OpenMulti-Processing

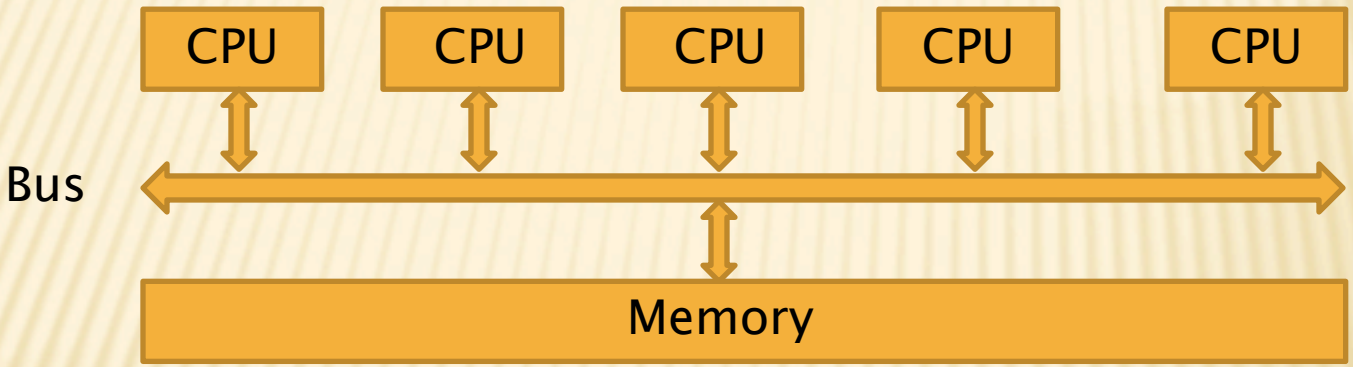
Message Passing Interface

Scientific Programming

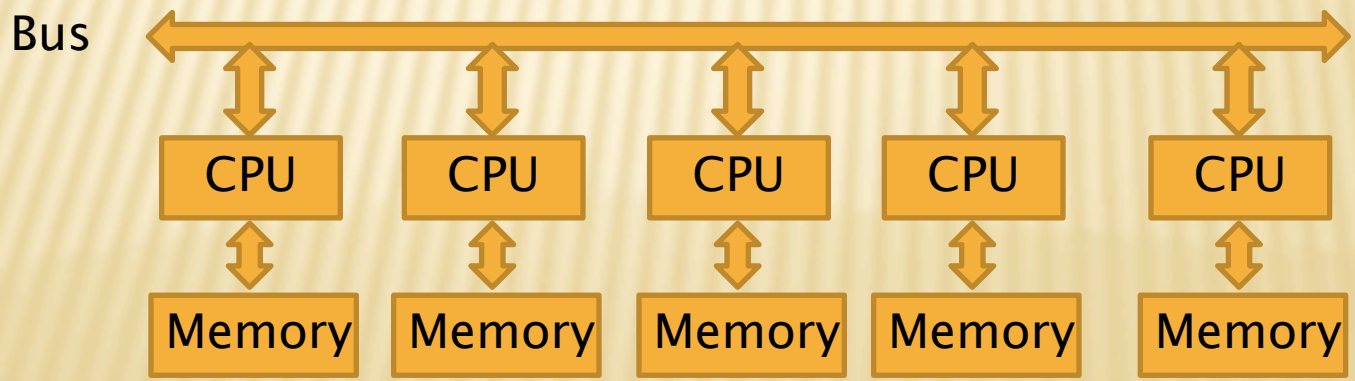
LECTURE 8: PARALLEL COMPUTING

PARALLEL COMPUTER ARCHITECTURE

- Shared memory machines: all processors are connected to each other and to the whole memory



- Distributed memory machines: memory is divided between (groups of) processors which are still interconnected



TWO PARALLELIZATION CONCEPTS

1. Totally independent programs that will run on separate processors and communicate with each other.

2. Parts of a single code that can be executed in arbitrary order. The compiler is instructed to create parallel threads for such parts.

INDEPENDENT PROGRAMS

- Works equally well on shared memory and on distributed memory computers
- Offers full control: you can decide when to run in parallel and when serial, synchronize, etc.
- The most popular incarnation is Message Passing Interface or **MPI** (look it up in Wikipedia)

MPI: HOW IT WORKS

- MPI is a library of functions
- Using MPI functions each program (referred to as process) can:
  - Figure out the total number of processes and its own serial number.
  - Send data to another process.
  - Wait for a data to arrive from another process.
  - Send data to all processes (this implies receive)
  - Add ID to each message so that one message is not taken for another.
  - Impose synchronization: street lights, barriers or simply waiting until data you sent is received.
  - Decide that a certain part of work (e.g. input/output) is done by one process only (convention is that process number 0 is the main process).

