

Parallel Computing

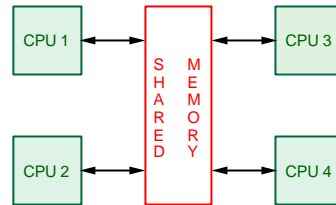
RGP I - week 3

Overview

- ◆ Parallel architectures
- ◆ Parallel programming styles
- ◆ Message Passing
- ◆ Message Passing libraries
- ◆ Deadlocks
- ◆ Communication styles
- ◆ Global communication
- ◆ Performance issues

Shared memory architectures

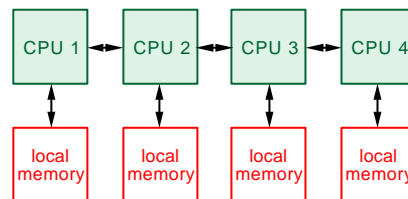
- ◆ share a common main memory
- ◆ are easy to program since all CPUs access the same data



- ◆ Disadvantages
 - ◆ scales well only to about 32 CPUs
 - ◆ concurrent access to memory is a problem
 - ◆ on PCs and workstations:
 - ◆ all CPUs share a path to the memory
 - ◆ one CPU that accesses the memory blocks all others
 - ◆ on vector computers like Crays, etc:
 - ◆ all CPUs have a full path to the memory
 - ◆ no interference between CPUs!
 - ◆ Fast memory is the main reason for the high price of a Cray!

Distributed memory architectures

- ◆ each CPU has access only to its local memory
- ◆ access to data of other CPUs only by communicating with these CPUs



- ◆ Disadvantages
 - ◆ access to remote memory is slow
 - ◆ harder to program efficiently
- ◆ Advantage
 - ◆ much cheaper

Coarse Grain Parallelism

- ◆ Parallelization can occur at many levels
- ◆ Coarse grain parallelization is simply running several independent programs on different CPUs
- ◆ Can be used to simulate many different parameter sets like
 - ◆ temperatures
 - ◆ system sizes
- ◆ This is very common in physics
- ◆ We just need an efficient queuing system

Medium Grain Parallelism

- ◆ For big problems we want to parallelize one program
- ◆ Medium grain parallelism makes use of the fact that some routines can be performed independently
- ◆ Examples from the Penna model:
 - ◆ Removing dead animals and creating new children are independent and could be performed at the same time on two different CPUs
 - ◆ The population could be split into several groups and each simulated independently on different CPUs
- ◆ This needs some extra programming work. For Monte Carlo simulations we have written a library which does this parallelization automatically

Fine Grain Parallelism

- ◆ In order to scale to many hundreds of CPUs often fine grain parallelism, within one function, is needed
- ◆ E.g. a loop


```
for (int j=0; j<N; ++j)
    a[j]=b[j]+c[j];
```
- ◆ could be split over M CPUs, each performing the summation on $1/M$ -th of the vectors
- ◆ This can sometimes be done automatically
 - ◆ in simple for loops
 - ◆ on shared memory machines
- ◆ High Performance Fortran can do this automatically for such simple loops even on shared memory machines
- ◆ In C++ libraries that can do this can be developed
 - ◆ Example: POOMA-2

Types of architectures

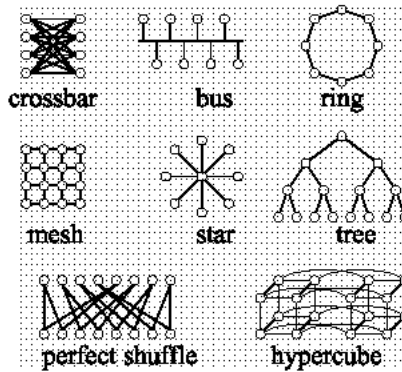
- ◆ SISD
 - ◆ single instruction - single data: an ordinary scalar CPU
- ◆ SIMD
 - ◆ single instruction - multiple data
 - ◆ all CPUs perform exactly the same operation on different data
 - ◆ was often used in the first parallel machines, now uncommon
 - ◆ also used inside vector CPUs
- ◆ SPMD
 - ◆ single procedure (program) - multiple data
 - ◆ all CPUs run the same program
- ◆ MIMD
 - ◆ multiple instruction - multiple data
 - ◆ nowadays the most common type - all CPUs can run independently, doing different tasks

