initialized data
library functions if statically linked (not usual case)
ra (return address)
Stack illustrated after the call func(72,73) called from main(), assuming func defined by:
func(int x, int y) {
int a;
int b[3];
/* no other auto variables */
Assumes int = long = char * of size 4 and assumes stack at high address and descending down.
Expanded view of the stack
Stack
Offset from current frame pointer (for func())
main() auto variables
Contents
+12 73 y
+8 72 x
+4 ra return address
0 mfp caller's frame pointer
−4 garbage a
−8 garbage b[2]
−12 garbage b[1]
−16 garbage b[0]
frame pointer points here
EBP
stack pointer (top of stack) points here
ESP
All auto variables and parameters are referenced via offsets from the frame pointer.
The frame pointer and stack pointer are in registers (for fast access).
When funct returns, the return value is stored in a register. The stack pointer is move to the y location, the code is jumped to the return address (ra), and the frame pointer is set to mfp (the stored value of the caller's frame pointer). The caller moves the return value to the right place.

Processes and Threads





Multi-threading - Recap

- A thread is a (lightweight) process - an instance of a program plus its own data (private memory)
- Each thread can follow its own flow of control through a program
- Threads can share data with other threads, but also have private data
- Threads communicate with each other via the shared data.
- A *master thread* is responsible for co-ordinating the threads group


OpenMP
TM

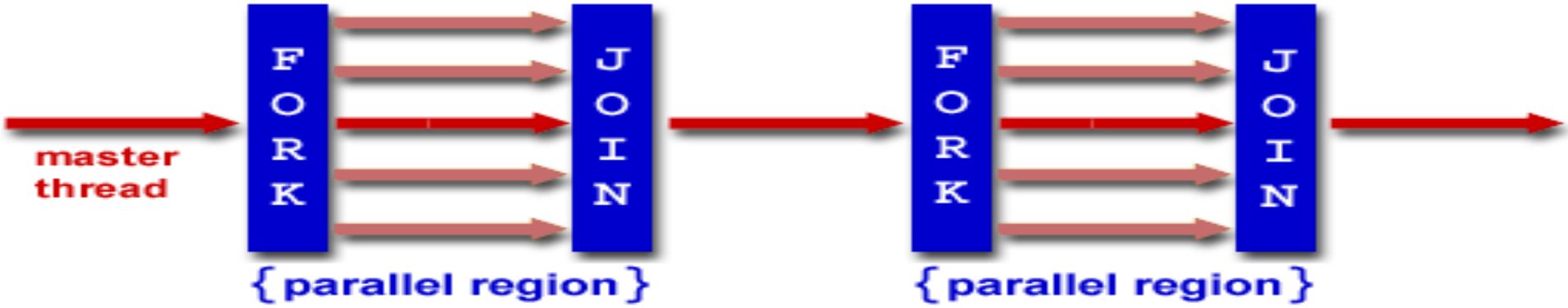
OpenMP (*Open spec. for Multi Processing*)

- OpenMP is not a computer language
  - Rather it works in conjunction with existing languages such as standard Fortran or C/C++
- Application Programming Interface (API)
  - that provides a portable model for parallel applications
  - Three main components:
    - Compiler directives
    - Runtime library routines
    - Environment variables

OpenMP Parallelization

- OpenMP is directive based
  - code (can) work without them
- OpenMP can be added incrementally
- OpenMP only works in shared memory
  - multi-socket nodes, multi-core processors
- OpenMP hides the calls to a threads library
  - less flexible, but much less programming
- **Caution:** write access to shared data can easily lead to race conditions and incorrect data

OpenMP Parallelization



- Thread-based Parallelism
- Explicit Parallelism
- Fork-Join Model
- Compiler Directive Based
- Dynamic Threads

Getting Started with OpenMP

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Work sharing
  - Data Environment (scope)
  - Synchronization
  - Runtime functions/environment variables
- OpenMP is essentially the same for both Fortran and C/C++

Directives Format

- A directive is a special line of source code with meaning only to certain compilers.
- A directive is distinguished by a sentinel at the start of the line.
- OpenMP sentinels are:
  - Fortran: **!$OMP** (or **C$OMP** or ***$OMP**)
  - C/C++: **#pragma omp**