EXAMPLE:

```
IMPLICIT NONE
INCLUDE 'mpif.h'
INTEGER IERR,MY_PROC,N_PROC,MAIN_PROC
...
CALL MPI_INIT(IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, N_PROC,IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MY_PROC, IERR)
MAIN_PROC=0
if(MY_PROC.EQ.MAIN_PROC) then
  open(1,file='disk_model.dat',status='OLD',form='UNFORMATTED')
  read(1) (((rho(iX,iY,iZ),iX=1,nX),iY=1,nY),iZ=1,nZ)
  close(1)
endif
CALL MPI_BCAST(rho,ntot,MPI_REAL,MAIN_PROC,MPI_COMM_WORLD,IERR)
do i=MY_PROC+1,ntot,N_PROC
  ...
  acc=rho(kX(i),kY(i),kZ(i))*volume/dist_cube
  ...
enddo
CALL MPI_REDUCE(acc,gX,ntot,MPI_REAL,MPI_SUM, ...
...
CALL MPI_FINALIZE(IERR)
end
```

MPI: BASIC

- The simplest thing is to write a single program that decides what it is supposed to do based on the total number of processes and on the process number assigned to it
- Now we can run a bunch of copies of this program
- To the OS they are still a single program!
- Normally when you install MPI you get special scripts for compilation and running:
  - mpif90 -o mpicode mpicode.f90
  - mpiexec -n 10 ./mpicode

MPI: MORE ADVANCED

- An MPI code can send information and wait until it is received but it can also send and not wait (non-blocking send)
- Alternatively one can issue a receive statement but not wait!
- This allows automatic load balancing:
  1. Send work order to one process
  2. Issue a non-blocking “receive” and continue with the next process until all are busy
  3. Wait until any response
  4. Receive the result and send the next work order

MPI: MASTER-WORKER SCHEME

Master

- I/O operations.
- Distribute data to workers.
- Collect results.

Worker

- Receive data from master.
- Carry-out the computations.
- Send results to master.

MPI: MASTER-WORKER

- Master-Worker(s) requires having two different codes.
- Master: performs the input, initialization and then distributing the tasks between “workers”.
- Worker: wait for initialization then wait for task, complete it, send back the results and wait for more.
- In the end Master must inform Workers that job is done and they can finish the execution.

MPI: SUPER-ADVANCED

MPI-2 (2009) adds:

- one-sided communication (one process can read or write to the variables that belong to another process).
- Establish a new MPI-communication space or join into existing one.
- The behavior of I/O is put under MPI control, that is additional sync allow to order read/write operations coming from different processes.

MPI: MORE INFO

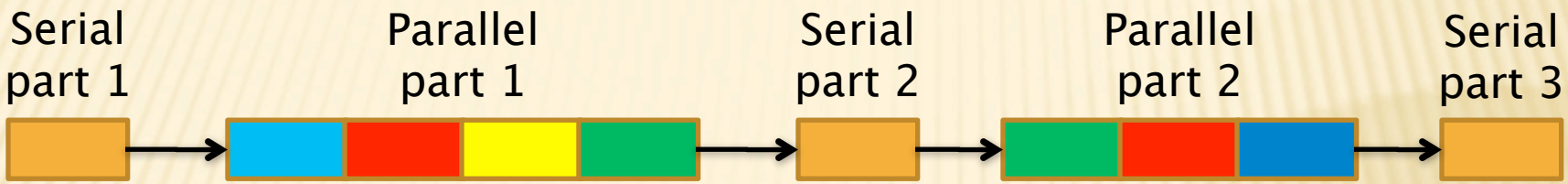
… can be found at:

