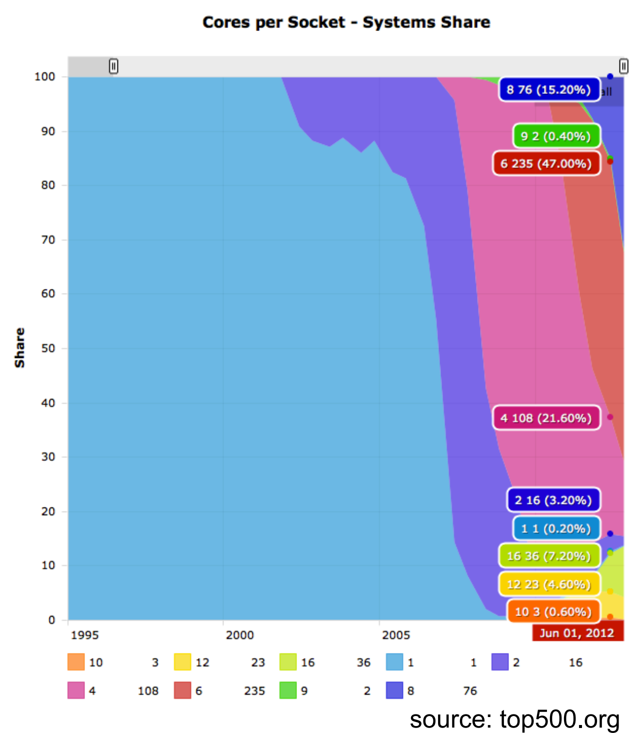
An Introduction to Parallel Computing

Parallel computing

- What is parallel computing?
- Where does one need parallel computing?
- Who needs parallel computing?
  - Application areas on top500.org



source: top500.org

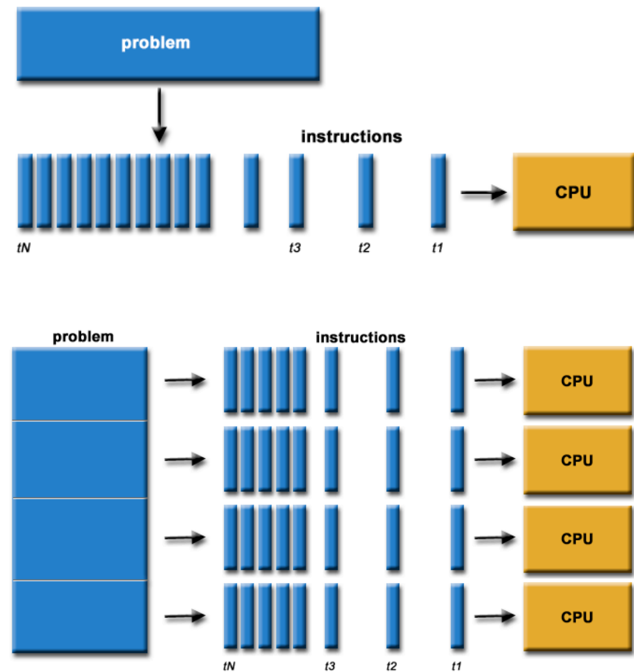
Let’s do parallel computing!

- Assume every seat in the room is a “computational core”
- How do we distribute N pieces of paper in parallel?
- How do we sum together numbers?

What were the key ideas?

The parallel idea

- Sequential:
  - Problem is split into pieces
  - Instructions are executed one after the other by the CPU
- Parallel:
  - Problem is split into many independents sequences
  - Many CPUs executes the sequences concurrently
- Not always easy to split the problem!