Parallel machines

- ◆ SIMD style
 - ◆ MasPar, Thinking Machines 1 and 2, now outdated
- ◆ All-cache machines
 - ◆ Kendal Square (KSR), now bankrupt
 - ◆ only local cache memory, simulates shared memory
- ◆ MIMD machines
 - ◆ Cray T3E, IBM SP2, Hitachi SR8000, ASCI Blue, Red and White
 - ◆ achieve more than 1 Teraflop performance!
 - ◆ fastest machines on the world
- ◆ Beowulf clusters
 - ◆ clusters of PCs running Extreme Linux
 - ◆ best price-performance ratio
 - ◆ pioneered by physicists at NASA, Los Alamos, Sandia, ...
 - ◆ 500-CPU cluster is installed at ETH

Network topologies

- ◆ all-to-all:
 - ◆ needs $N(N-1)/2$ connections, but fastest communication
- ◆ Star
 - ◆ used often in Beowulf clusters, nodes connected to Ethernet hub
- ◆ Ring
 - ◆ appropriate for some problems
- ◆ Hypercube
 - ◆ nodes on edges of hypercube



Automatic parallelization

- ◆ High Performance Fortran (HPF) can convert programs with long loops automatically into SIMD style parallel programs
Some extensions to C++ can do the same
- ◆ However C++ allows a more flexible approach:
- ◆ Parallelizing class libraries
 - ◆ for vectors, matrices, ...
 - ◆ for MC simulations
 - ◆ etc.
- ◆ We have several research projects to develop parallel class libraries for physics simulations

Message Passing

- ◆ Without automatic parallelization we need to program the communication between CPUs (also called nodes)
- ◆ This is called **message passing**
- ◆ Several libraries exist for message passing
 - ◆ PVM (Parallel virtual machine)
 - ◆ originally developed for workstation clusters
 - ◆ available on all major architectures
 - ◆ most flexible library for clusters
 - ◆ MPI (Message Passing Interface) Standard
 - ◆ the new official ISO standard, adopted by all vendors on all modern machines
 - ◆ Older, vendor-specific libraries
- ◆ All libraries are very similar to use

What is a message?

- ◆ A message is a block of data sent by one node to another
- ◆ It usually consists of
 - ◆ pointer to buffer containing data
 - ◆ length of data in the buffer
 - ◆ a message tag, usually an integer identifying the type of message
 - ◆ number of the destination node(s)
 - ◆ number of the sender node
 - ◆ optionally a data type
- ◆ The message is passed through the network from the sender to the receiving node

Sending and receiving a message

- ◆ A SPMD “Hello World” program
 - ◆ node 1 sends a string with tag 99 to node 0
 - ◆ node 0 receives a string with tag 99 from node 1 and prints it
- ◆ The program:

```

#include <iostream>
#include <string>
#include <mpi.h>
int main(int argc, char**
  argv) {
  MPI_Init(&argc, &argv);
  int num;
  MPI_Comm_rank(
    MPI_COMM_WORLD, &num);
  if(num==0) {
    // master
    MPI_Status status;
    char txt[100];
    MPI_Recv(txt,100,MPI_CHAR,
      1,99,MPI_COMM_WORLD, &status);
    std::cout << txt << "\n";
  }
  else {
    // slave
    std::string text="Hello world!";
    MPI_Send(text.c_str(),
      text.size(),
      MPI_CHAR,0,99,
      MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}

```

The MPI standard

- ◆ We have seen several functions
 - ◆ MPI_Init
 - ◆ MPI_Finalize
 - ◆ MPI_Comm_rank
 - ◆ MPI_Send
 - ◆ MPI_Recv
- ◆ detailed explanations are available in the MPI manuals on www.mpi-forum.org
- ◆ other message passing libraries have similar functions

MPI_Send and MPI_Recv

- ◆ `int MPI_Send(void* buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm);`
 - ◆ `buf` ... buffer containing data
 - ◆ `count` ... number of elements
 - ◆ `type` ... datatype (MPI_BYTE is raw data)
 - ◆ `dest` ... destination number
 - ◆ `tag` ... message tag
 - ◆ `comm` ... communicator, MPI_COMM_WORLD is default
- ◆ `int MPI_Recv(void* buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status* status)`
 - ◆ `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are wildcards
 - ◆ `count` ... size of buffer available for message
 - ◆ `status` ... returns information on the message

