RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Introduction to High Performance Computing

Lecture 02 – Introduction to Message Passing

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

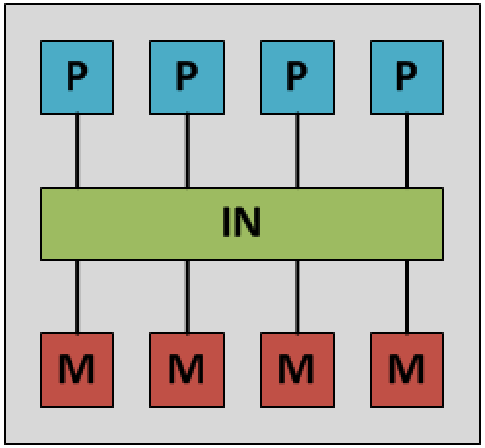
Concepts of Parallel Architectures

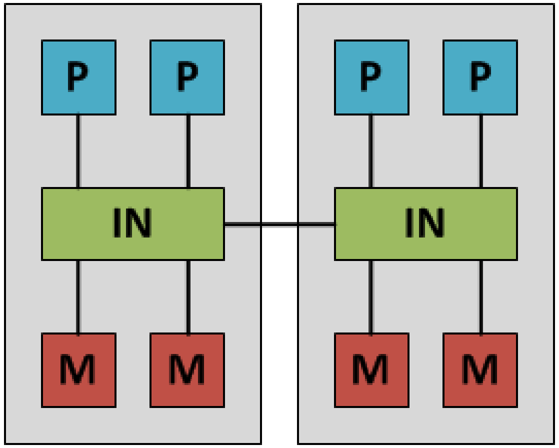
Concepts of Parallel Architectures

- Parallel Architectures
  - Are pervasive: smartphones, netbooks, notebooks, …
  - Most important resources for HPC: computing units, memory resources, interconnection network
  - Most common architecture found today: multi-core systems
- Multi-core system consists of:
  - Multiple computing/processing units, either:
    - Within one processor die (multi-core)
    - Within one computing system (SMP, multiple sockets)
  - One single or multiple memory controllers
- There are any more architecture types:
  - Most important: **Shared-Memory** and **Distributed-Memory**

Shared Memory

- Shared address space
  - Each computing unit can access any memory address
  - Physical memory may be distributed, access times may vary
    - UMA (uniform memory access)
    - NUMA (non-uniform memory access)
  - Any cache can hold copies from any location
    - Cache coherent NUMA (ccNUMA)
    - Usually limited scalabilty





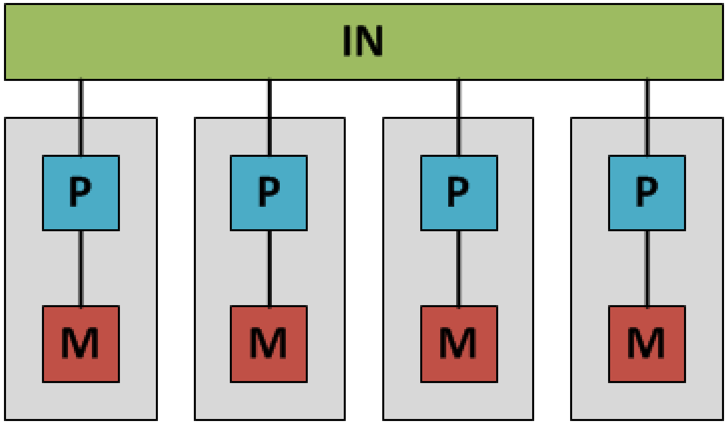
An address space defines a range of discrete addresses; each address may correspond to a different resource

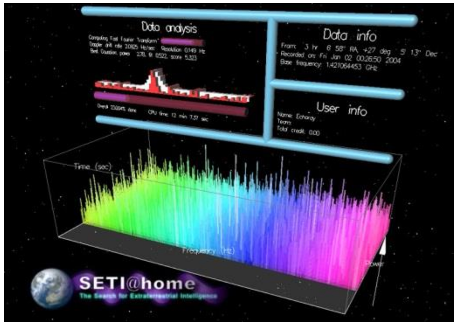
Shared Memory

- Shared variables
  - Operations, naming, ordering
- Communication purposes
  - Each unit can read/write any address
  - Used for data
  - Hazards and race conditions require synchronization
- Synchronization purposes
  - Resolve hazards and race conditions
  - Semaphores, condition variables, mutexes, locks
- Implicit communication, explicit synchronization