Types of architectures - taxonomy

- Taxonomy introduced M. Flynn in 1966 [Proc. IEEE, vol.54, no.12, pp. 1901- 1909, Dec. 1966]

data stream (columns) / instruction stream (rows)

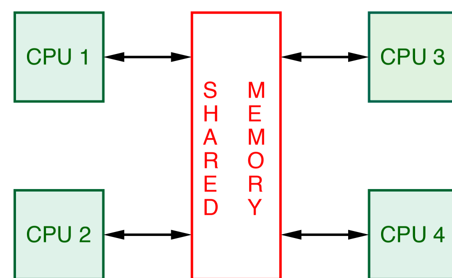
instruction stream	SISD	SIMD
	MISD	MIMD

- Today we cover parallel programing in terms of MIMD systems

Types of architectures - taxonomy

- 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
  - Altivec (PowerPC) and SSE (Pentium) are SIMD units
  - GPU are the new implementation of SIMD architectures
    - programming environment: CUDA, OpenCL
- MIMD
  - multiple instruction - multiple data
  - nowadays the most common type - all CPUs can run independently, doing different tasks

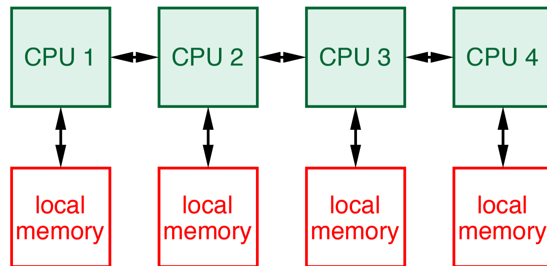
Shared memory architectures



- share a common main memory
- are easy to program since all CPUs access the same data
- Disadvantages
  - scales well only to about 48 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!

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

Parallel machines

- SIMD style
  - Old machines: MasPar, Thinking Machines 1 and 2
  - heterogeneous systems (CPU+GPU) appearing again in top500.org
    - Cray XK7, Tianhe-1A
- MIMD machines
  - Cray XE6, IBM BlueGene, Fujitsu K
  - achieve more than 10 Petaflops performance!
  - fastest machines on the world
- Beowulf clusters
  - clusters of PCs running Linux, best price-performance ratio
  - pioneered by physicists at NASA, Los Alamos, Sandia, …
  - 20’000-CPU cluster is available at ETH

ETH Brutus cluster

- Heterogeneous system with a total of 19’760 processor cores in ca. 1’000 compute nodes
  - Computation power of 200 Teraflops

- Shared memory
programming model on nodes
- Distributed memory
programming model across nodes

- Ranked the 88th fastest computer in the world in November 2009

CSCS - Swiss National Supercomputing Centre

CRAY XE6 – Rosa:

2’992 AMD 16-core Opteron @ 2.1 GHz --> 47’872 cores
46 TB DDR2 RAM
290 TB Disk
9.6 GB/s interconnect bandwidth

Computation power of 402 Teraflop/s

Cluster vs. Supercomputer

- What is the real difference?

Cluster vs. Supercomputer

Cray XE6

Cluster vs. Supercomputer

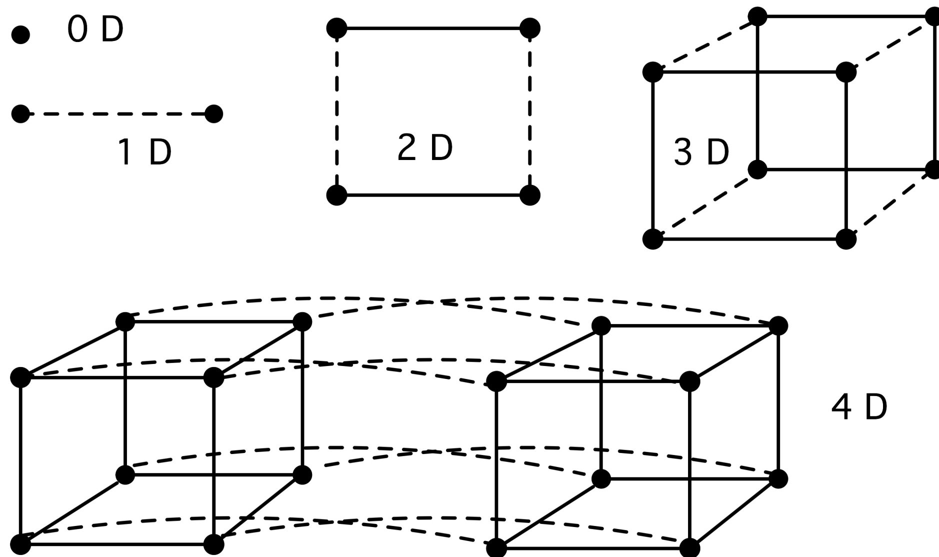
- Network is the main part of a Supercomputer!