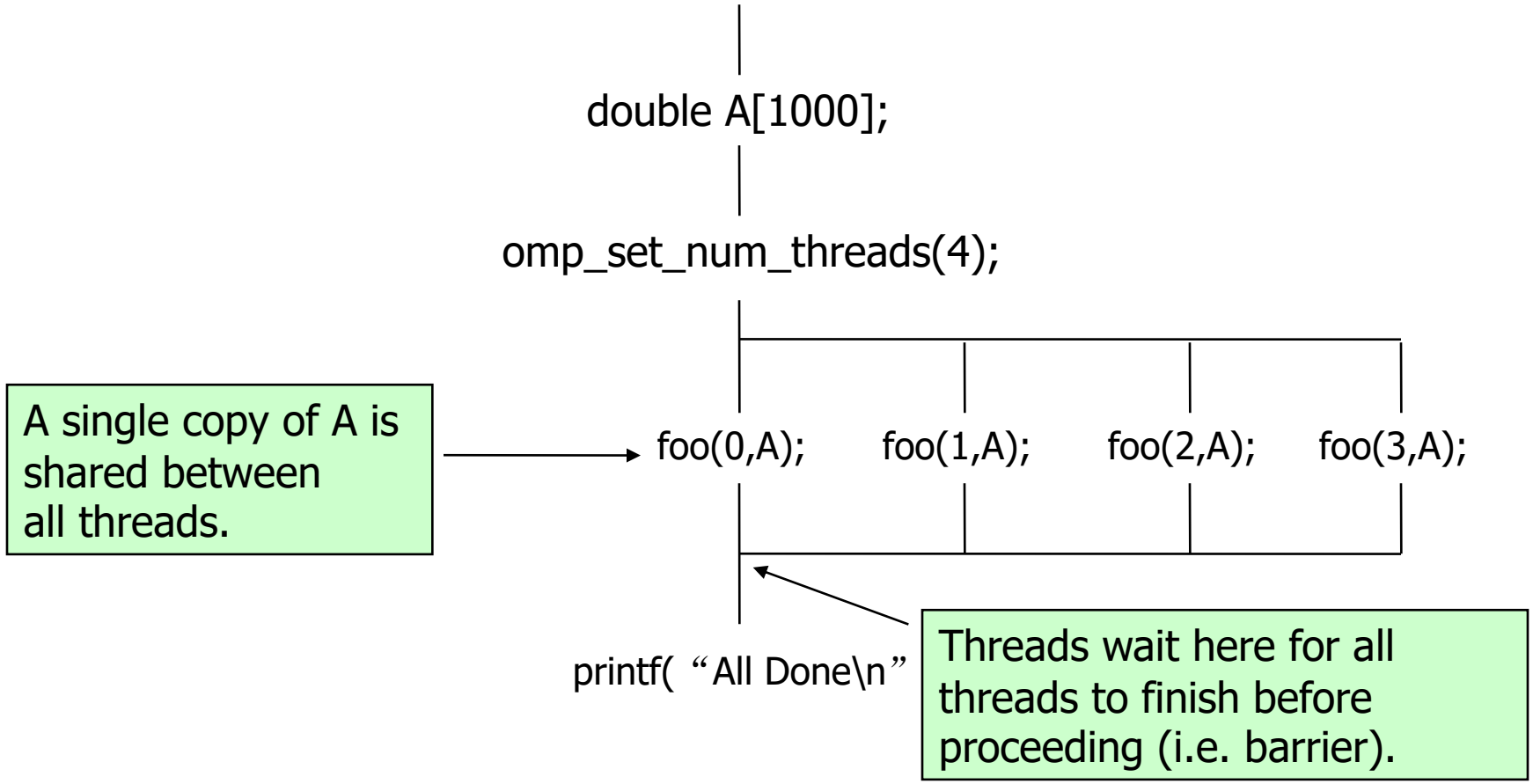
OpenMP: Parallel Regions

- For example, to create a 4-thread parallel region:
  - each thread calls foo(ID,A) for **ID** = **0** to **3**