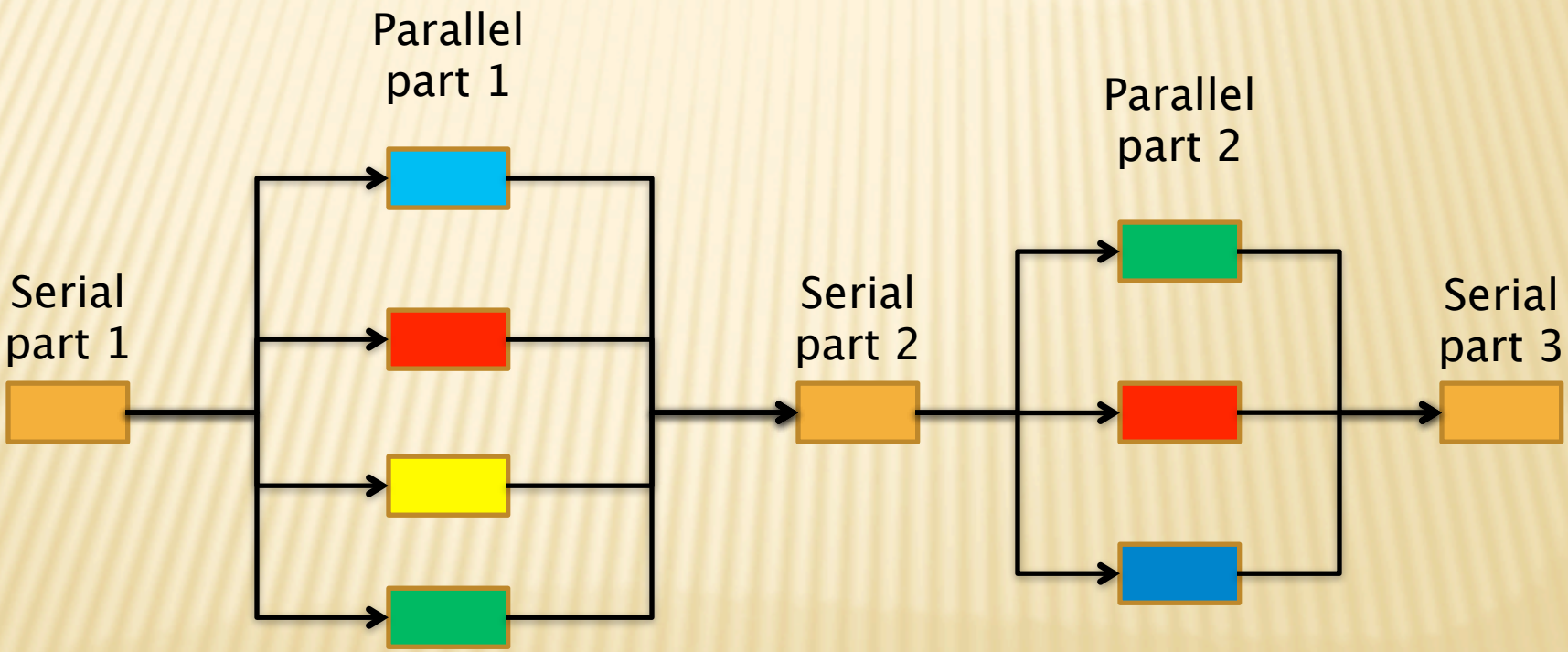
- http://www.mpi-forum.org/
- http://www.mcs.anl.gov/research/projects/mpi/
- https://computing.llnl.gov/tutorials/mpi/
- http://www.mcs.anl.gov/research/projects/mpi/tutorial/gropp/talk.html

MULTI-THREADING: OPENMP

- Open Multi-Processing (OpenMP) is a set of specifications (or interface) for the compilers capable of creating multi-threaded applications.
- The programmer gets a possibility to explain the compiler what parts of the code can be split in separate threads and run in parallel.
- The compiler does not have to guess – it relies on the hints contained in the OpenMP directives.
- Multi-threaded codes see the same variables and memory space: OpenMP is for shared memory computers!

OPENMP CONCEPT





OMP: BASIC

- The hints for the compiler are given as pre-processor statements.

C, C++

#pragma omp …

FORTRAN

C$omp …** or **!$omp …

- Most of statements contain indications for starting/finishing multi-threaded section

```
program hellof90
  use omp_lib
  integer:: id, nthreads
!$omp parallel private(id)
  id = omp_get_thread_num()
  write (*,*) 'Hello World '// &
     'from thread', id
!$omp barrier
  if(id == 0) then
    nthreads= omp_get_num_threads()
    write (*,*) 'There are', &
       nthreads, 'threads'
  end if
!$omp end parallel
end program hellof90
```

OMP: ADVANCED

- In **parallel** statements one can specify variables that must be **private** for each thread.
- The loop variable that is scheduled for parallel execution is automatically made private.
- Sometimes it is easier to declare certain sections of the code as **critical**. This means that although it will be computed by several threads, it will be done sequentially.
- OMP has even more sophisticated tools: barriers, reduction, conditional parallelization etc.
- OMP directives are not 100% obligatory for the compiler, they are just hints!

MORE INFO ON OPENMP

… can be found here:

- http://www.openmp.org
- https://computing.llnl.gov/tutorials/openMP
- http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11387
- http://www.clusterconnection.com/2009/08/comparing-mpi-and-openmp/

COMPARISON

MPI

- Runs on shared and distributed memory computers.
- Program behavior and performance are fully predictable.
- Very flexible.

- Needs modifications of the source code (will not run without MPI installed).
- All message passing requires a separate buffer (two in distributed memory system).
- Heavily depends on the latency of interconnection.

OpenMP

- Does not require code modification and is fully portable.
- Has minimum overhead in terms of memory.