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, IBM BlueGene
- 2D crossbar
  - nodes on square, 4N connections, used in older supercomputers
- Ring
  - 2N connections, slow connection but appropriate for some problems
- Star
  - used often in Beowulf clusters, nodes connected to a switch

Hypercube interconnect

- Network consists of $p = 2^d$ processors
  - example: 16 processors lies on the edges of a 4-dim hypercube



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
  - usually only in simple *for* loops,
  - and on shared memory machines
- In C++, libraries that can do this can be developed

Shared memory

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 auxiliary function calls

One additional line of code → perfect scaling

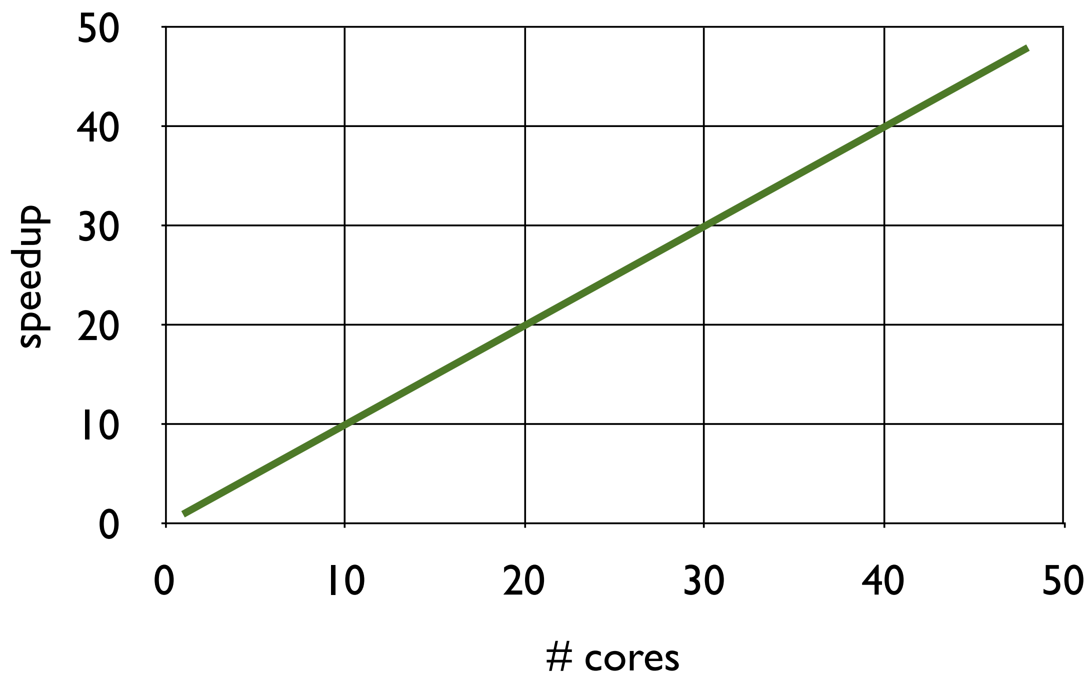
- Serial:

```
const std::size_t nsamples = 1E10;
double mean = 0.;
std::mt19937 rng(42);
mean = calcpi4(rng, nsamples/double(nthreads)+0.5);

double error = std::sqrt(1./(nsamples-1.) * (mean - mean*mean));
std::cout << "pi = " << 4*mean << " +/- " << 4*error << std::endl;
```

- Parallel:

```
const std::size_t nsamples = 1E10;
double mean = 0.;
#pragma omp parallel reduction(+:mean)
{
    std::mt19937 rng(rank);
    mean = calcpi4(rng, nsamples/double(nthreads)+0.5);
}
mean /= nthreads;

double error = std::sqrt(1./(nsamples-1.) * (mean - mean*mean));
std::cout << "pi = " << 4*mean << " +/- " << 4*error << std::endl;
```