Distributed Memory

- Exclusive address spaces
  - Operations, naming, ordering
  - No (direct) remote access possible (NORMA)
  - Physical memory is distributed
  - Usually shared memory systems as building blocks
  - No global cache coherency required
    - Unlimited scalability (from an architectural point of view)
  - Popular example: SETI@Home





Distributed Memory

- Messages for communication purposes
  - Data transfer (copies)
  - Read example: send a request, receive an answer
  - Write example: send a request, optional answer
- Messages for synchronization purposes
  - On the origin node, one process is sending the message
  - On the target node, one process is receiving the message
  - Both will be informed when the other has processed the message
- Error-prone programming
- Programmer is responsible for:
  - Data distribution
  - Explicit communication
  - Explicit synchronization
  - Many optimizations (huge set of messaging functions, …)

Shared vs. Distributed Memory

- Key distinction: number of address spaces
  - Is (direct) access to the complete memory possible?

	Shared Memory	Distributed Memory
Number of address spaces	1	N
Communication	Implicit	Explicit
Synchronization	Explicit	Explicit
Number of data copies	Typ. 1	Typ. [1..N]
Scalability	Limited (cache coherence)	Unlimited/High

Synchronization is the most common error source!

Main reason for HPC

Example

- A human discussion as joint work example:
  - Let‘s assume they use a sheet of paper for the discussion
- If everybody is in the same room, one single sheet of paper can be used
  - Shared resource, but synchronization required
    - Use of pencil and eraser simultaneously, multiple instances…
- If the persons are physically distributed without access to the „local“ sheet, this sheet has to be sent around
  - Messaging
    - Only one copy – weak performance, little synchronization overhead
    - Multiple copies – improved perf., lot‘s of synchronization
- For both approaches, partitioning is an essential optimization

HPC

- HPC favors distributed memory
  - Scalability
- Key tool for HPC today is message passing
- Much more than send and receive
  - Plenty of send variants, plenty of receive variants
  - Collective communication
  - Even communication without saying *receive* (remote memory access, RMA)
  - More in later lectures…

Basics of Message Passing

Parallel Programming

- Nobody wants to write ***N*** programs for ***N*** processes
- Single Program Multiple Data (SPMD) paradigm
  - Write one program, which is internally partitioned to act differently
    - Master/Slave(s), Consumer(s)/Producer(s), Peer model
  - Before execution, each instance is assigned a unique number
    - Let's call this „***rank***“
    - Let's call the total number of ranks „***size***“
  - During execution, the rank determines program behavior
    - Functionality
    - Associated input – better output – data
- SPMD widely used for shared-memory and distributed-memory programming

SPMD Examples

- Master/Slave

```
if ( rank == 0 ) {
   act_as_master();
} else {
   act_as_slave();
}
```

- Peer model

```
int work[n];
for ( i = rank/size * n;
        i < (rank+1)/size * n;
        i ++ ) {
   process ( work[i] );
}
```

Note: (n % size == 0) is not mandatory

SPMD and Distributed Memory

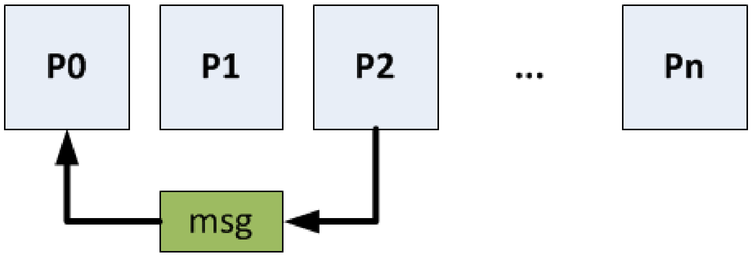
- Distributed Memory programs run locally with local data
- Everything explicit
  - Communication
  - Synchronization
  - Data distribution