Each thread redundantly executes the code within the structured block

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID =omp_get_thread_num();
   foo(ID,A);
}
printf( “All Done\n” );
```

thread-safe routine: A routine that performs the intended function even when executed concurrently (by more than one thread)


double A[1000];
omp_set_num_threads(4);
A single copy of A is shared between all threads.
foo(0,A);
foo(1,A);
foo(2,A);
foo(3,A);
printf("All Done\n")
Threads wait here for all threads to finish before proceeding (i.e. barrier).

How many threads?

The number of threads in a parallel region is determined by the following factors:

- Use of the omp_set_num_threads() library function
- Setting of the OMP_NUM_THREADS environment variable
- The implementation default

Threads are numbered from 0 (master thread) to N-1.

OpenMP runtime library

OMP_GET_NUM_THREADS() – returns the current # of threads.

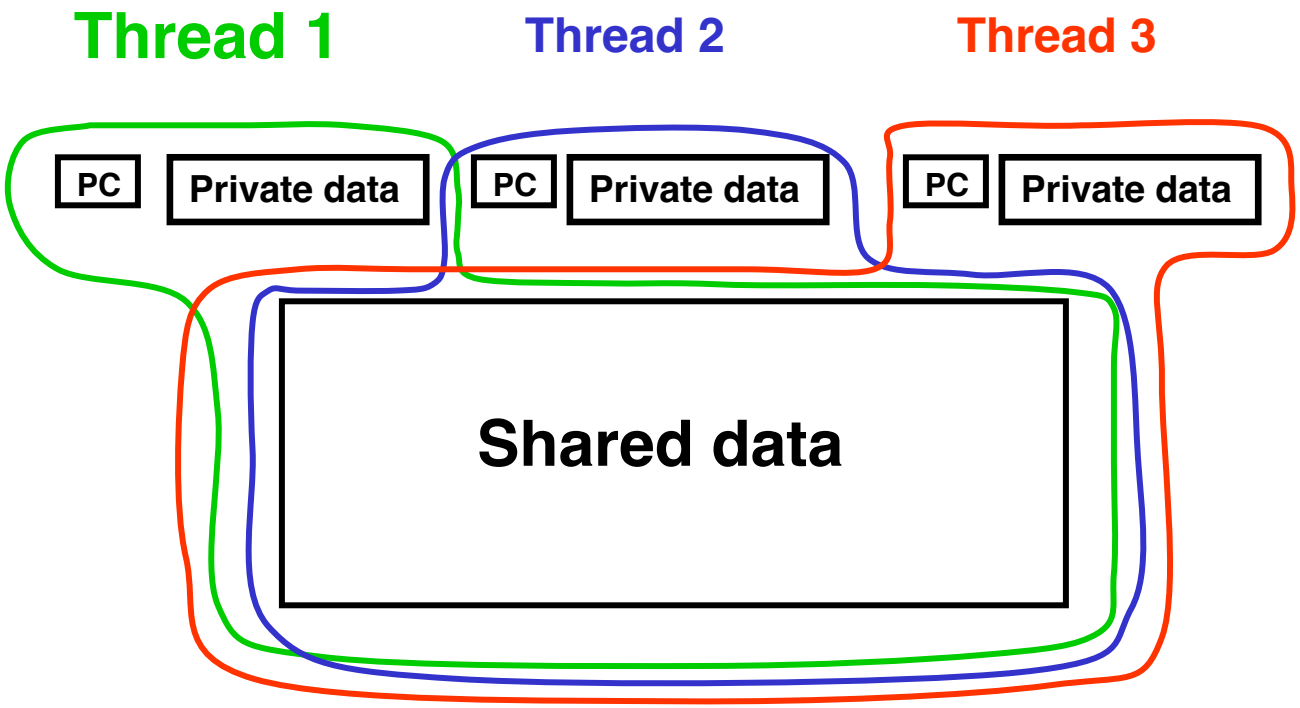
OMP_GET_THREAD_NUM() - returns the id of this thread.

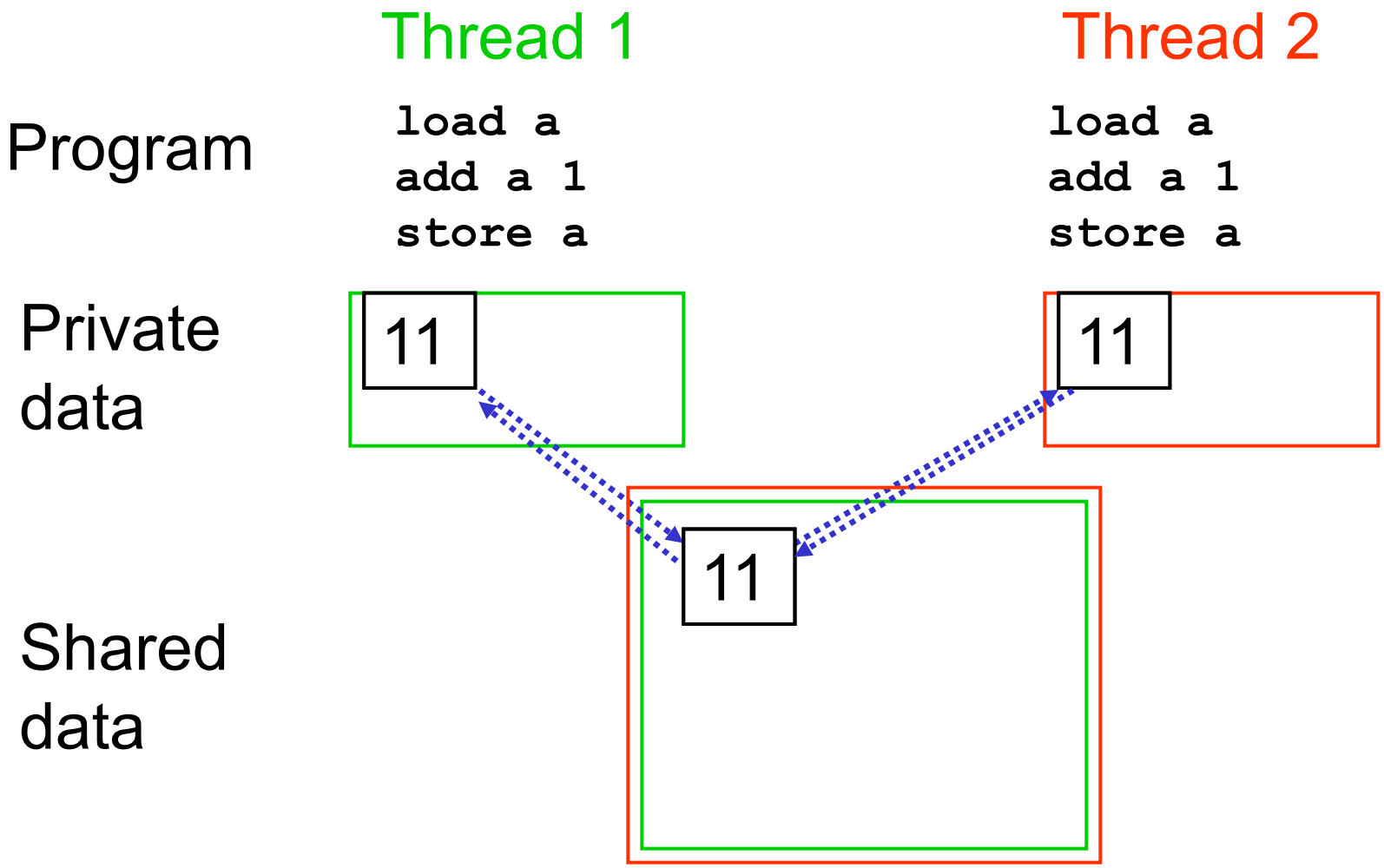
OMP_SET_NUM_THREADS(n) – set the desired # of threads.

OMP_IN_PARALLEL() – returns .true. if inside parallel region.

OMP_GET_MAX_THREADS() - returns the # of possible threads.

Memory footprint




Thread 1
Thread 2
Program
load a
add a 1
store a
load a
add a 1
store a
Private data
11
11
Shared data
11

Simple C OpenMP Program

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

int main ( ) {

   printf("Starting off in the sequential world.\n");
   #pragma omp parallel
   {
      printf("Hello from thread number %d\n", omp_get_thread_num() );
   }
   printf("Back to the sequential world.\n");
   return 0;
}
```