One additional line of code → perfect scaling



Parallel region

```cpp
#include <iostream>
#include <omp.h>

int main()
{
  #pragma omp parallel
  {
    // now we execute this block in multiple threads
    std::cout << "I am thread " << omp_get_thread_num()
              << " of " << omp_get_num_threads() << " threads." << std::endl;
  }
}
```

- Threads are spawn at the beginning of the parallel block
- At the end of the parallel block, the code is again serial

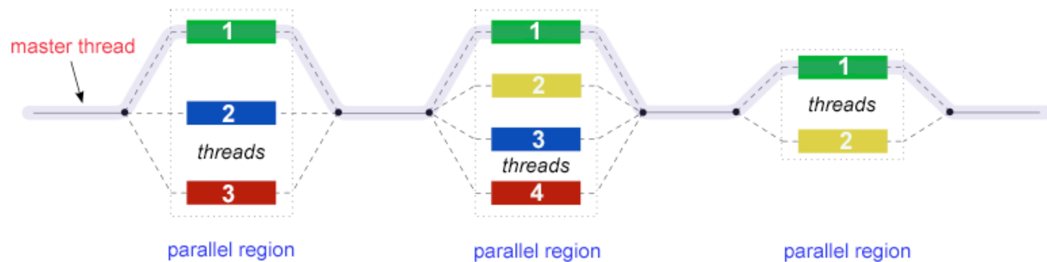


image from: https://computing.llnl.gov/tutorials/openMP

Parallel sum with OpenMP

- We want to perform the sum c[i] = a[i] + b[i] in parallel

```cpp
#include <iostream>
#include <omp.h>

int main()
{
    unsigned long const N = 100;
    std::vector<int> a(N, 1.), b(N, 1.5), c(N);

    for (std::size_t i = 0; i < N; ++i)
        c[i] = a[i] + b[i];
}
```

Parallel sum with OpenMP

- We want to perform the sum c[i] = a[i] + b[i] in parallel
  - all threads are now running the full loop

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

int main()
{
    unsigned long const N = 100;
    std::vector<int> a(N, 1.), b(N, 1.5), c(N);

#pragma omp parallel
    {
        for (std::size_t i = 0; i < N; ++i)
            c[i] = a[i] + b[i];
    }
}
```


Parallel sum with OpenMP

- We want to perform the sum c[i] = a[i] + b[i] in parallel
  - every thread work on different parts of the loop

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

int main()
{
    unsigned long const N = 100;
    std::vector<int> a(N, 1.), b(N, 1.5), c(N);

#pragma omp parallel
    {
        int t = omp_get_thread_num();
        int nthreads = omp_get_num_threads();

        long double const step = (nterms+0.5l) /  nthreads;
        int stop = (t+1) * step;
        for (std::size_t i = t * step; i < stop; ++i)
            c[i] = a[i] + b[i];
    }
}
```

Running a program with OpenMP

- Get sources from repository
- Compile the program
  - g++ -fopenmp hello1.cpp -o hello1
- Run the program (as usual)
  - ./hello1
  - it runs using the maximum number of threads
- The number of threads can be specified at run-time
  - export OMP_NUM_THREADS=4
  - ./hello1

Parallel sum with OpenMP

- Since loop parallelization is very common, there is an automatic shortcut

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

int main()
{
    unsigned long const N = 100;
    std::vector<int> a(N, 1.), b(N, 1.5), c(N);

#pragma omp parallel
    {
#pragma omp for
        for (std::size_t i = 0; i < N; ++i)
            c[i] = a[i] + b[i];
    }
}
```

Parallel sum with OpenMP

- Since loop parallelization is very common, there is an automatic shortcut
  - Even shorter!

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

int main()
{
    unsigned long const N = 100;
    std::vector<int> a(N, 1.), b(N, 1.5), c(N);

#pragma omp parallel for
    for (std::size_t i = 0; i < N; ++i)
        c[i] = a[i] + b[i];
}
```


DOT product with OpenMP

- Let's parallelize similar to the previous example.