MPI_Probe and MPI_Iprobe

- ◆ can be used to wait or check for a message
 - ◆ `int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)`
 - ◆ `int MPI_Iprobe(int source, int tag, MPI_Comm comm, int* flag, MPI_Status *status)`
- ◆ `MPI_Probe` waits for a message, `MPI_Iprobe` checks for one
- ◆ `flag` indicates if a message is there
- ◆ `status` can be queried about the message
 - ◆ `status.MPI_SOURCE` ... gets the source process
 - ◆ `status.MPI_TAG` ... gets the message tag
 - ◆ `status.MPI_ERROR`
 - ◆ `int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int* count)` ... gets the number of elements
- ◆ can be used to get size of unknown message before receiving it

Deadlocks

- ◆ Consider synchronous communication:
 - ◆ node 0:


```
MPI_Ssend(buf1, count, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD)
MPI_Srecv(buf2, count, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, status)
```
 - ◆ node 1:


```
MPI_Ssend(buf1, count, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD)
MPI_Srecv(buf2, count, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, status)
```
 - ◆ will deadlock as both wait for reception of message
- ◆ Solution:
 - ◆ node 0:


```
MPI_Srecv(buf2, count, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, status)
MPI_Ssend(buf1, count, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD)
```
 - ◆ node 1:


```
MPI_Ssend(buf1, count, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD)
MPI_Srecv(buf2, count, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, status)
```
- ◆ Check for this in your code!

Blocking communication types

- ◆ Synchronous send `MPI_Ssend`
 - ◆ returns only after recipient has started to receive
- ◆ Buffered send `MPI_Bsend`
 - ◆ makes a copy of buffer and returns once delivery is possible, can be before actual receipt, can be asynchronous
- ◆ Standard blocking send `MPI_Send`
 - ◆ either buffered or synchronous, decided by MPI
- ◆ Ready send `MPI_Rsend`
 - ◆ must be called only after other node has posted receive
 - ◆ optimized version!
- ◆ All these return only once the buffer can be reused
- ◆ Blocking receive `MPI_Recv`
 - ◆ returns only after message has been received

Nonblocking communication types

- ◆ are nonblocking, i.e. return before buffer can be reused
can be used to overlay communication and computation
 - ◆ `MPI_Issend`
 - ◆ `MPI_Ibsend`
 - ◆ `MPI_Isend`
 - ◆ `MPI_Irsend`
 - ◆ must be called only after other node has posted receive
 - ◆ optimized version!
 - ◆ `MPI_Irecv` also does not wait for completion
- ◆ `MPI_Test` checks for completion
- ◆ `MPI_Wait` waits for completion
- ◆ `MPI_Cancel` cancels request
- ◆ Compare
 - ◆ Blocking vs. nonblocking
 - ◆ Synchronous vs. asynchronous

Collective Communication

- ◆ Communication between many processes can be optimized
 - ◆ simple form of broadcast
 - ◆ step 1: 0 -> 1
 - ◆ step 2: 0 -> 2
 - ◆ ...
 - ◆ step N-1: 0 -> N
 - ◆ optimized broadcast
 - ◆ step 1: 0 -> 1
 - ◆ step 2: 0 -> 2, 1 -> 3
 - ◆ step 3: 0 -> 4, 1 -> 5, 2 -> 6, 3 -> 7
 - ◆ step 4: 0 -> 8, 1 -> 9, 2 -> 10, 3 -> 11, 4 -> 12, 5 -> 13, 6 -> 14, ...
- ◆ Optimized version in $\log_2(N)$ instead of N steps!