Variable Scooping

- All existing variable still exist inside a parallel region
  - by default SHARED between all threads
- But work sharing requires private variables
  - PRIVATE clause to OMP PARALLEL directive
  - Index variable of a worksharing loop
  - All declared local variable within a parallel region
  - The FIRSTPRIVATE clause would initialize the private instances with the contents of the shared instance
- Be aware of the sharing nature of static variables

Exploiting Loop Level Parallelism

Loop level Parallelism: parallelize only loops

- Easy to implement
- Highly readable code
- Less than optimal performance (sometimes)
- Most often used

Parallel Loop Directives

- Fortran do loop directive
  - !$omp do
- C\C++ for loop directive
  - #pragma omp for
- These directives do not create a team of threads but assume there has already been a team forked.
- If not inside a parallel region shortcuts can be used.
  - !$omp parallel do
  - #pragma omp parallel for

Parallel Loop Directives continued

- These are equivalent to a parallel construct followed immediately by a worksharing construct.

```
!$omp parallel do

Same as

!$omp parallel
...
!$omp do
```

```
#pragma omp parallel for

Same as

#pragma omp parallel
...
#pragma omp for
```

How is OpenMP Typically Used?

- OpenMP is usually used to parallelize loops:

Split-up this loop between multiple threads

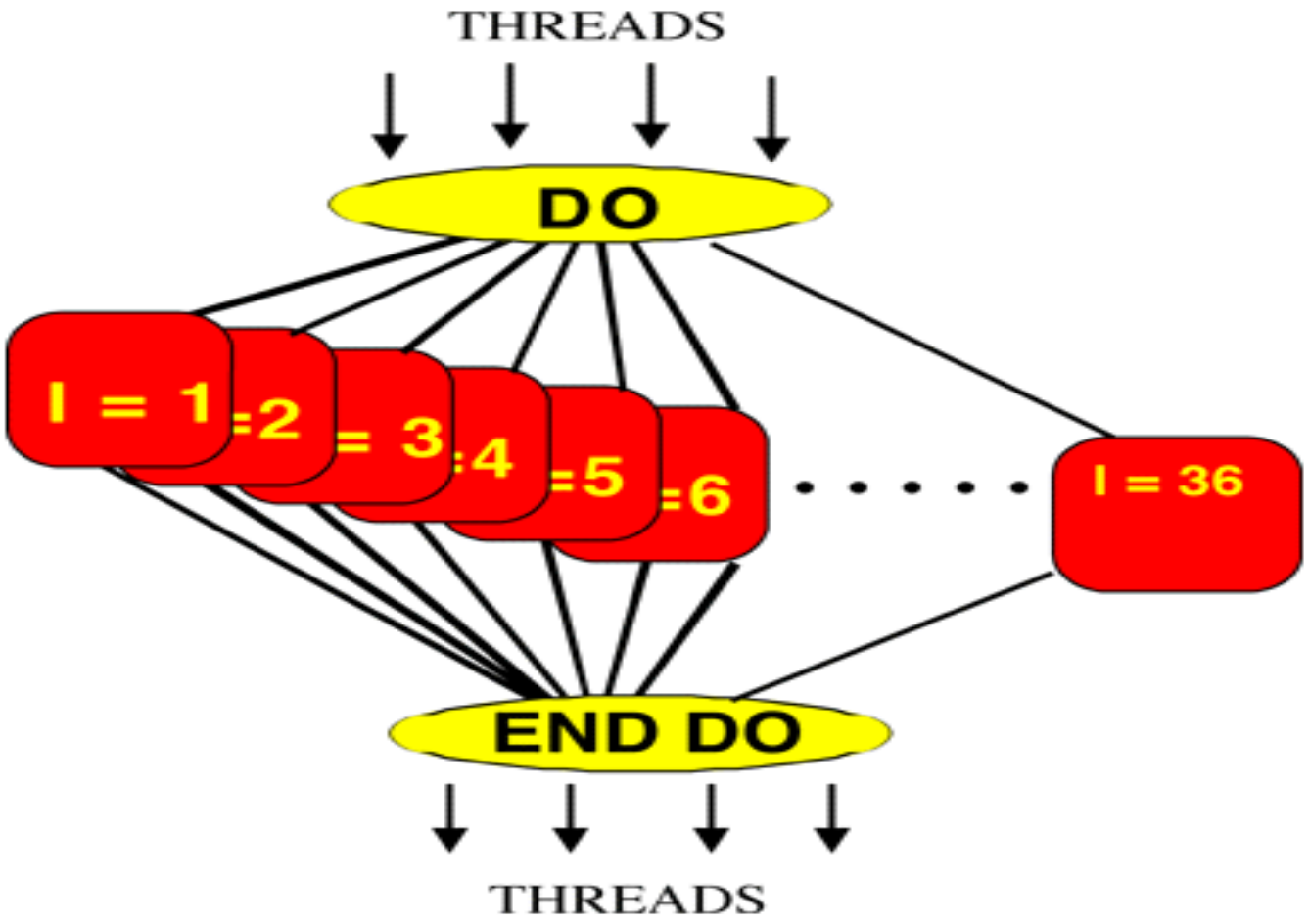
```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
   }
}
```

Sequential program

```
void main()
{
   double Res[1000];
   #pragma omp parallel for
   for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
   }
}
```

Parallel program


THREADS
DO
I = 1
=2
= 3
:4
=5
=6
I = 36
END DO
THREADS

Work-Sharing Constructs

- Divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs **do not** launch new threads.
- No implied barrier upon entry to a work sharing construct.
- However, there is an implied barrier at the end of the work sharing construct (unless nowait is used).