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

int main()
{
    unsigned long const N = 100;
    std::vector<int> a(N, 2.), b(N, 1.5);

    double sum = 0.;
#pragma omp parallel for
    for (std::size_t i = 0; i < N; ++i)
        sum += a[i] * b[i];

    std::cout << "Dot product is" << sum << std::endl;
}
```

- What is the output?

Race condition

- Sequential execution

Thread 1		*i*=
		0
read *i*	←	0
increment value		0
write back *i*	→	1
read *i*	←	1
increment value		1
write back *i*	→	2

- Multithreaded execution
  - (one of the many)

Thread 1	Thread 2		*i*=
			0
read *i*		←	0
increment value			0
	read *i*	←	0
write back *i*		→	1
	increment value		1
	write back *i*	→	1

DOT product with OpenMP

- OpenMP **critical** sections are executed by one thread at a time
  - it solves race conditions
  - but it makes the code slower

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

int main()
{
    unsigned long const N = 100;
    std::vector<int> a(N, 2.), b(N, 1.5);

    double sum = 0.;
#pragma omp parallel for
    for (std::size_t i = 0; i < N; ++i)
#pragma omp critical
        sum += a[i] * b[i];

    std::cout << "Dot product is" << sum << std::endl;
}
```

DOT product with OpenMP

- The sum can be performed in parallel in $O(\log(N))$ complexity
  - OpenMP has a shortcut for it
    - `#pragma omp for reduction(operator: variable)`

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

int main()
{
    unsigned long const N = 100;
    std::vector<int> a(N, 2.), b(N, 1.5);

    double sum = 0.;
#pragma omp parallel for reduction(+:sum)
    for (std::size_t i = 0; i < N; ++i)
        sum += a[i] * b[i];

    std::cout << "Dot product is" << sum << std::endl;
}
```


Penna model with OpenMP

- How to parallelize the `Population::step()` function?

```
void Population::step()
{
    // Age all animals
    for_each( begin(), end(), mem_fun_ref(&Animal::grow) );
}
```

- Remember that with a Bidirectional iterator

```
std::list<Animal> population;
// init
...
typedef typename std::list<Animal>::iterator iterator;
iterator start = population.begin();
iterator it1   = ++start;                  // O(1)
iterator it2   = std::advance(start, n);   // O(n)
```

Penna model with OpenMP

- The usual OpenMP approach would perform similar to

```
std::size_t step = population.size() / omp_get_num_threads();
#pragma omp parallel
{
    std::size_t t = omp_get_thread_num();
    iterator it   = std::advance(population.begin(), step*t);
    iterator end  = std::advance(population.begin(), step*(t+1));
    for(; it!=end; ++it)
        it->grow();
}
```

- **Terribly slow!**

Penna model with OpenMP

- New idea, OpenMP tasks.

```
#pragma omp parallel
#pragma omp single nowait
    for(iterator it=population.begin(); it!= end(); ++it)
#pragma omp task
        it->grow();
```

- Tasks are lightweight objects that get pushed into a task queue, idle threads pull tasks from the queue
- Allow to parallelize irregular problems:
  - unbounded loops
  - recursive algorithms
  - producer/consumer schemes
  - ...

Overview of OpenMP directives

- `#pragma omp parallel`
- Optional clauses:

`if (scalar_expression)`	Only parallelize if the expression is true. Can be used to stop parallelization if the work is too little
`private (list)`	The specified variables are thread-private
`shared (list)`	The specified variables are shared among all threads
`default (shared | none)`	Unspecified variables are shared or not
`copyin (list)`	Initialize private variables from the master thread
`firstprivate (list)`	A combination of private and copyin
`reduction (operator: list)`	Perform a reduction on the thread-local variables and assign it to the master thread
`num_threads (integer-expression)`	Set the number of threads

- Example:

```
#pragma omp parallel private(i) shared (n) if (n>10)
{
    ...
}
```

More at http://www.openmp.org/mp-documents/OpenMP3.1.pdf

Overview of OpenMP directives

- `#pragma omp for`
- Optional clauses:

`nowait`	There is no implicit barrier at the end of the for. Useful, e.g. if there are two for loops in a parallel section.
`ordered`	The same ordering as in the serial code can be enforced
`collapse (n)`	collapse n nested loops into one and parallelize it
`schedule (type [,chunk])`	specify the schedule for loop parallelization (see below)

- Scheduling options:

STATIC	Loop iterations are divided into fixed chunks and assigned statically
DYNAMIC	Loop iterations are divided into fixed chunks and assigned dynamically whenever a thread finished with a chunk.
GUIDED	Like dynamic but with decreasing chunk sizes. The chunk parameter defines the minimum block size
RUNTIME	decide at runtime depending on the OMP_SCHEDULE environment variable
AUTO	decided by compiler and/or runtime system

More at http://www.openmp.org/mp-documents/OpenMP3.1.pdf

Overview of OpenMP auxilliary functions

- In header: `#include <omp.h>`
- `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
```

More at http://www.openmp.org/mp-documents/OpenMP3.1.pdf

OpenMP conclusions

- OpenMP is
  - simple threading on shared memory platforms
  - portable and standardized across many platforms and compilers
  - supporting C/C++ and Fortran
  - lean and ease, easy to use, augment code with compiler directives

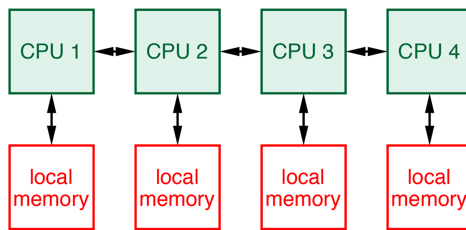
- OpenMP is easy to use but is **not**
  - checking for data dependencies, conflicts, race conditions, or deadlocks
  - giving you the best optimized code
  - implemented in the same way on all compilers

OpenMP is much more than this

- We didn't touch:
  - parallel sections
  - nested parallelism
  - synchronization (`barrier`, `nowait`, etc.)
  - worksharing constructs (`single`, `master`, etc.)
- More advanced lecture will have a more detailed view
  - High Performance Computing for Science and Engineering
- Learn from examples
  - http://www.openmp.org

Distributed memory

Message Passing on distributed memory architectures



- On distributed machined we need to program the communication between processes running on the 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

MPI standards

- Goals of the MPI standard:
  - portable, efficient, easy to use
  - works on distributed memory, shared memory and hybrid systems
- Versions of the MPI standard:
  - MPI-1 was first finished in 1992, minor updates over the years (1.1, 1.2, 1.3)
  - MPI-2 was first proposed 1998 and adds one-sided communication, I/O, and creation of processes
  - MPI-3 was finalized September 2012 and adds more features, in particular non-blocking collective communication
- We will cover mainly MPI-1 since that is what is needed for most codes

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 parallel “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

```
#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();
}
```

The structure of an MPI program

- Include the header <mpi.h>
- You need to initialize and terminate the MPI environment in your code.
- Note that you need to pass pointers to argc and argv. The MPI environment might grab some command line options and return a modified list of options.

```
#include <mpi.h>

int main(int argc, char** argv) {

  MPI_Init(&argc, &argv); // initialize the environment

  ... // do something

  MPI_Finalize();         // clean up at the end

  return 0;
}
```


Initialization and termination functions

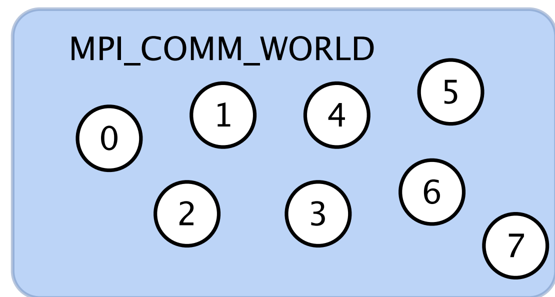
- You've seen two of the five functions connected with setting up the MPI environment.

```
int MPI_Init(int*argc, char***argv);
// initializes the environment

int MPI_Finalize()
// terminates the environment

int MPI_Abort( MPI_Comm comm, int errorcode );
// terminates all processes with the given error code

int MPI_Initialized( int *flag )
// sets the flag to true if MPI has been initialized

int MPI_Finalized( int *flag )
// sets the flag to true if MPI has been finalized
```