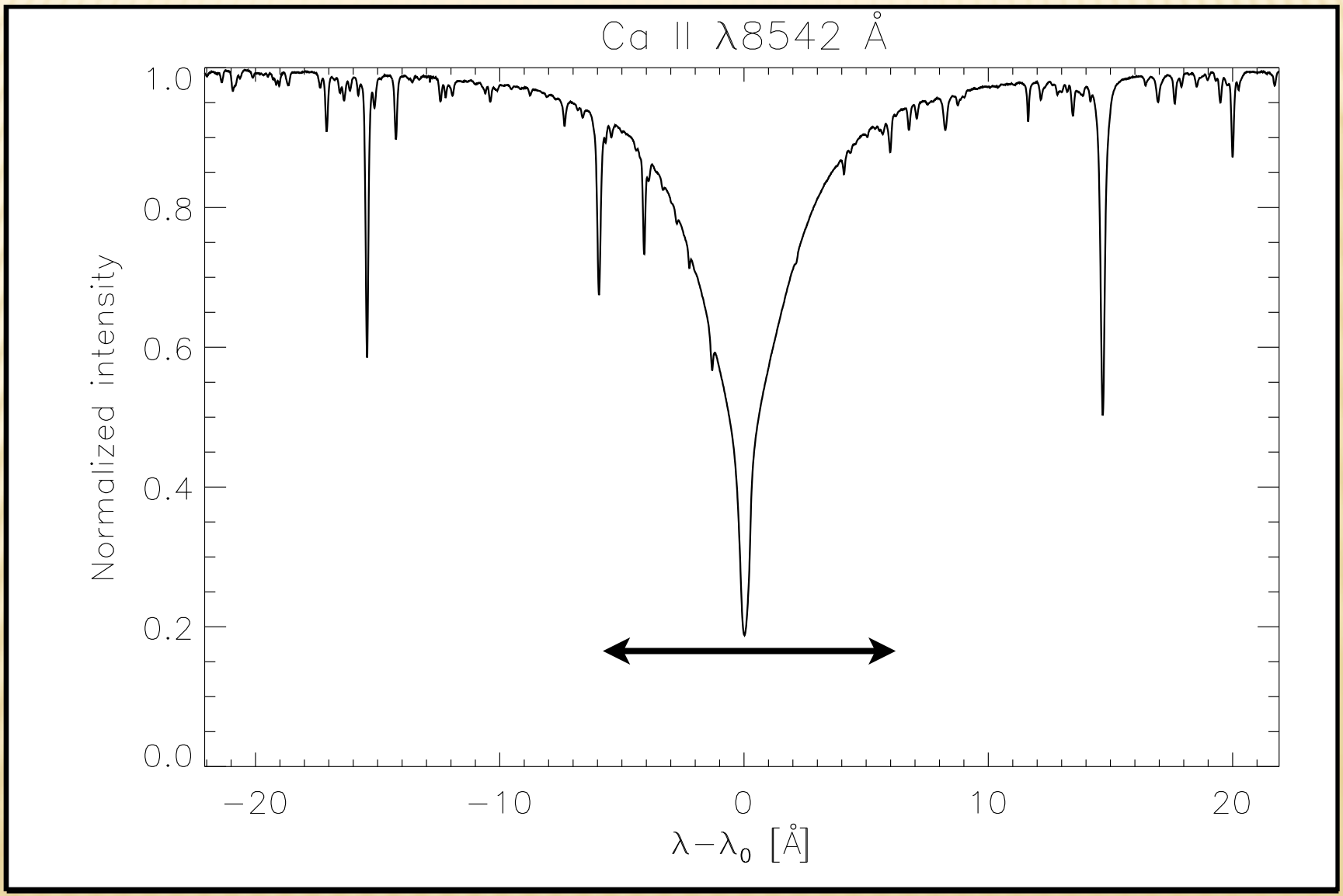
- Primarily aimed at parallelization of loops.
- Only runs on shared memory machines.
- No standard for I/O parallelization.

FINAL NOTE

- Nothing prevents you from combing MPI and OpenMP
- Most of modern computing clusters consists of distributed memory machines with fast interconnect (up 100 Gbyte/second!)
- Each processor is normally a multi-core CPU
- UPPMAX Isis: 200 IBM x3455 compute nodes with two dual- and quad-core AMD CPU in each.

- Test programs ...

SHIFT SPECTRUM (INTERPOLATING)



EXAMPLES STRUCTURE

- Read data from ca8542_satlas.txt.
- Propagate data to all processes.
- Apply a random shift to the wavelength array and interpolate to the new grid.
- Interpolation is carried-out with a Hermitian spline (intep.f90, mmath.h).
- Collect results.

EXAMPLES – TEST SCALABILITY

- cd ~/scientific_prog/{c++, fortran90}

- ./compile.sh
- ./ex1 (openmp parallel loop example)
- mpiexec -n # ./ex2 (MPI parallel loop)
- mpiexec -n # ./ex3 (MPI master-worker)