Work Sharing Constructs - example

Sequential code

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

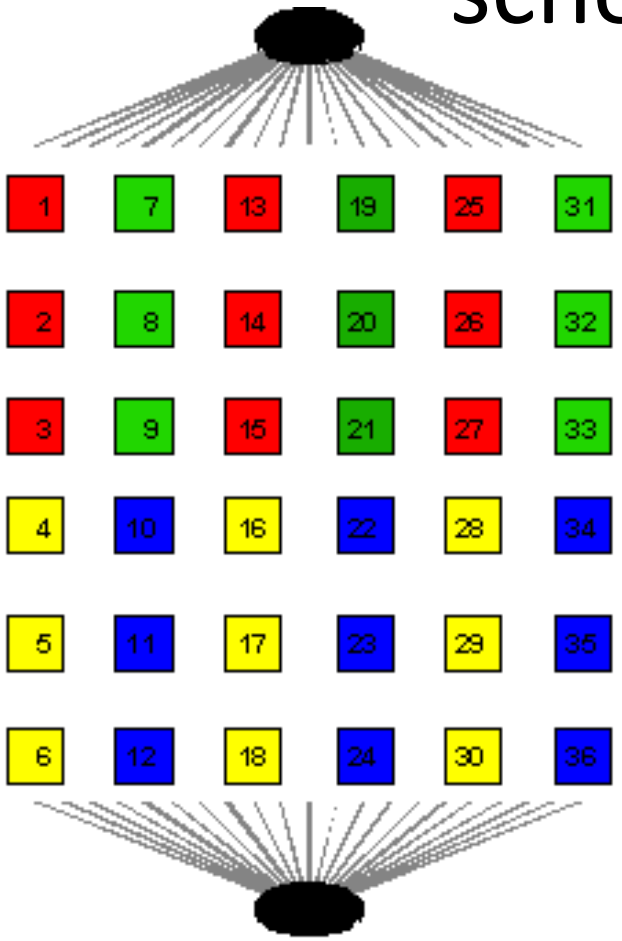
OpenMP // Region

```
#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];}
```

schedule(static [,chunk])



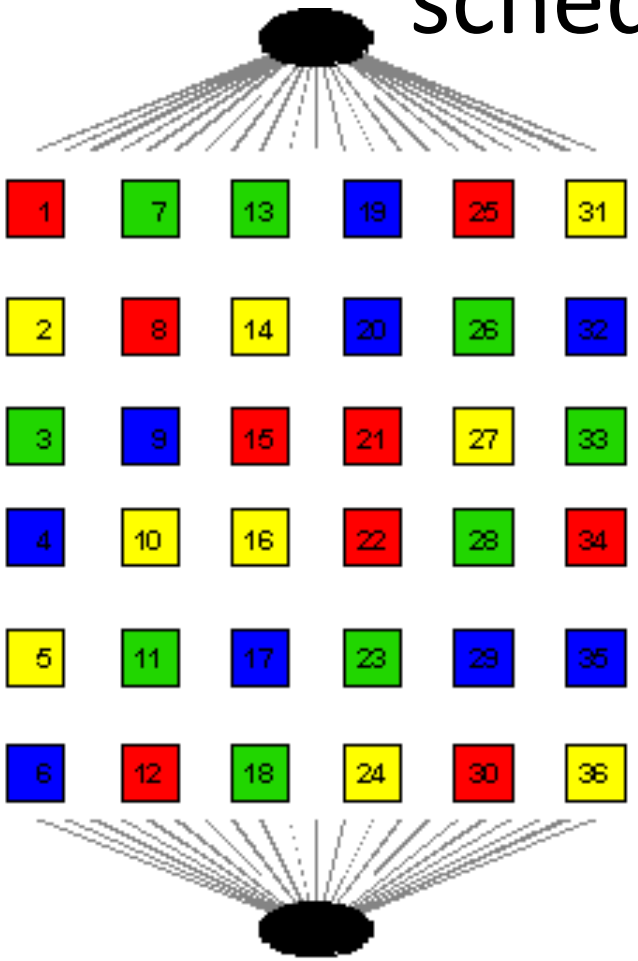
- Iterations are divided evenly among threads
- If chunk is specified, divides the work into chunk sized parcels
- If there are N threads, each thread does every N^{th} chunk of work.

```
!$OMP PARALLEL DO &
!$OMP SCHEDULE(STATIC,3)

DO J = 1, 36
Work (j)
END DO

!$OMP END DO
```

schedule(dynamic [,chunk])



- Divides the workload into chunk sized parcels.
- As a thread finishes one chunk, it grabs the next available chunk.
- Default value for chunk is one.
- More overhead, but potentially better load balancing.

```
!$OMP PARALLEL DO &
!$OMPSCHEDULE(DYNAMIC,1)

DO J = 1, 36
Work (j)
END DO

!$OMP END DO
```

The Schedule Clause SCHEDULE (type [,chunk])

- 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)