Obtaining the rank and size

- MPI numbers the processes inside **communicators**
- By default one communicator, **MPI_COMM_WORLD** is created containing all processes.

```
#include <iostream>
#include <mpi.h>

int main(int argc, char** argv) {

  MPI_Init(&argc, &argv);

  int rank;
  int size;

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  std::cout << "I am rank " << rank <<
               " of " << size << "." << std::endl;

  MPI_Finalize();

  return 0;
}
```



Running the example using Open MPI: D-PHYS machines

- Get the sources from the repository
- Compile the program:
  - `mpicxx -o hello1 hello1.cpp`
- Run the program in parallel using 4 processes:

```
$ mpirun -np 4 ./a.out
I am rank 1 of 4.
I am rank 2 of 4.
I am rank 0 of 4.
I am rank 3 of 4.
```

Things to do on your own machine

- Check if you already have a MPI installation
  - which mpicc
- In case you need to install it, try Open MPI
  - http://www.open-mpi.org
  - example of the installation is provided on the lecture homepage
- To run it in parallel on more than one machine
  - Setup automatic authentication
    - Use .rhosts with rsh
    - Use authorization keys with ssh (details on http://nic.phys.ethz.ch)
  - Prepare a file with the names of all PCs you want to use
  - Give that file as argument to mpirun
    - mpirun -hostfile <filename> (for Open MPI)

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

Probing for messages

- Instead of directly receiving you can probe whether a message has arrived:

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
// wait for a matching message to arrive

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
// check if a message has arrived.
// flag is nonzero if there is a message waiting

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int* count)
// gets the number of elements in the message waiting to be received
```

- The MPI_Status object can be queried for information about the message:

```
MPI_Status status;
int count;

// wait for a message
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, &status);
std::cout << "A message is waiting from " << status->MPI_SOURCE
          << "with tag "                  << status->MPI_TAG;

// get the element count
MPI_Get_count(&status, MPI_INT, &count)
std::cout << "and assuming it contains ints there are " << count << "elements";
```


Deadlocks: deadlock1.cpp, deadlock2.cpp

- Consider synchronous communication:
  - process 0:
    ```
    MPI_Ssend(&d,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    MPI_Recv(&d,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,&status);
    ```
  - process 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:
  - process 0:
    ```
    MPI_Recv(&d,count,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,&status);
    MPI_Ssend(&d,count,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    ```
  - process 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!

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

Scatter & Gather

- The scatter operation sends a different piece of data to each of the ranks
  - Example: take a vector and split it over the other ranks
- The gather operations collects data from the other ranks into a big buffer
  - Example: gathering pieces of a distributed vector into a big local one

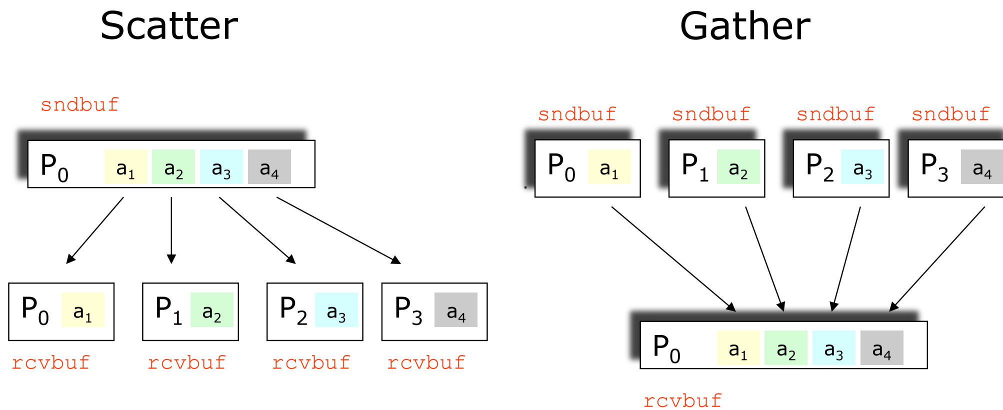


Image © CSCS

All-to-all

- **MPI_Alltoall**: n-th rank sends k-th portion of its data to rank k and receives n-th portion from process k.
  - Everyone scatters and gather at the same time
  - like a matrix transpose. Attention: slow!

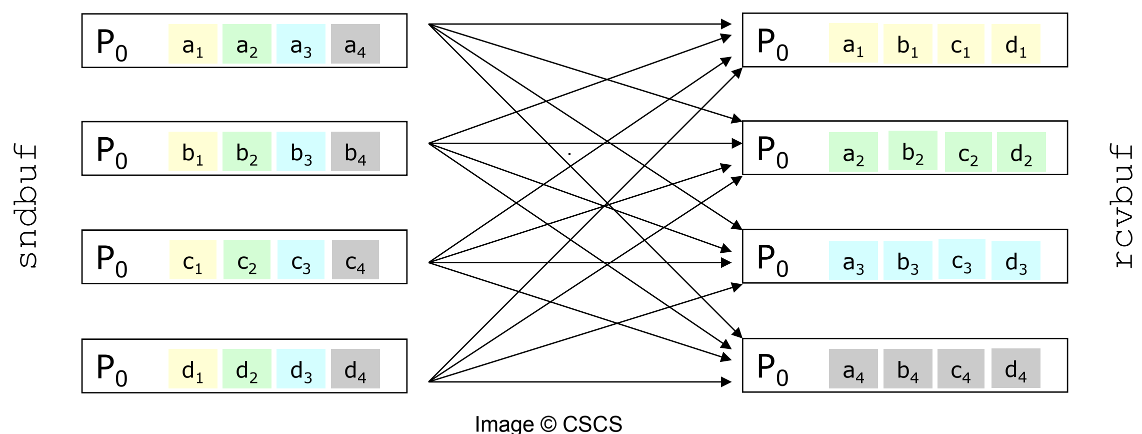


Image © CSCS

SPMD style

- All processes execute the same program: `integrate1.cpp`
- Example: Integration of a function *f* over [*a*,*b*[on *N* processes

