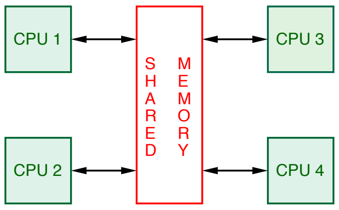
An Introduction to Parallel Computing

Shared memory architectures

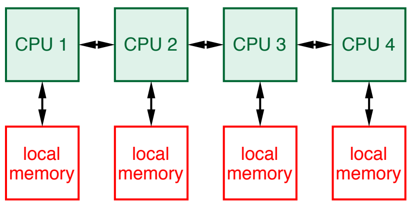
- share a common main memory
- are easy to program since all CPUs access the same data



- Disadvantages
  - scales well only to about 32 CPUs
  - concurrent access to memory is a problem
  - all CPUs share a path to the memory
  - one CPU that accesses the memory blocks all others but each has its own cache

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 much cheaper

Types of architectures

- SISD
  - single instruction - single data: an ordinary serial 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
  - But coming back in SSE units, graphics cards, …
- 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
  - Old machines: MasPar, Thinking Machines 1 and 2
- Massively parallel machines
  - IBM BlueGene, Cray XT5, SGI Altix
  - achieve more than 1 petaflop performance!
  - fastest machines on the world
- Clusters
  - clusters of PCs running Linux, best price-performance ratio
  - pioneered by physicists at NASA, Los Alamos, Sandia, …
  - 7000-CPU Brutus cluster is available at ETH

Network topologies

- all-to-all:
  - needs N(N-1)/2 connections, but fastest communication
- Hypercube
  - nodes on edges of hypercube, N $\log_2 N$ connections
- 3D crossbar
  - nodes on cube, 6N connections, used in Cray XT3, IBM BlueGene
- 2D crossbar
  - nodes on square, 4N connections, used in Hitachi supercomputers
- Ring
  - 2N connections, slow connection but appropriate for some problems
- Star
  - used often in clusters, nodes connected to Ethernet hub

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
  - mutation rates
- 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
- This needs some extra programming work

Fine Grain Parallelism

- In order to scale to many hundreds of CPUs often fine grain parallelism, within one function, is needed
  - Example:

    ```
    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 by smart compilers
  - in simple for loops
  - on shared memory machines

- In C++ libraries that can do this can be developed

Message Passing on distributed memory architectures

- Without automatic parallelization we need to program the communication between CPUs (also called nodes)

- This is called **message passing**

- Vendor specific libraries have been replaced by the MPI standard

- If you know how to send Christmas greetings by postal mail you know all you need to know

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

- An 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(
  const_cast<char*>(text.c_str()),
    text.size()+1, MPI_CHAR,
    0,99, MPI_COMM_WORLD);
    }
   MPI_Finalize();
  return 0;
}
```

Running the example using OpenMPI

- Get the sources from the web page
- Use your machine's MPI installation or get the OpenMPI libraries from http://www.open-mpi.org
- Compile the program:
  - `mpiCC -o example1 example1.C`
- Run the program in parallel using 2 processes:
  - `mpirun -np 2 ./example1`

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_ptr)`
  - `int MPI_Iprobe(int source, int tag, MPI_Comm comm, int* flag, MPI_Status *status_pts)`
- **`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_ptr, MPI_Datatype datatype, int* count)` … gets the number of elements
- can be used to get size of unknown message before receiving it

Deadlocks: deadlock1.C, deadlock2.C

- Consider synchronous communication:
  - node 0:
    ```
    MPI_Ssend(&d,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    MPI_Recv(&d,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,&status);
    ```
  - node 1:
    ```
    MPI_Ssend(&d,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD);
    MPI_Recv(&d,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,&status);
    ```
  - will deadlock as both wait for reception of message
- Solution:
  - node 0:
    ```
    MPI_Recv(&d,count,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,&status);
    MPI_Ssend(&d,count,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    ```
  - node 1:
    ```
    MPI_Ssend(&d,count,MPI_DOUBLE,0,tag,MPI_COMM_WORLD);
    MPI_Recv(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 (small messages) or synchronous,
- All these return only once the data 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 on most massively parallel machines
  - included in the MPI-2 standard

- only useful on special hardware

SPMD style

- All nodes execute the same program: `example2.C`
- 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_Comm_rank(MPI_COMM_WORLD,&num)
double interval=(b-a)/total;
double start=a+interval*num;
double end=start+interval;
integrate(start,end,steps/total);
… // collect results, print them and quit
}
```

Master - Slave style

- One node, the Master distributes tasks: example3.C
- 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: Amdahl's law

- The sequential, non-parallel part will dominate the CPU time!
  - Assume *N* nodes
  - on one node: $T_1 = T_{serial}+T_{parallel}$
  - on N nodes: $T_N = T_{serial}+T_{parallel}/N+T_{communication}(N)$
  - define serial ratio $s= T_{serial}/T_1$
- Reduce serial parts
  - The optimum speedup would be $T_1/T_N< N / (1+s(N-1)) < 1/s$
  - *even if 1% is serial it does not scale well beyond 100 nodes! ASCI machines have >10000 nodes!*
- Reduce communication time
  - Try to keep $T_{communication}$ 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!!!**

OpenMP standard for shared memory architectures

- Home page: http://www.openmp.org
  - Contains the specification of the standard including many examples
- We will look at the C/C++ standard
- Semi-automatic parallelization using directives
  - A directive is written as a line before the statement or block of statements:

    ```
    #pragma omp directive
    ```
- Some auxilliary function calls

A first parallel example

- A simple loop is parallelized
  - Possible only if there are no dependencies

-
  ```
  #pragma omp parallel
  {  // each parallel region starts with this directive
  #pragma omp for
    for (i=1; i<n; i++)
      b[i] = (a[i] + a[i-1]) / 2.0;
  }
  ```
- Or the shortcut version for a single loop

  ```
  #pragma omp parallel for
  for (i=1; i<n; i++)
    b[i] = (a[i] + a[i-1]) / 2.0;
  ```

Two loops

- Two loops are parallelized

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=1; i<n; i++)
    b[i] = (a[i] + a[i-1]) / 2.0;
#pragma omp for nowait
  for (i=0; i<m; i++)
    y[i] = sqrt(z[i]);
}
```

- The `nowait` directive avoids the implied barrier at the end of the first loop. A thread may start on the second loop before the first is finished.

Loop parallelization schedules

- Several scheduling strategies can be specified

```
#pragma omp parallel for schedule(schedule)
for (i=1; i<n; i++)
  b[i] = (a[i] + a[i-1]) / 2.0;
```

- Static scheduling: `schedule(static,`*`chunksize`*`)`
  - Assigns each thread blocks of size chunksize at compile time
  - Useful if all iterations take the same amount of work and all threads are equally fast
- Static scheduling: `schedule(dynamic,`*`chunksize`*`)`
  - Assigns each thread blocks of size chunksize
  - Once a thread finishes a block it gets the next block to be done
  - Useful if all iterations take the same amount of work but not all threads are equally fast
- Static scheduling: `schedule(guided,`*`chunksize`*`)`
  - Assigns blocks of size at least chunksize
  - At first bigger blocks are used, later smaller ones to give optimal performance
  - Useful if neither thread speed nor work load are equal

Simple parallel regions

- Split the work

```
#pragma omp parallel shared(x, npoints) private(iam, np, ipoints)
{
  iam = omp_get_thread_num();
  np = omp_get_num_threads();
  ipoints = npoints / np;
  subdomain(x, iam, ipoints);
}
```

- `shared` specifies which variables are shared between the threads
- `private` specifies variables of which each thread has its own
- By default all variables except for loop counters in for loop are shared
- For more information, and `firstprivate`, `lastprivate`, `copyin` see the OpenMP specification

Auxilliary functions

- `omp_get_thread_num()` … returns the number of the current thread
- `omp_set_num_threads(int)` … sets the number of threads
- `omp_get_num_threads()` … returns the number of threads
- `omp_get_max_threads()` … returns the maximum number of threads
- `omp_get_num_procs()` … returns the number of processors used
- `omp_set_dynamic(bool)` … enables/disables automic adjustment of the number of threads
- `omp_get_dynamic()` … returns if automatic adjustment is allowed
- All these functions work only with OpenMP. To make the code portable use the following trick to e.g. enforce four threads if OpenMP is used:

```
#ifdef _OPENMP
omp_set_dynamic(false);
omp_set_num_threads(4);
#endif
```

Critical sections

- Some parts of code may be critical
  - Only one thread may enter it at any time
  - Example: assigning a new task

-
  ```
  #pragma omp parallel shared(x, y) private(x_next, y_next)
  {
    #pragma omp critical ( xaxis )
    x_next = dequeue(x);
    work(x_next);
    #pragma omp critical ( yaxis )
    y_next = dequeue(y);
    work(y_next);
  }
  ```
- Different critical sections may be distinguished by names
  - Each section with a certain name may be entered only by one thread at a time.
  - More than one section may have the same name: only one thread at a time may be in any section with the given name

Performing atomic updates

- We need to store results of calculations
  - No two threads should try to update the same location simultaneously
- Solution 1: make the writing critical: only one thread will ever write terribly slow, no speedup!

```
#pragma omp parallel for shared(sum)
for (i=0; i<n; i++) {
  #pragma omp critical
  sum+= f(i);

}
```

- Solution 2: make the writing atomic: no two threads will ever have the same value of `index[i]` simultaneously, much faster

```
 #pragma omp parallel for shared(x, y, index, n)
for (i=0; i<n; i++) {
  #pragma omp atomic
  x[index[i]] += work1(i);
  y[i] += work2(i);
}
```

Reductions

- Loops like the following may appear in an integration code

```
for (i=0; i<n; i++)
  sum += f(a+i*delta);
```

- This is one way to parallelize, using features we know

```
#pragma omp parallel shared (sum) private(partial)
{
  partial = 0.;
  #pragma omp for
  for (i=0; i<n; i++)
    partial += f(a+i*delta);

  #pragma omp atomic
  sum += partial;
}
```

- Or better, automatically using the reduction clause

```
#pragma omp parallel for reduction(+: sum)
for (i=0; i<n; i++)
  sum += f(a+i*delta);
```

Parallelizing macro tasks

- Consider three functions that can be executed simultaneously:

```
#pragma omp parallel sections
{
  #pragma omp section
  xaxis();
  #pragma omp section
  yaxis();
  #pragma omp section
  zaxis();
}
```

Statements executed only by a single thread

- Only a single thread should ever print

```
 #pragma omp parallel
{
  #pragma omp single
  std::cout << “Beginning work1.\n”;
  work1();
  #pragma omp single
  std::cout << Finishing work1.\n";
  #pragma omp single nowait
  std::cout << "Finished work1 and beginning work2.\n";
  work2();
}
```

Keeping the same order

- Consider a loop

```
for (i=lb; i<ub; i+=st)
  work(i);
```

- This function works but prints the number in arbitrary order

```
void work(int k)
{
  #pragma omp critical
  std::cout << k;
}
```

- The ordered pragma ensures to get the same order as in sequential execution

```
void work(int k)
{
  #pragma omp ordered
  std::cout << k;
}
```