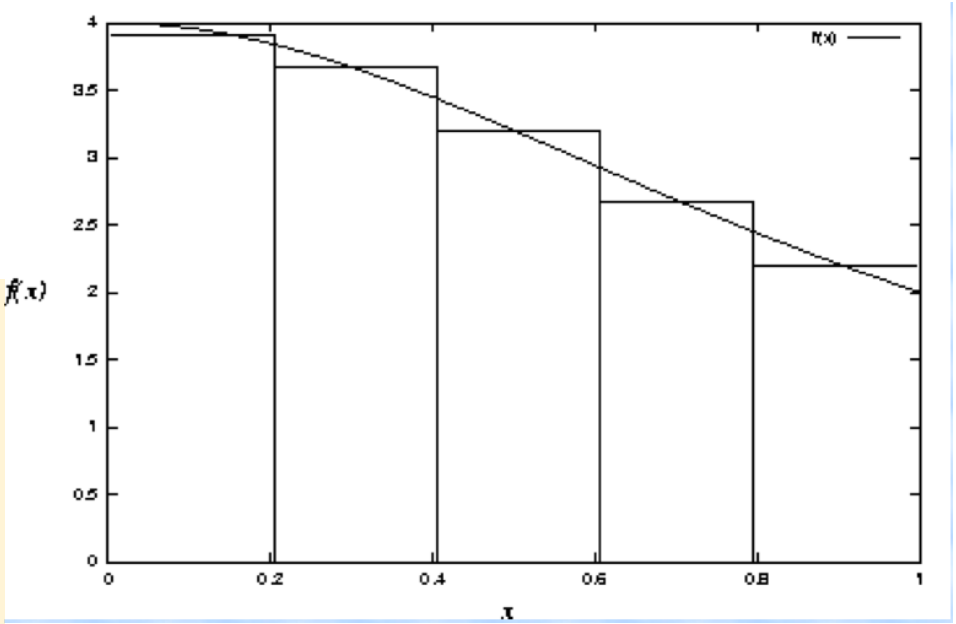
- Schedule and chunk size taken from the OMP_SCHEDULE environment variable

Compute PI

$$\int_0^1 \frac{1}{1+x^2} dx = arctan(x)\Big|_0^1 = arctan(1) - arctan(0) = arctan(1) = \frac{\pi}{4}$$

$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx$$

Integrate, i.e determine area under function numerically using slices of h * f(x) at midpoints



No Wait Clauses

- No wait: if specified then threads do not synchronise at the end of the parallel loop.
- For Fortran, the END DO directive is optional with NO WAIT being the default.
- Note that the nowait clause is incompatible with a simple parallel region meaning that using the composite directives will not allow you to use the nowait clause.

OpenMP: 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: A 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;
   }
}
```

if CLAUSE

We can make the parallel region directive itself conditional.

Fortran: IF (*scalar logical expression*)

C/C++: if (*scalar expression*)