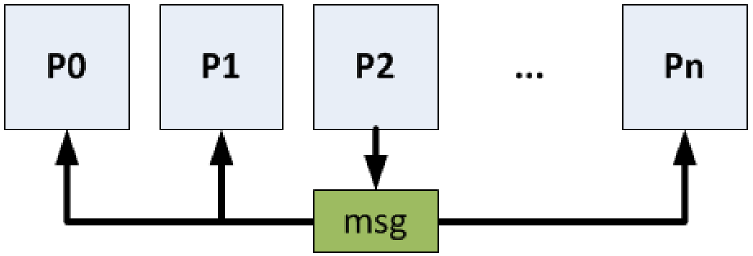
done using messages

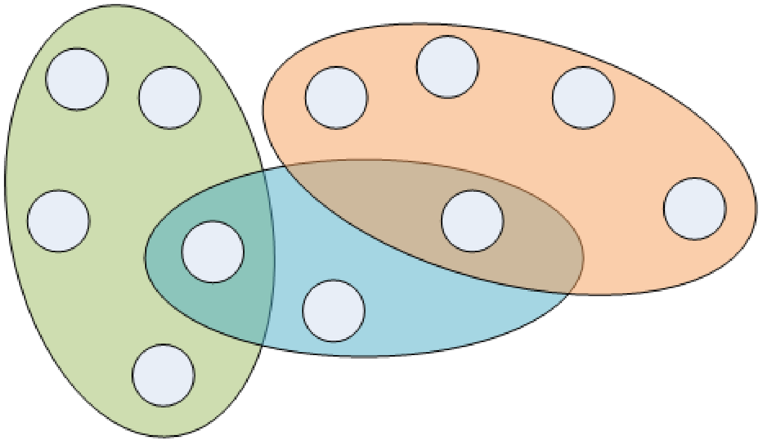
- Complicated programming
  - Not compatible with shared-memory programming
  - Code porting
  - Development of new algorithms
- Some help
  - Messaging libraries

Messaging Overview

- Point-to-point operations
  - Data movement between two peers
- Collective operations
  - Collective data movement (among peers or master/slaves)
  - Barriers (among peers)
- Communicators
  - Define process groups
  - Ranks / size of a communicator
  - Every message is assigned to one communicator
  - Collective operations refer to all ranks within one communicator







Message Details

- A message consists of the following
  - Source
  - Destination
  - Tag
  - Size
  - Data (optional)
  - Type

- Send operation
  - Provide all these information
- Receive operation
  1. Receive any message
  2. Receive any message from a **given source**
  3. Receive any message with a **given tag**
  4. Receive any message from a **given source** with a **given tag**

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Introduction to MPI

Message Passing Interface (MPI)

- De-facto standard for message passing in HPC
  - MPI 1.0 in 1994, to MPI 2.2 in 2009
- Library for C, Fortran, C++
- http://mpi-forum.org

- Plenty of functions – here only a couple
  - Mandatory are about 10-20

- One predefined communicator
  - MPI_COMM_WORLD (containing all processes)

Basic functions

- Always include MPI header file

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

- Initialize library
  - Call prior to any other MPI function

```
MPI_Init(&argc,&argv);
```

- Finalize library
  - Call after all MPI calls are done

```
MPI_Finalize();
```

- Get rank (unique ID) and size (number of processes) of current process for a given communicator (here default)

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

„Hello parallel world!“

```
// A very short MPI test routine to show the involved hosts
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
   int size,rank;
   char hostname[50];
   MPI_Init ( &argc, &argv );

   MPI_Comm_rank ( MPI_COMM_WORLD, &rank);  // Who am I?
   MPI_Comm_size ( MPI_COMM_WORLD, &size ); // How many processes?

   gethostname (hostname, 50 );
   printf ("Hello Parallel World! I'm process %2d out of %2d (%s)\n",
      rank, size, hostname );

   MPI_Finalize ();
   return 0;
}
```