```
int main(int argc, char** argv) {
    // do some initialization
    ...
    // find interval for this process
    double interval=(b-a)/total;
    double start=a+interval*num;
    double end=start+interval;
    // partial integral between [start,end[
    double partial=integrate(sin,start,end,steps/total);

    // sum up partials
    double sum=0.;
    MPI_Allreduce(&partial,&sum,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);

    // print and finish
    ...
}
```


Master - Slave style

- One process, the Master distributes tasks: integrate2.cpp
- Other processes (slaves) ask for tasks and perform them

```
void master()
{
    // find tasks & distribute them
    ...
}
void slave()
{
    // ask master for tasks & perform them
    ...
}

int main(int argc, char** argv) {
    ... // (init)
    if (rank==0)
        master();
    else
        slave();
    ...
}
```

- Master and slave can run different programs!

MPI is much more than this

- We didn't touch:
  - Asynchronous, Non-blocking communication
    - very important to overly communication and computation
  - One-sided communication
  - Custom datatypes
  - Communicator subgroups
  - etc.

- More advanced lecture will have a more detailed view
  - High Performance Computing for Science and Engineering

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
- Open MPI explains how to use debuggers (gdb, totalview)
  - http://www.open-mpi.org/faq/?category=debugging

- Hints
  - first write a working serial program
  - Parallelize it and run it one one process first
  - two processes next
  - …
- **Good luck!!!**

Scaling with number of processes: Amdahl's law

- The sequential, non-parallel part will dominate the CPU time!
  - Assume *N* processes
  - on one process: $T_1 = T_{serial} + T_{parallel}$
  - on N processes: $T_N = T_{serial} + T_{parallel}/N + T_{communication}(N)$
  - define serial ratio $s = T_{serial}/T_1$
- Reduce serial parts
  - The optimum speedup would be

$$\text{speedup} = \frac{\mathbf{T_1}}{\mathbf{sT_1 + (1-s)T_1/N + T_{communication}}} \leq \frac{\mathbf{T_1}}{\mathbf{sT_1 + (1-s)T_1/N}} \leq \frac{\mathbf{1}}{\mathbf{s}}$$

  - *even if 1% is serial it does not scale well beyond 100 processes!*
    current machines have >10000 processes!
- 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!