```
#pragma omp parallel if (tasks > 1000)
{
  while(tasks > 0) donexttask();
}
```

SYNCHRONIZATION

OpenMP: How do Threads Interact?

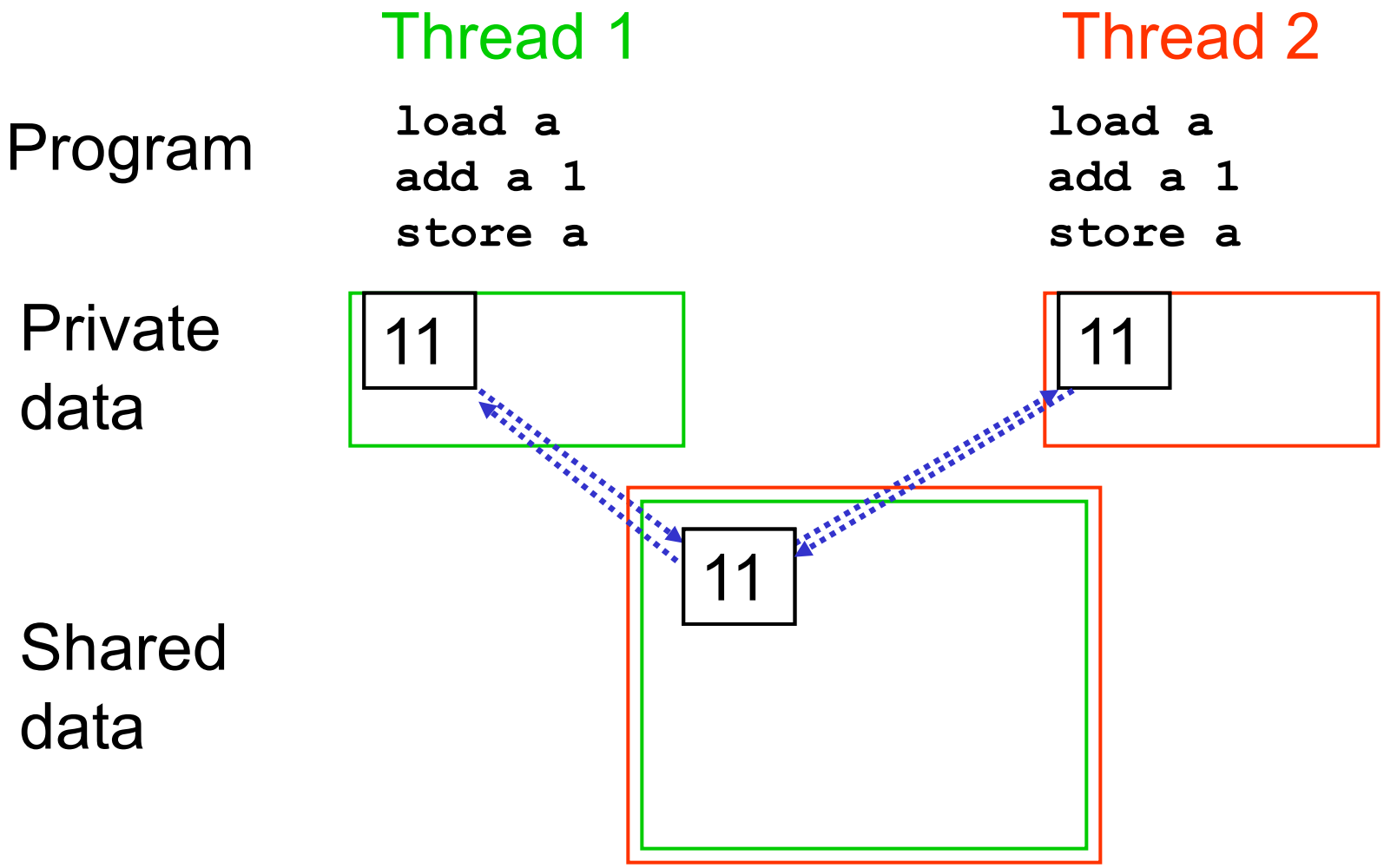
- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.
- Unintended sharing of data can lead to race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect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

Note that updates to shared variables:

(e.g. a = a + 1)

are *not* atomic!

If two threads try to do this at the same time, one of the updates may get overwritten.


Thread 1
Thread 2
Program
load a
add a 1
store a
load a
add a 1
store a
Private data
11
11
Shared data
11

Barrier

Fortran - !$OMP BARRIER

C\C++ - #pragma omp barrier

- This directive synchronises the threads in a team by causing them to wait until all of the other threads have reached this point in the code.
- Implicit barriers exist after work sharing constructs. The nowait clause can be used to prevent this behaviour.

Critical

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

Example: pushing and popping a task stack

```
!$OMP PARALLEL SHARED(STACK),PRIVATE(INEXT,INEW)
      ...
!$OMP CRITICAL (STACKPROT)
      inext = getnext(stack)
!$OMP END CRITICAL (STACKPROT)
      call work(inext,inew)
!$OMP CRITICAL (STACKPROT)
      if (inew .gt. 0) call putnew(inew,stack)
!$OMP END CRITICAL (STACKPROT)
      ...
!$OMP END PARALLEL
```

Atomic

- Atomic is a special case of a critical section that can be used for certain simple statements

Fortran: **`!$OMP ATOMIC`**

statement

where *statement* must have one of these forms:

x = x op expr, x = expr op x, x = intr (x, expr) or

x = intr (expr, x)

op is one of **+**, *****, **-**, **/**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**

intr is one of **MAX**, **MIN**, **IAND**, **IOR** or **IEOR**

Non Parallelizzabile

- Show an example of Instruction dependency

HANDS-ON ON THREADED LIBRARIES

- Source: http://www.llnl.gov/computing/tutorials/openMP/#ProgrammingModel

1) Create the submission script

3 fundamental steps for job execution

```
[hpc01@java2 ~]$ more sub_script.sh
```

```
#!/bin/bash
#PBS -q mhpc
#PBS -l nodes=1:ppn=16
#PBS -l walltime=2:00:00

cd $PBS_O_WORKDIR

export OMP_NUM_THREADS=8

#for Intel Compiler

module load intel/2013

./my_prog.x
```

2) Submit the script

```
[hpc01@java2 ~]$ qsub sub_script.sh
70.java2.grid.lipi.go.id
```

3) Monitor job execution

```
[hpc01@java2 ~]$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
70.java2                   test_pbs          ictp                   0 R nogpu
```