„Hello parallel world!“

- Compile:

```
mpicc -Wall mpi_hello.c -o mpi_hello
```

  - Any additional compilation parameters (-O3, -L, -l, -h, …)
  - See also `mpic++, mpicxx`
- Prepare execution (only once per user):
  - MPI is usually based on ssh-connections. Ensure that no password is required for remote login:

```
ssh-keygen -t dsa -b 1024
cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

„Hello parallel world!“

- Execute, simplest flavor, 2 processes on one node:

  mpirun -host creek01[,creek01] -np 2 ./mpi_hello
- Execute, 16 processes on two nodes:

  mpirun -host creek01,creek02 -np 16 ./mpi_hello

- Parameters:
  - **-np <n>**: start n processes
  - **-host**: list of hostnames, used cyclic if more processes than entries in this list
  - **-mca …**: parameters to pass to the MPI library, here use TCP for communication (-mca btl tcp,self)

„Hello parallel world!“

```
Output:

# mpirun -host creek01,creek02 -np 4 ./mpi_hello
Hello Parallel World! I'm process  0 out of  4 (creek01)
Hello Parallel World! I'm process  2 out of  4 (creek01)
Hello Parallel World! I'm process  1 out of  4 (creek02)
Hello Parallel World! I'm process  3 out of  4 (creek02)


# mpirun -host creek01,creek01,creek02,creek02 -np 4
  ./mpi_hello | sort
Hello Parallel World! I'm process  0 out of  4 (creek01)
Hello Parallel World! I'm process  1 out of  4 (creek01)
Hello Parallel World! I'm process  2 out of  4 (creek02)
Hello Parallel World! I'm process  3 out of  4 (creek02)
```

A first message…

- (Blocking) send of a message

```
int MPI_Send(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
```

  - Example:

```
signal = 42;
MPI_Send ( &signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
```

- A blocking send returns if the provided buffer can be used again.
- Plenty of MPI_Datatypes

A first message…

- (Blocking) receive of a message

  int **MPI_Recv**(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

  - Example:

```
MPI_Recv ( &signal, 1, MPI_INT, i, 0,
                MPI_COMM_WORLD, &status );
```

- A blocking receive returns if a message with the required properties has been received (source, tag)
  - MPI_ANY_SOURCE
  - MPI_ANY_TAG
- Status contains more information about the receive
  - MPI_STATUS_IGNORE

Communication example

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv)
{
   int i, size, rank, signal;
   MPI_Status status;
   double starttime, endtime;

   MPI_Init (&argc, &argv);   //Initialisation of MPI
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);

   if (rank == 0) { //Master: send to every proc and wait for answer
      <see 2nd slide>
   } else { //Clients: wait for message from master and send back
      <see 3rd slide>
   }
   MPI_Finalize(); //Deinitialisation of MPI
   return 0;
}
```

Communication example

```
if (rank == 0) {

    signal = 666;
    starttime=MPI_Wtime();
    for (i=1; i<size; i++) {
       MPI_Send(&signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
       printf ("%d: Sent to %d\n", rank, i);
       MPI_Recv (&signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                 &status);
       printf ("%d: Received from %d\n", rank, i);
    }
    endtime=MPI_Wtime();
    printf ("%d: Time passed: %f\n", rank, endtime-starttime);
}
```

`double MPI_Wtime()`

Returns time in seconds since an arbitrary time in the past

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Communication example

```
//Clients: wait for message from master and send back
else {
   MPI_Recv (&signal, 1, MPI_INT, MPI_ANY_SOURCE, 0,
             MPI_COMM_WORLD, &status);
   printf ("%d: Received from %d\n", rank, status.MPI_SOURCE);
   MPI_Send(&signal, 1, MPI_INT, status.MPI_SOURCE, 0,
            MPI_COMM_WORLD);
   printf ("%d: Sent to %d\n", rank, status.MPI_SOURCE);
}
```

Blocking vs. Non-blocking

- Blocking
  - Returns after send or receive complete
  - What means complete?
- Non-blocking variants
  - Guarantees nothing, may return immediately
  - Used for improved overlap between computation and communication

```
int MPI_Isend(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm,
MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm,
MPI_Request *request)
```

Non-blocking: test/wait for completion

- Blocking: Wait

  ```
  int MPI_Wait(MPI_Request *request, MPI_Status
  *status)
  ```

  - Guarantees that request has been processed
- Non-Blocking: Test

  ```
  int MPI_Test(MPI_Request *request, int *flag,
  MPI_Status *status)
  ```

  - Updates the request, but does not ensure completion

Questions?

Next lecture:

Basics