Types of collective communication

- ◆ **Broadcast** sends same data to all nodes
- ◆ **Scatter / Gather**
 - ◆ scatter: caller sends n-th portion of data to n-th node
 - ◆ gather: caller receives n-th portion of data from n-th node
- ◆ **All-gather**
 - ◆ everyone receives n-th portion of data from n-th node
- ◆ **All-to-all**
 - ◆ n-th node sends k-th portion to node k and receives n-th portion from node k; like a matrix transpose
- ◆ **Reduce**
 - ◆ combines gather with operation (e.g. sum all portions)
- ◆ **All-reduce, Reduce-scatter, ...**
- ◆ **Barrier**: waits for all nodes to call it; for synchronization

One-way communication

- ◆ a normal communication needs handshake
 - ◆ sender requests to send
 - ◆ recipient agrees to accept
 - ◆ sender sends data
- ◆ *This needs three one-way messages!*
- ◆ Remote Memory Access (RMA) allows one processor to directly write/read another's memory through messages
 - ◆ implemented by Cray on the T3D and T3E
 - ◆ included in the MPI-2 standard
- ◆ only useful on special hardware

SPMD style

- ◆ All nodes execute the same program
- ◆ Example: Integration of a function f over $[a,b]$ on N nodes

```
int main(int argc, char** argv) {
  // do some initialization
  ...
  // find interval for this node
  int num, total;
  MPI_Comm_size(MPI_COMM_WORLD, &total);
  MPI_Rank(MPI_COMM_WORLD, &num)
  double interval=(b-a)/total;
  double start=a+interval*num
  double end=start+interval
  integrate(f, start, end, steps/num);
  ... // collect results, print them and quit
}
```

Master - Slave style

- ◆ One node, the Master distributes tasks
- ◆ Other nodes (slaves) ask for tasks and perform them

```
int main(int argc, char**
  argv) {
  ... // initialize
  int num;
  MPI_Rank(MPI_COMM_WORLD,
           &num);
  if(num==0) master();
  else slave();
```

```
void master() {
  ... // find tasks and
      // distribute them
}
```

```
void slave() {
  ... // ask master for tasks
      // and perform them
}
```

- ◆ Master and slave can run different programs!

Scaling with node number

- ◆ The scalar part will dominate the CPU time!
 - ◆ Assume N nodes
 - ◆ on one node: $T_1 = T_{\text{scalar}} + T_{\text{parallel}}$
 - ◆ on N nodes: $T_N = T_{\text{scalar}} + T_{\text{parallel}}/N + T_{\text{communication}}(N)$
 - ◆ define scalar ratio $s = T_{\text{scalar}}/T_1$
- ◆ Reduce scalar parts
 - ◆ The optimum speedup would be $T_1/T_N < N / (1+s(N-1)) < 1/s$
 - ◆ *even if 1% is scalar it does not scale well beyond 100 nodes! ASCI machines have >10000 nodes!*
- ◆ Reduce communication time
 - ◆ Try to keep $T_{\text{communication}}$ as small as possible
 - ◆ Overlay communication with computation
- ◆ Make a plot of the speedup vs. N for your program!

Debugging a parallel program

- ◆ is very hard
- ◆ main problem are deadlocks
- ◆ some graphical tools exist:
 - ◆ xpvm
 - ◆ xmpi
- ◆ can help to understand what is going on

- ◆ Hints
 - ◆ first write a working serial program
 - ◆ Parallelize it and run it one one node first
 - ◆ two nodes next
 - ◆ ...
- ◆ **Good luck!!!**

Parallelizing the heat equation

- ◆ Will be done by you as an exercise

- ◆ Bonus exercise for interested students:
 - ◆ Write a scalar heat flow program
 - ◆ Implement the time evolution as an operator on a vector of data
 - ◆ Then provide a parallel evolution operator and vector class
 - ◆ Advantages:
 - ◆ Program identical for scalar and parallel versions
 - ◆ The vector class can be used in many other contexts
 - ◆ Can you think of mesh operators that might be useful for other partial differential equations?

- ◆ Problems like this are important current research topics

Tera (Cray MTA)

- ◆ is a new interesting but not much tested architecture
 - ◆ has 128 copies of all registers in CPU
 - ◆ this allows context switches between 128 threads without cost of CPU time
- ◆ allows memory and disk access within one cycle of CPU time!
 - ◆ this is simulated by running another thread in the waiting time
 - ◆ needs many threads on the CPU
- ◆ how to parallelize a program for the Tera
 - ◆ just split it into many independent small tasks
 - ◆ easy because of shared memory
 - ◆ the CPU will do the rest
- ◆ more information on www.cray.com